Multi-mode Redundancy Removal

Stephen M. Plaza[1]#, Prashant Saxena^, Thomas Shiple+, Pei-Hsin Ho^
#Janelia Farm Research Campus, 19700 Helix Drive, Ashburn, VA, USA
^Synopsys Inc., 2025 NW Cornelius Pass Rd, Hillsboro, OR, USA
+Synopsys, 700 East Middlefield Road, Mountain View, CA, USA
plazas@janelia.hhmi.org, {saxena,shiple,pho}@synopsys.com

ABSTRACT

Redundancy removal, *i.e.*, identifying and eliminating redundant logic, is an essential optimization strategy for decreasing design area, reducing critical path delay, and simplifying circuit testability analysis. However, redundancy removal strategies invoke time-consuming proof engines with worst-case exponential behavior. While continual enhancements to heuristics resident in these engines result in large average runtime improvements, inherent intractability still leads to sub-optimal optimization and occasional large runtime outliers. Such outliers are unacceptable in an industrial setting where an outlier compromises design turnaround time. Our work introduces a *semi-local* optimization algorithm that mitigates the inherent intractability in redundancy removal and eliminates crippling runtime outliers. We further embed this algorithm within a framework that minimizes any negative impact to delay and area metrics. Using the cutting-edge Synopsys® Design Compiler® logic synthesis tool, we demonstrate 1) statistical neutrality in area and timing while achieving consistent runtime improvements on a large set of proprietary circuit designs of varying type and complexity and 2) over 50% runtime improvement on a suite of computationally intractable industrial designs, even when measured at the end of the physical synthesis flow.

1. INTRODUCTION

An RTL design often contains internal pins whose logical functionality does not impact the input/output behavior of the design. Consider the example in Figure 1, where the output *E* of the circuit does not depend on signal *d*, and, in turn, primary input *A*. Eliminating this redundant logic reduces area, simplifies the netlist for downstream optimization, and improves the testability of the circuit. Although it is often carried out independent of physical optimization, redundancy removal can also often improve timing by reducing the capacitive load on a pin. To exploit these simplification opportunities, logic synthesis methodologies typically invoke redundancy removal (RR) as an integral aspect of logic optimization.

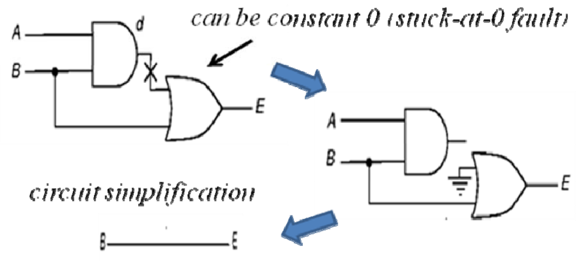


Figure 1. Identifying and eliminating redundancy in a circuit.

RR is an NP-complete problem and is inherently global in scope since the primary inputs/outputs are needed to prove that a pin is not redundant. Therefore, RR can result in prohibitive runtime. A logic synthesis methodology which uses RR can counteract this problem by deploying RR conservatively within the flow. While the focus of our paper will be on improving the underlying RR engine, understanding how RR is typically called is vital to designing an engine that achieves a good balance between optimization quality and algorithmic runtime. In particular, chip designers typically create component hierarchies, which confine the scope of RR and other logic transformations, mitigating the large runtime that would otherwise occur on a flat hierarchy. However, because synthesis operations such as remapping and rewriting can introduce new redundancies, RR can be called multiple times in an optimization flow. Therefore, any approach that speeds up an RR engine without compromising on the number of redundancies detected by it can improve the overall runtime of the entire logic synthesis flow significantly.

RR algorithms are traditionally based on ATPG search algorithms [7], which derive a sensitizing input pattern, if one exists, for each pin in the design. Continual advances in ATPG algorithms [5,7,11,12] have led to significant runtime improvements allowing most practical circuit designs to be analyzed in polynomial time. Additional innovation in satisfiability solving in both clause learning [10] and fast implications procedure [8] further improves RR algorithms [12]. In a related application domain, the authors in [1] achieve a significant speedup in redundancy addition and removal (RAR) optimization by directly finding candidates for redundancy addition rather than performing time-consuming trial-and-error identification of redundant wires. The cited approaches achieve considerable average-case improvement within their respective application domains (and can complement the techniques introduced in this paper). However, operating within an industrial framework uncovers a challenge not adequately addressed in previous work: avoidance of *any* runtime outlier. The existence of one outlier can severely hinder design turnaround time that can prevent time-to-market goals.

In general, algorithms that solve NP-complete problems, such as proving redundant faults, can lead to runtime outliers orders-of-magnitude larger than the average over a set of different circuits. One potential solution to contain this variability uses a portfolio of different algorithms running on multiple processors to solve the same problem instance [3]. The parallel task completes after the first algorithm finishes. However, this portfolio consumes multiple computational resources, while providing no guarantee that an efficient algorithm exists. Our approach addresses the fundamental challenge of the inherent and difficult-to-estimate complexity of RR problem instances, by dynamically pruning the

[1] This work was carried out while Stephen Plaza was with Synopsys, Inc., Hillsboro, OR, USA.

problem space in a manner that gracefully trades off the quality of optimization against improved runtime.

