Lazy Man's Resynthesis For Glitching-Aware Power Minimization

Andrea Costamagna[1,2], Xiaoqing Xu[1], Giovanni De Micheli[2], Dino Ruic[1]
[1] Google DeepMind, Mountain View, USA
[2] Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—This paper presents a novel resynthesis engine for minimizing the dynamic power of digital circuits. Traditional logic synthesis methods primarily focus on zero-delay toggles—logic state changes occurring between the start and end of a clock cycle. In contrast, our engine targets both zero-delay toggles and glitches, unintended transitions within a clock cycle caused by path imbalances. Glitches significantly contribute to power consumption in arithmetic circuits, making their minimization a critical challenge in electronic design. The proposed method uses a database of Pareto-optimal netlists to replace sub-networks in the target circuit with power-efficient alternatives. These replacements are guided by a simulation-driven cost function that evaluates workload-independent switching activity and penalizes gates with high fan-out. We call our approach *Lazy Man's Resynthesis* because it builds on an algorithm named *Lazy Man's Synthesis*, extending it from technology-independent delay optimization to post-mapping power optimization. Applied to the ISCAS and EPFL benchmarks, our method reduces glitching activity by 4.72% and dynamic power by 9.44%, achieving a 7.61% improvement over the state-of-the-art.**

***Index Terms*—Resynthesis, Dynamic Power, Glitching**

I. Introduction

MINIMIZING the power consumption of digital circuits is a fundamental concern in electronics. Poor design choices in the early stages of the design flow can propagate biases that degrade circuit's efficiency. Hence, addressing these issues early in the process, such as at the logic synthesis stage, is crucial for optimizing circuits' performance metrics.

A significant contributor to dynamic power consumption comes from *glitches*, unwanted transitions within a single clock cycle. In arithmetic circuits, glitches can account for up to 70% of the dynamic power consumption [1]. As systems increasingly rely on arithmetic-intensive operations, mitigating glitches is a critical challenge of modern design automation.

Logic synthesis typically consists of two stages: optimizing an abstract circuit representation and transforming it into a gate-level netlist through *technology mapping*. However, estimating glitching activity accurately from technology-independent representations is challenging. This limitation highlights the need for netlist-level restructuring after technology mapping, a process known as *resynthesis* [2].

This paper presents a versatile resynthesis engine for minimizing power in digital circuits. The proposed engine extends the *Lazy Man's Synthesis* (LMS) paradigm to the mapped design space, enabling efficient power optimization. Originally developed for delay-oriented, technology-independent optimization, LMS leverages low-cost substructures identified during graph traversal [3]. In our approach, LMS is adapted to address structural issues, such as path imbalances in reconvergent paths, which are a major source of glitching.

A key advantage of our engine is the possibility to use it for workload-independent power estimation and optimization, setting it apart from most prior works. Experimental results on the EPFL and ISCAS benchmarks demonstrate its effectiveness, achieving a 4.72% average reduction in glitching activity and 9.44% in dynamic power consumption.

II. Background

A. Logic Synthesis Basics

A *Boolean network* is a directed acyclic graph where nodes represent logic gates and edges represent wires. *And-inverter graph*s (AIGs) are technology-independent Boolean networks where each node is a two-input AND gate, and complemented edges identify inverters. A *mapped network* is a Boolean network where each node is a cell from a technology library. *Technology mapping* is the process of converting a technology-independent Boolean network into a mapped network [4], [5].

Investigating switching requires identifying the time intervals when toggles occur. The *arrival time* of a node v, t_v^A, is the earliest time at which its output is stable. The *sensing time*, t_v^S, is the earliest time at which a signal can change. The *activity interval* is the interval $[t_v^S, t_v^A]$ in which the node's output might toggle within a clock cycle.

Each node v in a combinational circuit implements a Boolean function, named *global node functionality*. A *dependency cut* is a subset $\mathcal{C} = (v, \mathcal{L})$, where $\mathcal{L}$ contains nodes that are not on any path from v to any circuit's output and such that there is a Boolean function f, named *cut functionality*, relating the global functionalities of v and $\mathcal{L}$ as $v = f(\mathcal{L})$.