In this paper, we observe that most redundancies can be proven considering only a small subset of logic surrounding the analyzed pin. Therefore, we introduce a new "semi-local" RR algorithm that dramatically reduces the number of decisions made in the underlying ATPG-based engine. On a large industry suite, we first reduce the percentage of RR computation by 39% (leading to a significant improvement in the runtime of the entire logic synthesis flow) at the cost of a small end-of-flow slack and area degradation. Next, we eliminate even this small degradation by generalizing our new semi-local algorithm to an adaptive multi-mode framework, without sacrificing the runtime benefits of the semi-local algorithm. This multi-mode RR engine relies on our observation that only a small percentage of circuit designs require prohibitive runtime to perform RR. It exploits the infrequency of these runtime outliers by dynamically adjusting its aggressiveness based on its current effectiveness on that design and the intrinsic properties of the design. In this manner, our multi-mode engine 1) finds >99% of the redundancies, 2) is area and timing neutral on average, and 3) significantly reduces the runtime of intractable RR outlier cases without compromising their quality metrics. Our techniques are far more sophisticated than the widely used straightforward computational cut-offs that stop examining a potentially redundant fault when computation is prohibitive. If the cut-off is very low, several redundancies will be missed. Even if the cut-off is carefully chosen to avoid large outliers in examining a fault site, repeatedly reaching the cut-off, when averaged over the whole design, will lead to large global runtime increases. Our techniques avoid these problems by combining a global budget for computation (in addition to a local cut-off) with an RR heuristic that constructs smaller problem instances so that the total computation can better fit within this runtime budget while still being able to maximize the number of redundancies discovered. Consequently, we eliminate runtime outliers and minimally impact optimization quality.

Our contributions include:

1) Revealing the sensitivity of design optimization to thorough redundancy identification and elimination.
2) Improving RR runtime by restricting decisions to small sections of logic that are likely to prove a pin redundant.
3) Proposing a multi-mode redundancy removal engine that dynamically varies the effort required to prove a pin redundant in a manner that maximizes the optimization quality for a target global runtime budget.
4) Deploying our multi-mode engine to exploit the typical behavior of redundancy removal within the leading-edge industrial tool Synopsys® Design Compiler®, and demonstrating its effectiveness on a large suite of real-world designs.

In Section 2, we discuss the underlying algorithm in RR, previous work, and runtime bottlenecks. Section 3 introduces our faster RR engine. Section 4 proposes a multi-mode RR engine to enable effective deployment in industrial tool flows. Finally, Section 5 highlights area, timing, and runtime results on a comprehensive industrial benchmark suite.

2. BACKGROUND

In the following section, we will outline the major components of an effective RR engine [7]. We will then discuss some of the key heuristics that determine the effectiveness of the RR engine. With this background, we examine the key runtime bottlenecks in RR and describe some prior work to address these bottlenecks.

2.1 RR Framework

The work in [7] describes a straightforward and effective ATPG-based framework that identifies and removes redundancies as a byproduct. Figure 2 outlines major components of the framework:

```
eliminateRedundancies(Circuit C)
{
  random_simulate_and_detect(C);
  for_each(C, pin) {
    if(undetected(pin, C)) {
      detectFault(pin));
    }
  }
}
detectFault(Circuit C, Pin pin)
{
  if (pattern = findSensitizingPattern(pin)) {
    simulate_and_detect(pattern, C)
  } else {
    replace_with_constant(pin);
    propagate_constraints(C);
  }
}
```

Figure 2. Framework for redundancy removal.

RR begins by applying random input patterns to the primary inputs of the circuit with *random_simulate_and_detect*(). These random patterns can identify non-redundant pins by checking whether a given pin's logic impacts the output values for a simulation pattern. In testing terminology, such a pin is called *testable,* meaning that a *stuck-at 0* or *stuck-at 1* fault is *detectable*. By executing this procedure, typically 80-90% of all pins are detectable, so that an explicit proof does not need to be generated to determine their redundancy. If random simulation fails to detect a sensitizing input pattern, a formal proof engine, *findSensitizingPattern*, is called. This function will be discussed in greater detail throughout this paper. If the pin is not redundant, a counter-example is derived in the form of a sensitizing input pattern. This *dynamic simulation* can detect stuck-at faults at other pins, thereby limiting the number of calls to the proof engine. If a redundancy is detected, that redundancy is replaced with a constant *1* or *0* (as the case may be) and this constant value is propagated through the circuit to simplify the surrounding logic.

Properly tuning the decision procedure is critical to the runtime performance of *findSensitizingPattern*. Previous research in RR is largely focused on determining an effective decision procedure [2,3,4,6,10]. The *D-algorithm* [2] focuses on propagating faults to the primary outputs before performing implication. This can result in needless backtracking and decisions during the propagation procedure. The *PODEM* algorithm [4] is a goal-based decision engine, which makes decisions on the primary inputs to justify certain values. By making decisions on the primary inputs, the decision engine is often far removed from the location of the pin being analyzed.

The decision procedure in [7] used in *findSensitizingPattern* combines aspects of different decision heuristics. The propagation procedure starts by asserting two values, 0 and 1, and then decides on logic values in the fanout cone of the pin, so that the difference propagates to the primary outputs. The implications of these decisions are also fully made. Justifying the values used to propagate the fault then requires finding satisfying primary input values. The works in [5,11] explore different combinations of decision strategies to achieve good RR runtime.

2.2 Runtime Bottlenecks in RR Framework

The largest runtime bottleneck in the framework presented in Figure 2 is the function *findSensitizingPattern* which solves an NP-complete problem. In our experience, this is greater than 50% of RR runtime. The simulation and subsequent fault detection performed in *random_simulate_and_detect* and *simulate_and_detect* require worst-case $O(n^2k)$ for k input patterns and n logic pins in the circuit. This worst-case runtime can be improved in practice by simulating multiple patterns simultaneously using *bit-parallel simulation* as in [6,7], although the use of bit-parallel operations (such as & and |, in C syntax), requires representing the circuit with simple logic primitives. In practice, improving the quality of the dynamic and random input patterns can significantly limit the number of time-consuming calls to *findSensitizingPattern*. However, because the best-case runtime to detect the fault of every pin in the circuit is $O(n^2)$, too much simulation can be counter-productive.

To contain the runtime of outlier cases, it is common to implement time-outs in the proof engine [14]. For instance, when the number of decisions made in the proof engine exceeds a certain threshold, the function *findSensitizingPattern* returns incomplete and the pin's status remains unknown. The drawback of such context-insensitive timeout mechanisms is that the limit is not set in a manner that gives more flexibility for cases when a redundancy is likely to be found with additional computation and less flexibility when it is unlikely that a redundancy exists at all. Our work leverages timeouts in a context sensitive way that strikes this balance between finding redundancies and being runtime efficient.

A main contribution of our paper is the introduction of a semi-local algorithm that avoids computation that is not necessary to prove a fault redundant by restricting the amount fanout logic considered in the proof. We experimented with but then abandoned a more naive windowing approach that restricts both the fanin and fanout of a fault (or a conceptually similar circuit partitioning traditionally used to combat complexity in global-based algorithms) since the semi-local strategy more effectively trades off quality and runtime. Restricting the fanout cone through our semi-local algorithm often has the greatest impact on the size of the RR instance because the ignored fanout and its transitive fanin cone is typically larger than just the fanin cone of the fault site. Our semi-local algorithm possesses many nice properties not realizable in partitioning/windowing strategies: 1) random and dynamic simulation applied at the circuit level are consistent with the controllability of the semi-local regions but not with windowed regions, 2) counter-examples found with the semi-local algorithm are relevant to parts of the circuit controllable by them, and 3) the data structures in the underlying RR engine can be tailored to perform justification more efficiently than propagation, since propagation is trivial in the semi-local approach.

The work in [5] on Single-Path-Oriented Fault Effect Propagation (SPOP) has indicated potential runtime savings by avoiding the justification procedure. This idea can complement our approach and yield further runtime gains. However, we have observed that this complementary approach identifies fewer redundancies, which leads to unacceptable area and timing degradation for the limited runtime improvement. The algorithms in [11] expand on [5] by using static learning techniques and decision heuristics to potentially identify redundancies more efficiently. It should be noted that heuristics which avoid decision procedures and simply propagate implications until a potential conflict (and therefore proof for redundancy) is found can result in prohibitive runtime if the set of implications is large.

We share some conceptual similarities to the algorithms in [9,13] that consider small sections of output logic to determine logic equivalences in the netlists. However, there are key differences and limitations in those approaches when compared to our approach. Firstly, the algorithms in [9] and [13] focus on merging equivalent nodes rather than the more focused task of removing redundant nodes. As such, their emphasis is on SAT-based engines finding equivalences rather than ATPG-based engines that identify redundancies. Furthermore, neither of these works develops any framework that dynamically adjusts the strength of the underlying engine for a given context. By dynamically adjusting our algorithm, we find many more redundancies while limiting runtime increases.

3. SEMI-LOCAL RR

We first introduce an algorithm to substantially reduce RR runtime while identifying most redundancies. The key aspect of our approach eliminates decisions and logic implications past a certain logic depth in the fanout cone. Our techniques are general enough to be applicable to and complement different decision strategies. Another aspect of our work is viewing RR as a standalone problem that is not merely a subset of ATPG. Although ATPG (or SAT-based) techniques are vital to current RR methodology, finding detectable vectors is only beneficial if it results in runtime savings.

The following subsections motivate and introduce the semi-local algorithm.

3.1 Motivation

When propagating a fault to the primary outputs, we observe that often only a few levels of downstream logic levels are necessary to prove that the pin is redundant. To qualify this intuition, consider the simple circuit depicted in Figure 3. In this example, the goal is to determine whether pin *a* is redundant. First, both possible logic values are assumed at the *fault site* of *a* and then non-controlling values are chosen for the side inputs *b*, *c*, and *d,* to propagate this difference to the output of the circuit. With an AND chain of 3, as depicted Figure 3, only 1/8 of random input patterns sensitizes the fault. The probability that a random input pattern sensitizes the fault decreases exponentially with the depth of the chain. Redundancies exist if some input combinations are not possible due to reconvergence or other limits on the controllability of these side inputs. Conceptually, the probability that a redundancy is provable within the first few downstream logic levels (a few levels of the AND chain in this example) is much greater than a proof requiring more logic levels.

The intuition for considering only a few levels of downstream logic is strengthened when accounting for fanout. From a given pin, several disjoint paths may exist to a number of primary outputs. If the difference that needs to be propagated resides on several different sensitization paths, the probability that a difference will reach a primary output is high. Furthermore, because multiple sensitized paths are considered, the runtime required to prove a redundancy increases. Our approach avoids the adverse impact of these high fanout situations.