A dependency cut $\mathcal{C} = (v, \mathcal{L})$ is a *structural cut* if the circuit contains a sub-network implementing the cut functionality. A *reconvergence-driven cut* is a structural cut built following heuristics that maximize the number of reconvergent paths from the *leaves* $\mathcal{L}$ to the root v. Reconvergence is a key concept for this paper because it generates *don't cares* [6], important degrees of freedom for Boolean optimization, and unbalanced reconvergent paths are a source of glitching [7].

B. Dynamic Power Optimization in Logic Synthesis

The power dissipated by a gate v_i depends on its switching activity $S(v_i)$, which is influenced by the input switching patterns and the circuit's structure. Let V_{dd} denote the supply

979-8-3315-2801-0/25/$31.00 ©2025 IEEE

voltage, and $C(v_i)$ the output load of node v_i. The average dynamic power dissipation can be expressed as:

$$\mathcal{P} = \frac{1}{2}\sum_{i=1}^{n} V_{dd}^2 S(v_i) C(v_i) \tag{1}$$

Accurately estimating $S(v_i)$ is crucial for low-power synthesis, but it is challenging because workloads are rarely known at design time. Additionally, the vast input space in industrial-scale designs makes exhaustive simulation impractical, and sampling input pattern pairs for statistically significant evaluation becomes computationally intractable.

Switching activity consists of two components: *zero-delay switches* $S_0(v_i)$, which are logic transitions within a clock cycle, and *glitches* $S_g(v_i)$, unintended transitions that happen before the gate stabilizes [7]. Several techniques have been proposed to optimize power before and during technology mapping. Most approaches primarily target zero-delay switching [8]. Although efforts to mitigate glitches during LUT mapping have been explored [9], extending these techniques to standard-cell-based designs remains largely unexplored.

Panda et al. [2] proposed addressing glitching after mapping, using an optimization algorithm based on *rewiring*, which replaces one or more gate inputs with other network signals while preserving the design's functionality. They introduced heuristics to reduce switching activity, accounting for glitching, and employed statistical methods to estimate power consumption when workloads are unknown.

C. A Cost Function Aligned with the Mapping Assumptions

Technology libraries offer multiple size instances for each cell type, with larger cells providing higher driving strength. To improve efficiency, modern mappers rely on characteristic cells with load-independent pin-to-pin delays, assuming that driving constraints can be met through *gate sizing* [4].

Gate sizing can also mitigate unwanted toggles by balancing signal delays and aligning information flow [10]. Another relevant approach, *gate freezing*, prevents unnecessary transitions using gates that can be frozen with a control signal [11]. However, these techniques cannot resolve structural sources of glitching, such as imbalances on reconvergent paths.

This work addresses this challenge by optimizing mapped netlists to minimize the following cost function:

$$\mathcal{F} = \sum_{i=1}^{n} S(v_i)|\mathrm{FO}(v_i)| \tag{2}$$

Minimizing this cost function is important for two key reasons. First, the output load of a gate is influenced by factors such as the input load of the nodes in its fan-out and wiring capacitances. While these quantities are unavailable during mapping, a first-order approximation assumes the output load is proportional to the fan-out size: $C(v_i) \propto |\mathrm{FO}(v_i)|$. This assumption combined with the load-independent delay model implies that $\mathcal{F} \propto \mathcal{P}$. Additionally, $\mathcal{F}$ favors netlists with smaller fan-out sizes, reducing the likelihood of selecting large cells during sizing and minimizing internal gate switching.

D. Boolean Network Optimization and Resynthesis

Logic synthesis algorithms fall into three categories:

1) *Cut-based rewriting*: Identifies structural cuts within a circuit and replaces the corresponding sub-networks with functionally equivalent, lower-cost alternatives [12].
2) *Window-based resubstitution*: Extracts large sub-networks, referred to as *windows*, performs exhaustive simulation to determine the local functionality of the nodes they contain, and replaces portions of the windows with more efficient sub-networks [13].
3) *Simulation-guided resubstitution*: A specialized form of window-based resubstitution in which the functional information of the window's nodes is represented by *simulation signatures*, Boolean vectors that approximate the global functionality of the nodes [14].

These techniques can be applied to both technology-independent representations (e.g., AIGs) and mapped networks. In the latter case, logic optimization is referred to as *resynthesis*. Resynthesis leverages technology-specific information while maintaining computational efficiency through simple delay models and lightweight data structures.