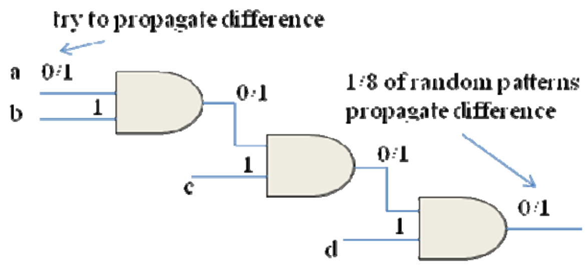


Figure 3. Most redundancies need only a few levels of downstream logic for proof.

3.2 Algorithm

Our algorithm starts from the fault site and statically analyzes the logic topology. We create a window that captures *n* levels of fanout from the fault site, along with the entire fanin cone for these fanouts. Because the window is unbounded on the input side, we refer to our technique as being *semi-local*. When we count the *n* logic levels, we consider both the absolute number of logic levels (with respect to the whole circuit) and the relative level with respect to the fault site. Thus, for example, if the fault site has two fanouts, one on level 4 and the other on level 9, we consider the inclusion of both levels as two levels of logic from the fault site, even though they are both one level from the fault site. At an intuitive level, we wish to avoid situations where unnecessarily large sections of the logic need to be analyzed to ensure there is no reconvergence. In our implementation, buffers and inverters are ignored since they always transmit logic differences. Finally, reconvergent nodes are also not included in the levelization count since in many cases, the reconvergent node increases the probability that a fault will not propagate further downstream.

Figure 4 illustrates our algorithm at an abstract level. Note that in the original algorithm depicted at the top, the entire fanout cone and the fanin cone of this fanout cone could be examined to prove a redundancy. In our approach depicted at the bottom, we see a dramatically restricted fanout cone, which, in turn, significantly shrinks the fanin cone that needs to be analyzed. Although this algorithm does not lead to asymptotic speedup, the complexity of the propagation procedure is reduced to being approximately a constant if the implications of the decisions are not propagated on the input side. We experimented with restrictions of the logic window on the fanin side of the fault site also, but the degradation of optimization quality, coupled with the small runtime gains, made such restrictions undesirable.

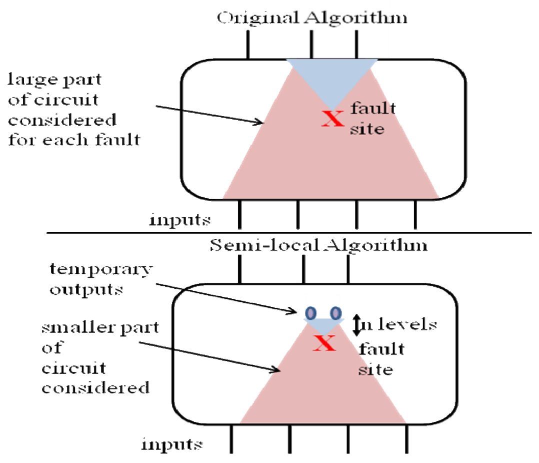


Figure 4. Original algorithm (top) compared to our new semi-local algorithm (bottom).

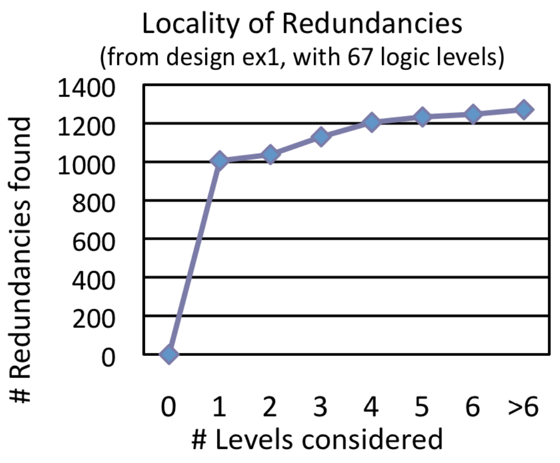


Figure 5. The number of redundancies proven within *n* downstream logic levels.

Figure 5 provides an example of how varying the value of *n* (i.e., the upper bound on the number of propagation levels, as described earlier) impacts the number of redundancies found. Notice that most of the redundancies require only one level of downstream logic in order to be identified. Although this figure presents the data for merely one design, the general trend depicted here holds true for almost all the designs in our benchmark suite. Therefore, in our experiments, we only consider values of *n* under 5.

4. MULTI-MODE REDUNDANCY REMOVAL

In the previous section, we introduced a semi-local algorithm to more efficiently identify redundancies. We show in Section 5 that although this algorithm is very fast and finds a large percentage of redundancies (as posited) compared to the baseline, there is some undesirable variability in the area and timing metrics. In this section, we propose a multi-mode RR framework that automatically invokes the semi-local algorithm only on those problem instances that are dynamically identified as outliers. In practice, we observe that RR runtime is linearly related to the design size on most instances. The infrequency of outlier instances does not diminish the importance of containing them, as they can easily cripple design turnaround time on that *one* design that a designer really cares about, if it unfortunately happens to be a pathological case for the RR engine.

We first present some data that motivates the use of different RR modes. Then, we show a RR framework that effectively eliminates runtime outliers by combining an adaptable semi-local algorithm to a global runtime budget such that the computation performed maximizes the number of redundancies identified.

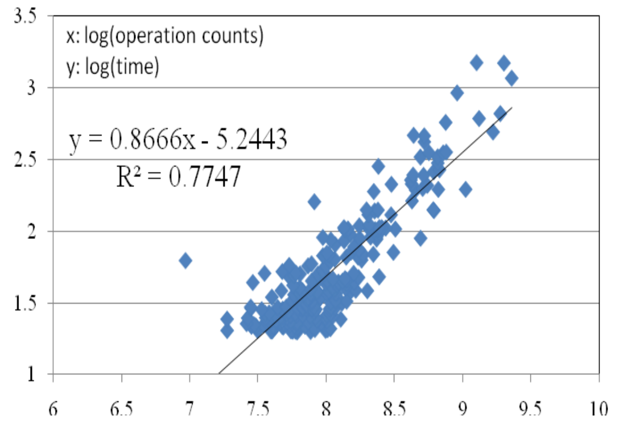


Figure 6. Accuracy of a simple reproducible operation counter and the prevalence of simple RR instances.

4.1 RR Runtime Characteristics

We analyzed the performance of the RR engine (without invoking the semi-local algorithm) over hundreds of different problem instances over a diverse collection of industrial designs. We filtered out the thousands of very short RR calls (under 20 seconds) that occurred in our experimental flows. Instead of estimating the complexity of a particular instance by CPU runtime (that can fluctuate from run to run), we implemented a simple operation counter. Fine-grained operation counters have been proposed in the past [14], since such counters allow reproducibility in the results independent of the architecture that the underlying RR engine is executed on. Figure 6 is a scatter plot of these different RR calls with the y-axis as the CPU runtime and the x-axis as the operation counter. This chart provides a quick validation of the reasonably tight correlation between actual runtime and our operational counter. Figure 6 also reveals that most of the RR calls are clumped in the lower left corner, indicating that most of the calls are pretty small. The upper half of the graph has a disproportionally smaller number of sample points. In the next subsection (4.2), we describe how to leverage this behavior by turning off the semi-local algorithm for the simplest RR calls as determined by the operation counter. However, in order to make our multi-mode RR strategy more robust, we also need to consider other design aspects as described next.

Figure 6 provides a straightforward criterion for separating the easier RR instances from the harder ones. Because our main goal is to reduce the percentage runtime devoted to RR in the synthesis flow, we consider a more specific distinguishing criterion in Figure 7. Here the x-axis is an empirically determined function of both the runtime needed to process each pin and the percentage of area of the design component with respect to the entire design. The y-axis gives the percentage of runtime in the synthesis flow dedicated to RR (indicated here by %RR). We find that the distribution of RR calls in Figure 7 is independent of the overall size of the circuit. This plot exposes the tendency of synthesis flows to invoke RR on each hierarchical component, rather than on flattened versions of the design that can vary tremendously in size. As a consequence, the worst runtime outliers require that the component have both large runtime per pin as well as large area compared to the entire design. The correlation of this function to %RR runtime is tight and highlights that RR is generally well-behaved and that the worst outliers are readily distinguishable.

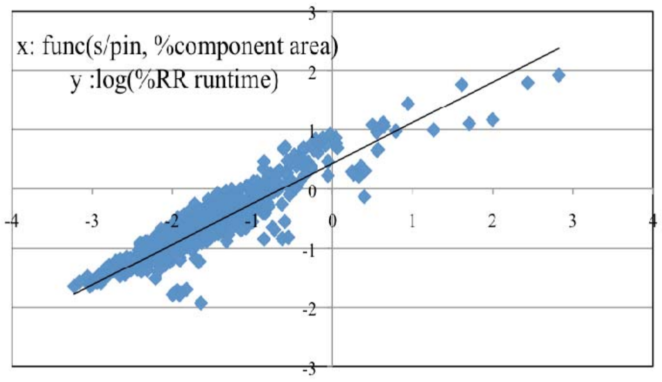


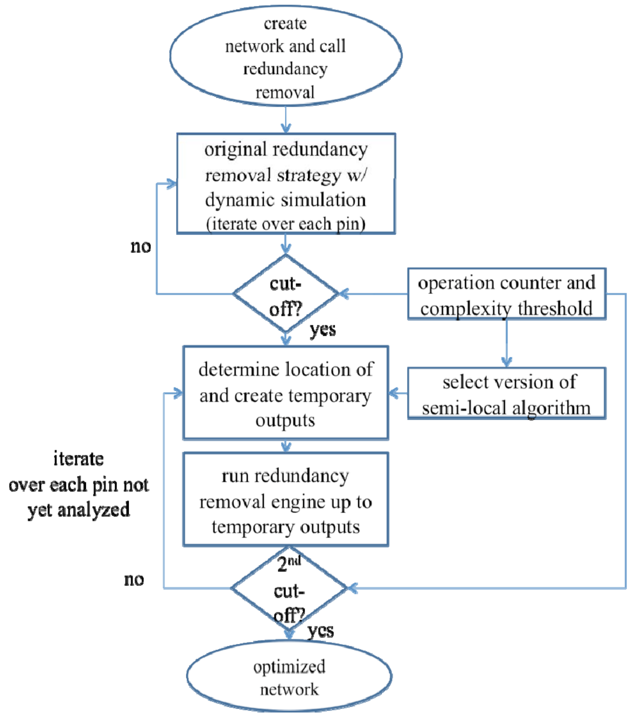
Figure 7. Scarcity of large runtime outliers and their separability from the remaining RR calls.