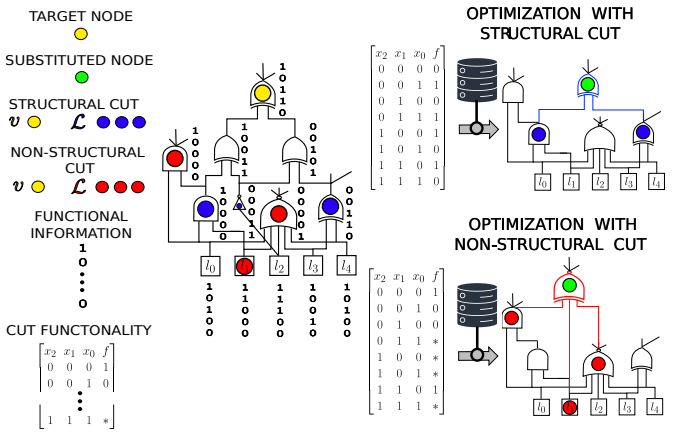

Fig. 1. Resynthesis techniques illustrated. The node v (●) serves as the root for a reconvergent-driven cut $\mathcal{C} = (v, \mathcal{L} = \{l_i\}_{i=0}^{4})$, defining a *window* of nodes. Each node in the window is simulated to extract functional information. Two optimizations follow: (top) a structural cut with blue leaves (●) or (bottom) a non-structural cut with red leaves (●). In both cases, the cut-functionality is extracted, an optimal sub-network is selected from a database, and the original root node (●) is replaced with a new node (●) preserving the functionality.

Figure 1 illustrates two possible transformations within a mapped network. The top example illustrates cut-based rewriting, while the bottom example shows simulation-guided resubstitution using small simulation signatures. If exhaustive simulation of the leaves were employed instead, the latter example would serve to exemplify window-based resubstitution.

A notable example of cut rewriting is *Lazy Man's Synthesis* (LMS) [3]. LMS minimizes the depth of AIGs by exploiting smaller substructures that are optimized for specific input arrival profiles. These substructures are extracted from larger AIGs and stored in a database for use as rewriting candidates.

An emerging trend in logic synthesis involves orchestrating multiple optimization strategies during graph traversal [15]. For instance, a recently proposed area-oriented resynthesis

engine integrates cut-based rewriting, window-based resubstitution, and simulation-guided resubstitution [16]. The engine leverages a database of precomputed netlists, derived by mapping minimum-size AIGs, to guide optimization. This work extends the resynthesis engine in [16] to target power optimization. Inspired by the LMS approach, databases are constructed through the traversal of large mapped networks.

III. Power-Oriented Lazy Man's Resynthesis

This section presents our novel resynthesis engine for power-oriented optimization of combinational circuits after technology mapping, assuming that no information on the workload is available. We propose an efficient functional simulator for accurate power estimation, a timing-aware Boolean function canonization technique for compact netlist storage, and dominance relations, inspired by Lazy Man's Synthesis, to construct a database with close-to-optimum sub-networks.

A. Word-Level Simulation for Power Evaluation

We propose a word-level simulator for detailed analysis of signal transitions, enabling efficient power evaluation. Each node v in the circuit stores a Boolean matrix $\boldsymbol{\varphi}^v \in \mathbb{B}^{P\times T}$, where P is the number of input pattern pairs and T is the number of quantization steps for the interval $[t_v^S, t_v^A]$. Column j of $\boldsymbol{\varphi}^v$ corresponds to time $t_v^j = t_v^S + (t_v^A - t_v^S)j/(T-1)$.

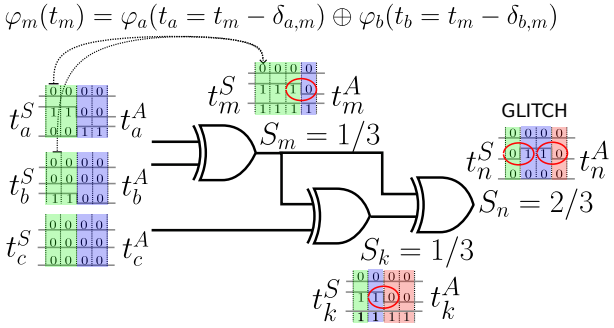

Fig. 2. Representation of the power simulator for a three-input pairs workload for (a, b, c): $(0,0,0) \to (0,0,0)$, $(1,0,0) \to (0,0,0)$, and $(0,1,0) \to (1,0,0)$. The activity interval of each node v ($[t_v^S, t_v^A]$) is quantized into $T = 4$ steps, identifying a Boolean matrix $\varphi_v \in \mathbb{B}^{3\times 4}$, where each column is associated with a quantization step of the *activity interval*. To obtain $\varphi_m(t_m)$, we must evaluate the EXOR function at its fanins, in the quantization step corresponding to $t_a = t_a^S + (t_m - \delta_{a,m} - t_a^S)/(T-1)$ and $t_b = t_b^S + (t_m - \delta_{b,m} - t_b^S)/(T-1)$, with $\delta_{v,m}$ the pin-to-pin delay from the technology library.