4.2 Multi-mode Strategy

The intuition gleaned in the previous subsection motivates a RR framework that can switch to different modes automatically to satisfy a global runtime budget. We propose a framework that adjusts the amount of downstream logic considered when proving whether a fault is redundant. It initially leverages both random and dynamic simulation without using the semi-local algorithm (i.e., n is considered to be ∞). Using the results in Figure 6, an aggressive cut-off is made to trigger the semi-local algorithm. We capitalize off the general robustness of the semi-local algorithm by erring on the side of runtime savings. This invocation of the semi-local algorithm uses a relatively small value of n. We then use the intuition behind Figure 7 to reduce the value of n even further when both the per-pin computation and overall computation pass certain thresholds. In the infrequent case when the semi-local algorithm results in a yet larger RR runtime, we abort the call to RR. Conceptually, Figure 7 with the circuit size provides a rough estimate for a global runtime budget that dictates the aggressiveness of the semi-local algorithm in a manner that allows graceful optimization degradation. Figure 8 below details our algorithm.

After creating a component network, we invoke RR using the algorithm outlined in Figure 2 (with random simulation and dynamic simulation). As we process the component design, we check whether an operational cut-off is reached. When this cut-off is reached, dynamic simulation is disabled and the semi-local mode is invoked with a relatively small value of n (typically, 5) by placing temporary outputs n levels from the fault site. Theoretically, dynamic simulation vectors can still be used with respect to these temporary outputs. However, the efficiency of performing dynamic simulation is often worse than running the semi-local mode.

As the engine continues to run, we check another threshold that is based on the size of the component and per pin operation count, in order to adjust the behavior of the semi-local algorithm dynamically. More precisely, we reduce *n* to one (implying that temporary outputs are only one level from the fault site) in our current flow when we reach the new threshold. If computation still persists for an empirically determined second cut-off, we abort the RR and return the optimized component network. As noted, our multi-mode framework rarely aborts without finishing.



create network and call redundancy removal
original redundancy removal strategy w/ dynamic simulation (iterate over each pin)
no
cut-off?
yes
operation counter and complexity threshold
determine location of and create temporary outputs
select version of semi-local algorithm
iterate over each pin not yet analyzed
run redundancy removal engine up to temporary outputs
no
2nd cut-off?
yes
optimized network

Figure 8. Multi-mode RR framework.

5. EMPIRICAL RESULTS

We implemented our multi-mode RR engine within Synopsys® Design Compiler® [15], a cutting-edge industrial logic synthesis tool. Our engine includes a semi-local algorithm that places temporary outputs in a circuit, an operation counter as an experimentally stable proxy for CPU runtime, and a framework that invokes different RR modes dynamically based on the complexity of the problem being solved. We test our framework by running Design Compiler using a comprehensive standardized logic synthesis methodology[2] that takes as input RTL and outputs a logically mapped netlist. The flow entails (among other optimizations), logic rewriting, substitution, and mapping. We evaluate the runtime changes of Design Compiler on an industrial design suite consisting of 110 designs for which the standard cell instance count ranges from 9K to 1.2M. The suite represents a diverse set of designs including controller logic, small pipelines, adders, etc. Each design in the suite invokes the RR engine multiple times for the different hierarchical components, interleaved between other synthesis transformations on those components. Each RR invocation iterates through the netlist at least twice, unless the first pass identified no redundancies.

We provide a robust analysis of our new RR techniques with respect to Design Compiler. To more accurately evaluate the impact of our redundancy removal on end-of-flow area and timing, we performed performance-driven placement and physical synthesis on the designs subsequent to logic optimization. We evaluate runtime improvements by considering relative improvement of RR runtime between our techniques and the ones in Design Compiler. Even though RR typically makes up a small portion of the logic synthesis runtime (except for the outliers we are targeting), we also report runtime improvements relative to the entire synthesis flow.

5.1 Semi-local Algorithm Results

Figure 9 and Figure 10 show the results obtained by running the RR engine only in the semi-local mode. These results indicate the effectiveness of the semi-local algorithm within our comprehensive synthesis methodology. Figure 9 reveals the runtime improvement obtained with respect to the entire logic synthesis flow using our faster RR engine. Improvements are denoted as negative values. The runtime of only the RR engine improves by 39% on the average, while still detecting 93% of the redundancies found by the original (unrestricted) RR engine. Recall that even the original RR engine will not necessarily find every redundancy, as internal cut-offs get triggered for complex RR calls. Without these cut-offs, the RR invocations (and consequently, the synthesis flow) would become intractable on many problem instances.

Even though the RR engine is merely one small component of Design Compiler, our faster engine leads to a significant improvement in the runtime of the overall synthesis flow (-2.3% average runtime improvement) as depicted in Figure 9. In other words, we are not trading runtime reduction in the RR engine for an increase elsewhere in the synthesis flow due to a changed optimization trajectory caused by the fewer redundancies found during RR.

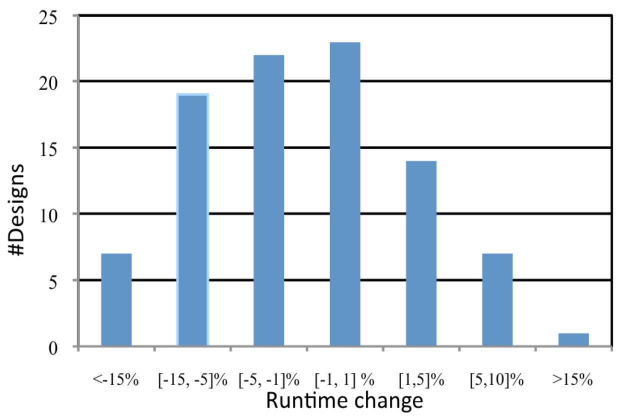


Figure 9. Runtime improvement of a comprehensive synthesis flow due to the pure semi-local algorithm (negative values indicate speedups).

[2] Although specific details of our synthesis tool and methodology are omitted in this paper, we provide a high-level description of the same. This tool and its associated methodology have been used successfully in numerous diverse customer settings for many years.

Figure 10 highlights the impacts of the synthesis flow using our semi-local strategy on the a) area and b) timing of the entire design. Despite finding 93% of redundancies compared to the baseline of $n=\infty$, there is a small degradation to these quality-based metrics. In these figures, degradation is denoted by positive values. Although the average timing degradation (measured using the Worst Negative Slack (WNS) metric normalized to the clock period in the industry-standard manner) is negligible (0.05%), there is one large outlier at >10%. There is also a small but discernable 0.15% average area degradation. Our goal is area and timing neutrality on average and the removal of all outliers; the next section will show how the multi-mode approach can enable us to meet these quality goals while still managing to automatically identify the intractable RR cases.

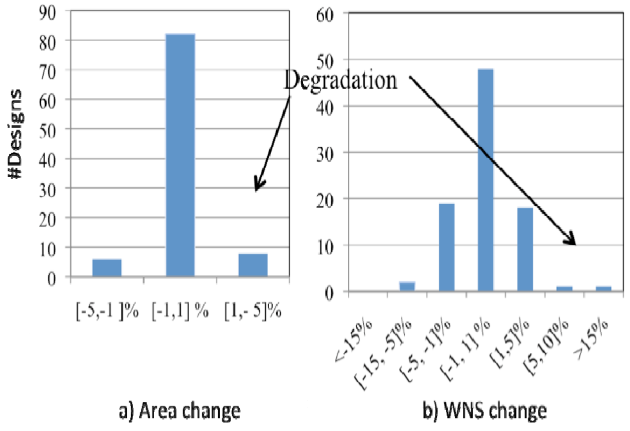


Figure 10. Performance of the comprehensive synthesis flow using the pure semi-local RR engine on the industrial design suite. a) Histogram showing an average end-of-flow area increase of 0.15%. b.) Histogram showing an average end-of-flow timing degradation of 0.05%, along with a couple of large outliers.

5.2 Multi-mode RR Results

In this section, we show that the multi-mode algorithm does not degrade area and timing on our comprehensive design suite, even as we still greatly reduce runtime outliers. Figure 11 shows that the a) area and b) timing of the synthesis flow using our new algorithm are very stable, with no statistically discernable degradation or significant outliers. Furthermore, the multi-mode engine found over 99% of the redundancies in the baseline. The multi-mode strategy only activates the semi-local algorithm on 25 designs in the entire 110 design suite, but there is still an absolute percentage RR runtime reduction of 2.11%, leading to a small improvement in the overall synthesis flow turnaround time, even when averaged across the entire suite. None of the RR calls in this suite terminated early because of reaching the second cut-off. Thus, the use of the multi-mode RR engine entails no loss of quality or runtime vis-à-vis the unrestricted engine. At the same time, this engine gives us the capability to automatically detect and degrade gracefully on the rare intractable problem instances, as detailed in the next section.

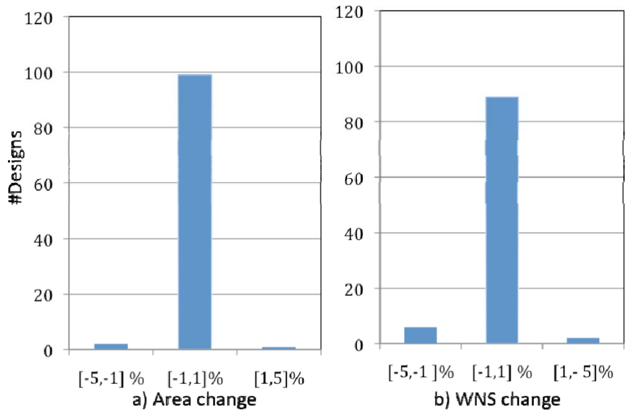


Figure 11. Performance of the comprehensive synthesis flow using the multi-mode RR engine on the industrial design suite: statistical neutrality on end-of-flow a) area and b) timing.

5.3 Elimination of Large RR Outliers

As mentioned in the introduction, there are occasional circumstances where RR becomes the dominant runtime bottleneck in the logic synthesis flow. We observe large variability in circuit characteristics among runtime outliers where small designs sometimes result in large runtimes. Failure to perform RR on these designs would lead to undesirable degradation in design quality, especially in the area metrics, but running unrestricted RR can result in unacceptable logic synthesis turnaround time. Our multi-mode RR algorithm improves the runtime of eliminating redundancy and enables these intractable designs to be processed. *It targets the worst outliers effectively in a manner that does not degrade the common case* (as described in the previous section).