Each column j of $\boldsymbol{\varphi}^v$ is computed in parallel by evaluating the corresponding columns of the fan-in node matrices. For each fan-in and time-step, the relevant column in the fan-in matrix is determined by shifting the simulation time t_v^j by the pin-to-pin cell delay and extracting the appropriate column based on the fan-in's activity interval. Efficient simulation is achieved by representing each gate as a chain of binary Boolean operators, enabling word-level evaluation [17]. This approach allows the rows of $\boldsymbol{\varphi}^v$ to be populated in parallel. Algorithm 1 outlines the computations performed by our simulator to evaluate the cost function $\mathcal{F}$ for a netlist G, given a workload $\mathbf{W}$. Specifically, $S(v)$ denotes the total switching activity of node v, $S_0(v)$ represents its zero-delay switching,

Algorithm 1: Window-Based Power Estimation

Data: Mapped network G, workload $\mathbf{W} \in \mathbb{B}^{I\times P\times T}$
Result: A power estimation $\mathcal{F}$ for G.

1 $\{\boldsymbol{\varphi}^v\}_{v\in G} \leftarrow$ simulate G with $\mathbf{W}$;
2 **for** $v \in G$ **do**
3 $\quad S(v) = \frac{1}{P}\sum_{p=1}^{P}\sum_{t=1}^{T} |\varphi_{p,t}^v \oplus \varphi_{p,t-1}^v|$;
4 $\quad S_0(v) = \frac{1}{P}\sum_{p=1}^{P} |\varphi_{p,0}^v \oplus \varphi_{p,T-1}^v|$;
5 $\quad S_g(v) = S(v) - S^0(v)$;
6 **return** $\sum_{v\in G} S(v) \cdot |\mathrm{FO}(v)|$;

and $S_g(v)$ isolates its glitching activity. These computations are efficiently executed using bit-wise operations, leveraging word-level Boolean differences and pop-count techniques.

After simulating the network with the workload $\mathbf{W}$, we obtain a simulation matrix $\boldsymbol{\varphi}^v$ for each node v (line 1), from which switching activity and dynamic power evaluation follow directly with efficient bit-wise operations (lines 2-6). Figure 2 illustrates the application of the simulator to a small network, highlighting the simulator's ability to capture dynamic switching activity, including glitches.

B. Leveraging the Simulator for Power-Oriented Optimization

To ensure computational efficiency, we limit the number of pattern pairs considered to $P \sim 2^{12}$. Let I represent the number of primary inputs in the network. When $P \ll 2^{2I}$—a common scenario in industrial designs—the workload $\mathbf{W}$ significantly under-samples the vast space of possible input transitions. Optimizing netlists based on this limited simulation data can lead to suboptimal or counterproductive results, especially if the network is later evaluated with a different workload $\mathbf{W}$.

As described in Section II-D, our resynthesis approach combines three strategies, two of which involve constructing *windows* using the heuristic from Riener et al. [6]. This process involves identifying a reconvergence-driven cut $\mathcal{C} = (v, \mathcal{L})$, collecting the nodes between the root v and the leaves $\mathcal{L}$, and adding side nodes with fanins in the current set (see Fig.1). Windows constructed in this way are optimal regions for optimization and allow us to achieve workload independence.

The reconvergence-driven cut maximizes the presence of *don't-cares*, enabling effective Boolean optimization [6]. Additionally, these windows are particularly suitable for glitching-aware power minimization, as unbalanced reconvergent paths are a significant source of glitches. Furthermore, the number of leaves of a window is considerably smaller than the number of primary inputs. Hence, a local workload $\mathbf{W}$ can capture a substantial portion of potential events at the window's inputs. We assume that any pattern pair might occur at the window's leaves since *don't-cares* signals can appear as glitches.

We leverage this observation to reduce workload dependence during optimization by assigning $\mathbf{W}$ to the matrices $\boldsymbol{\varphi}^v$ of the window's leaves. Efficient simulation then computes the corresponding matrices for the other nodes within the window. This information serves two key purposes:

1) Evaluating the reduction in power consumption when specific nodes are removed.

2) Assessing the additional cost of incorporating a sub-network whose inputs are nodes within the window.

Efficient simulation of sub-networks with local workloads is a crucial step in our approach, as it allows us to compare the current design with potential alternative restructurings.

C. Timing-Aware Permutation-Canonization

We adopt a database-rewriting approach that relies on pre-synthesized sub-networks [16]. Inspired by modern technology-aware synthesis [18] and LMS [3], our method stores multiple sub-network implementations for each functionality. This accounts for the fact that a sub-network's optimality depends on the input arrival times.

From a memory perspective, storing all possible sub-network implementations for different arrival patterns is infeasible. To overcome this, we introduce a *timing-aware Permutation-canonization*, which maximizes the number of storable functions. This process ensures that only representative functions are stored, with other functions derived through input permutations, thus reducing storage requirements. The timing-awareness of this approach further enhances our engine's ability to optimize for balance during resynthesis.

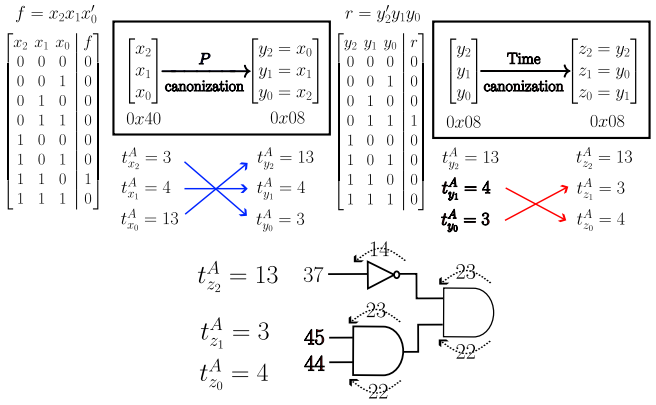

Fig. 3. Timing-aware P-canonization of a netlist for efficient database storage. The example assumes a cut $\mathcal{C} = (v, \mathcal{L})$ with functionality $f = x_2x_1x_0'$. To attempt resynthesis, we first apply P-canonization to find the input permutation yielding the lexicographically smallest truth table. Swapping x_2 and x_0 gives $0x08 < 0x40$. The resulting characteristic function $r = y_2'y_1y_0$ is symmetric in y_1 and y_0, so we reorder them to prioritize the earliest arrival. Netlists are stored in the database with symmetric pins ordered so that those with the longest path to the output pin appear first. Consequently, sorting input variables in the reverse order enhances balancing.

The left-hand side of Figure 3 illustrates the *Permutation-canonization* procedure. Given a Boolean function extracted from a network, we identify input permutations that minimize the function's truth table lexicographically. For instance, consider $f = x_2x_1x_0'$ (hexadecimal representation: 0x40). By permuting x_2 and x_0, we obtain $f = y_2'y_1y_0$ (0x08), which is lexicographically smaller. Input permutations are cost-free; only the smallest representation is stored.

Given a Boolean function f, if two variables x_i and x_j satisfy the condition $f(..., x_j, ..., x_i, ...) = f(..., x_i, ..., x_j, ...)$, e.g., x_0 and x_1 for $f = x_2(x_1 \oplus x_0)$, we exploit this symmetry to enhance balancing during database matching. We assign labels to each sub-network input based on the longest path to the output. Inputs within the same symmetry group are then sorted in decreasing order of path length. During matching, inputs with the latest (earliest) arrival times are paired with pins having the shortest (longest) path lengths.

This timing-aware matching strategy helps balance the paths to the target node. For example, Figure 3 shows that the variables in the symmetry group $\{y_0, y_1\}$ are sorted by their longest paths, assuming $t^A_{y_1} > t^A_{y_0}$. Balancing such paths reduces the potential for glitches in reconvergent sub-networks.

D. Netlist Dominance

A key challenge in optimizing database storage is determining which structures to retain, given limited space compared to all the possible netlists. Using the LMS terminology, the goal is to define *dominance* of a netlist over another one.

Let G_i and G_j be two netlists implementing the same Boolean function $f : \mathbb{B}^n \to \mathbb{B}$. Their sorted pin-to-pin delays are $\{d^i_k\}_{k=1}^n$ and $\{d^j_k\}_{k=1}^n$. We define the following properties:

- *Delay Dominance*: $G_i \overset{\text{D}}{\succ} G_j$ if $d^i_k \geq d^j_k$ for all k, with at least one strict inequality.
- *Area Dominance*: $G_i \overset{\text{A}}{\succ} G_j$ if the area of G_i is greater than the area of G_j.
- *Power Dominance*: $G_i \overset{\text{P}}{\succ} G_j$ if the zero-delay power consumption of G_i is greater than that of G_j.

If all three dominations apply, G_i is *dominated* by G_j ($G_i \succ G_j$). A netlist is *Pareto-optimal* if it is not dominated by any other functionally equivalent netlist.

E. Database Construction

A key distinction of our method from prior resynthesis techniques lies in its innovative application of the LMS approach. While traditional LMS focuses on delay-optimization for AIGs, we extend its scope to post-mapping resynthesis and consider metrics beyond delay.

Algorithm 2 outlines the database construction process. For each node, we enumerate a set of structural cuts (line 2). Each cut $\mathcal{C}_i$ defines a sub-network G_i (line 4) that implements a Boolean function f (line 5). If no sub-network implementing f exists in the database, we insert G_i (lines 6–7). Otherwise, we evaluate G_i against the stored sub-networks based on delay, power, and area, where power estimation is limited to zero-delay power for efficient comparison. If G_i dominates any stored sub-network, it replaces the weaker one (lines 10–11). If it outperforms all stored sub-networks in at least one metric and the database has capacity, we add it (lines 13–14). If no additional sub-network can be stored, we retain the most balanced one, as symmetry arguments suggest that it will maximize the success rate in resubstitution.

This algorithm runs in linear time with respect to the network size, ensuring high efficiency. Moreover, it can be executed offline across various benchmarks to populate the database. Subsequently, database lookups occur in constant time, making the rewriting process highly efficient, as demonstrated in the experimental section.

Algorithm 2: Database Construction

Data: A mapped circuit G, a database D
Result: The database D optimized.

```
1  for v ∈ G do
2      C = {(v, L_i)}_i ← enumerate structural cuts;
3      for C_i = (v, L_i) ∈ C do
4          G_i ← get sub-network for C_i;
5          f ← extract the cut functionality;
6          if D[f] = ∅ then
7              D[f] ← {G_i};
8          else
9              for G_j ∈ D[f_s] do
10                 if G_j ≻ G_i then
11                     replace G_j with G_i in D;
12                 else if G_i is Pareto-optimal then
13                     if |D[f]| <max capacity then
14                         D[f] = D[f] ∪ {G_i};
15                     else
16                         Store the most balanced one;
17 return D;
```

Algorithm 3: LMS Resynthesis

Data: mapped circuit G, database D
Result: A new mapped circuit optimized for power

```
1  W ∈ B^{P×T} ← sample a local workload;
2  G ← sort the network G;
3  for v ∈ G and it++<N do
4      Build and simulate a window for node v;
5      C_1, G_1 ← Find the best structural cut;
6      C_2, G_2 ← Find window-based dependency cut;
7      R_1, R_2 ← reward of substituting G_1 and G_2;
8      if ∃R* > 0 and constraints satisfied then
9          v_new ← Synthesize the netlist G* with R*;
10         Substitute v with v_new;
11     else if level(v) < L then
12         C_3, G_3 ← Find signature-based dependency cut;
13         R_3 ← reward of substituting G_3;
14         if R > 0 and constraints satisfied then
15             v_new ←Insert G_3;
16             if v_new and v are globally equivalent then
17                 Substitute v with v_new;
18 return the optimized circuit;
```

F. Glitching-Aware Resynthesis

Algorithm 3 presents our resynthesis engine, which extends the foundational work of [16], introduced in Sec. II-D, to incorporate power optimization. The process starts with an initial dynamic power evaluation to identify nodes with high switching activity (line2). For each node v, the switching activity $S(v)$ is estimated by simulating a large reconvergence-driven cut rooted at v using a local workload $\mathbf{W}$. Optimization is then applied to the top N nodes ranked by their cost $S(v)|\texttt{FO}(v)|$, where N is an user defined parameter.

Next, the algorithm orchestrates various optimization strategies to identify the optimal transformation for each window, if any. The process begins by evaluating potential improvements using the least expensive procedures: cut rewriting and window-based resubstitution (lines 5-10). If either method can improve the network, the best transformation is applied.

If no improvements are identified, the algorithm proceeds to simulation-guided resubstitution, provided the node's level in the network does not exceed a predefined limit L (lines 11-17). This restriction helps mitigate excessive runtime, as validating the correctness of a substitution via equivalence checking gets increasingly computationally expensive at higher node levels.

When substituting the root of the window v with another node, a sub-network G_r can be removed as it becomes redundant. The cost of G_r is calculated as:

$$\mathcal{F}_r = \sum_{v' \in G_r} \left(S(v')|\texttt{FO}(v')| + \sum_{v'' \in \texttt{FI}(v') \wedge v'' \notin G_r} S(v'') \right) \quad (3)$$

The second term accounts for the fact that removing a node in G_e also eliminates its connections with nodes in its fan-ins.

The cost of a candidate replacement G_c is computed by obtaining the matrices $\boldsymbol{\varphi}^{v_i}$ for its nodes, and computing

$$\mathcal{F}_c = \sum_{v' \in G_c} S(v')|\texttt{FO}(v')| + S(v_{c,out})|\texttt{FO}(v)| \quad (4)$$

The last term combines the switching activity of the output of the candidate netlist $v_{c,out}$ with the fan-out nodes it would inherit from the root of the window if the resubstitution is successful. The reward of this replacement is $\mathcal{R} = \mathcal{F}_r - \mathcal{F}_c$.

IV. Experiments

In this section, we evaluate Lazy Man's Resynthesis (LMR). The experiments were performed on an `i7-1365U` CPU machine and use the $7nm$ `asap7` technology library [19].

A. Glitch-Aware Resynthesis After Zero-Delay Synthesis

This experiment shows that the dynamic power in circuits optimized with traditional zero-delay synthesis techniques, can be reduced by incorporating glitching-aware optimization.

We evaluate the EPFL and ISCAS benchmarks by applying all available power-oriented optimization algorithms in ABC [8]. First, we apply a rewriting step to reduce area and eliminate functional redundancies by merging equivalent nodes. Next, we perform depth-reducing AIG balancing to minimize glitches at the AIG level, followed by high-effort power-oriented mapping. To further optimize the baseline for power, we convert the network to an LUT (`strash; dch; if -p;`), where efficient power-oriented logic restructuring is available in ABC. We then run `mfs -p;` twice. Finally, we apply power-oriented mapping using `map -p;`.

Table I shows that circuits optimized using traditional approaches exhibit glitches, which can be reduced through LMR.

The resulting network is evaluated using a random global workload, generated through random processes with seeds distinct from those used for the local workload. This ensures that the quality of the results is validated on input pattern pairs that differ from those of the local workload.Note that the average includes both the ISCAS and the EPFL benchmarks.

TABLE I
GLITCHING-AWARE RESYNTHESIS AFTER ZERO-DELAY SYNTHESIS.

Benchmark	Glitching		Switching		Time [s]
	ABC [8]	LMR	ABC [8]	LMR	
adder	100.81	93.53	399.72	366.41	0.21
bar	953.98	874.56	1725.85	1713.04	1.31
div	83.46	70.65	9560.70	9571.96	66.73
hyp	63926.70	63418.64	130022.05	129539.99	512.34
log2	8487.71	3276.94	14576.59	9297.93	52.60
max	134.71	113.06	987.68	961.89	0.62
multiplier	9621.40	4675.39	15442.18	10448.13	34.28
sin	2333.03	603.95	3503.66	1756.83	5.19
sqrt	416.09	151.58	5099.89	5270.87	27.73
square	7086.76	6790.39	10698.16	10559.58	15.57
arbiter	198.12	178.12	2136.80	1981.88	440.59
cavlc	67.14	65.98	182.13	186.84	0.30
ctrl	10.55	10.64	31.37	32.24	0.05
dec	8.48	14.44	23.13	40.74	0.54
i2c	79.89	75.07	272.20	264.69	1.10
int2float	20.15	19.98	63.48	66.37	0.05
mem_ctrl	2702.79	2164.49	8788.43	8373.39	698.63
priority	101.23	88.87	307.48	