Over the years, we have collected eight customer designs for which RR has been a major runtime bottleneck in our logic synthesis flow. These problematic designs have varying numbers of logic cells, as shown in Table 1. This table also shows the effectiveness of our RR framework. The first column gives a name for the design. The second column shows the absolute runtime for running a complete synthesis flow with the original RR algorithm, with the designs sorted from those requiring the largest runtime to those requiring the least. The third column shows the percentage of RR runtime for each design in the original flow. The fourth column shows the runtime improvement and the fifth column show the new percentage of RR. Notice that most designs achieve over a 2x improvement in overall tool runtime (and are shown in the table using lightly shaded rows). On average, there is 2.7x runtime improvement for the entire logic synthesis flow. Because the semi-local algorithm is still global in nature, there are some cases, such as design *c* and *f*, where the computation of the semi-local algorithm triggers time-outs reducing the magnitude of runtime savings. Design *h* achieves significant runtime savings using our approach; however, in this particular case the runtime of the other synthesis optimizations, such as rewriting and mapping, is comparatively small. This leads to only small improvement in the percentage of RR runtime, but still improves the overall synthesis flow turnaround time dramatically.

Table 1. Large runtime improvement for RR runtime outliers.

Design	Design Compiler (hrs)	%RR	Design Compiler Speedup	New %RR
a	32.8	82	4.2x	21
b	31.4	65	2.5x	15
c	29.2	67	2.0x	32
d	16.2	61	2.4x	5
e	7.1	39	1.3x	12
f	7.0	90	5.0x	47
g	3.3	57	1.8x	16
h	0.9	98	4.3x	91

Furthermore, in some cases, the multi-mode RR engine is able to find even more redundancies for these designs than the original unrestricted RR engine, resulting in end-of-flow area improvements for these designs. Although this might appear counter-intuitive, this is explained by the observation that even an unrestricted RR engine in a practical logic synthesis flow must eventually time-out on particularly intractable problem instances (since the underlying problem is NP-hard), and may thus miss processing many easy-to-find redundancies. In contrast, our multi-mode engine manages to process all the fault-sites in the design in most designs without getting trapped in any intractable proof.

6. CONCLUSIONS

We introduce a comprehensive redundancy removal algorithm and framework which we deploy effectively inside a cutting-edge industrial logic synthesis tool, namely, Synopsys Design Compiler. Firstly, we propose a semi-local algorithm that reduces the runtime of the RR engine by 39%, leading to large overall runtime improvements in a logic synthesis flow that invokes this RR engine. Our approach, that focuses on redundancies, contrasts with ATPG engines that find redundancies as a byproduct of identifying sensitizing vectors. Secondly, we introduce a dynamic multi-mode RR engine that leads to graceful degradation in optimization quality enabling a good runtime/quality tradeoff. Our results indicate a significant 2.7x runtime improvement for the logic synthesis flow for real-world outlier designs, and we observe no area or timing degradation or runtime penalty on a comprehensive industrial design suite, even when measured at the end of the physical synthesis flow. We believe our approach is broadly effective in applications using RR as we demonstrate robust results on a diverse design set and our semi-local algorithm can complement different types of underlying RR engines by focusing its computation on parts of the search-space that are likely to find a redundancy. Furthermore, our multi-mode strategy can effectively handle problem instances of arbitrary complexity since we dynamically adjust the size of instance as constrained by a global runtime budget in a way that maximizes the redundancies identified.

7. ACKNOWLEDGMENTS

We would like to thank Janet Olson for her invaluable support and guidance of this project.

8. REFERENCES

[1] C.-W. Chang, M.-F. Hsiao, and M. Marek-Sadowska. “A new reasoning scheme for efficient redundancy addition and removal”, *TCAD* '03, pp. 945-951.

[2] P. Goel, An implicit enumeration algorithm to generate tests for combinational logic circuits, *IEEE Trans. Comp* ’81, pp. 215-222.

[3] C. Gomes and B. Selman, “Algorithm portfolios”, *AI’01*, pp. 43-62.

[4] J. P. Roth, W. G. Bouricius, and P. R. Schneider, Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits, *IEEE Trans. Electron. Comput.* ’67, pp. 567-579.

[5] M. Henftling, H. Wittmann, and K. Antreich, “A single-path-oriented fault effect propagation in digital circuits considering multiple-path sensitization”, *ICCAD* ’95, pp. 304-309.

[6] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification”, *TCAD* ’02, pp. 1377-1394.

[7] S. Kundu, L. Huisman, I. Nair, V. Iyengar, and L. N. Reddy, “A small test generator for large designs”, *Test Conference*, 1992.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: engineering an efficient SAT solver”, *DAC* ’01, pp. 530-535.

[9] S. Plaza, K.-H. Chang, I. Markov, and V. Bertacco, “Node mergers in the presence of don’t cares”, *ASP-DAC* ’06, pp. 414-419.

[10] J. Marques-Silva and K. Sakallah, “GRASP: A search algorithm for propositional satisfiability”, *IEEE Trans. Comp* ’99, pp. 506-521.

[11] C. Wang, I. Pomeranz, and S. Reddy, “REDI: an efficient fault oriented procedure to identify redundant faults in combinational logic circuits”, *ICCAD* ’01, pp. 370-374.

[12] C. Wang, S. Reddy, I. Pomeranz, X. Lin, and J. Rajski, “Conflict driven techniques for improving deterministic test pattern generation”, *ICCAD* ’02, pp. 87-93.

[13] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, “SAT sweeping with local observability don’t cares”, *DAC* ’06, pp. 229-234.

[14] M. Berkelaar and K. van Eijk, “Efficient and effective redundancy removal for million gate circuits”, *DATE* ’02, pp. 1088.

[15] Synopsys® Design Compiler®, http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx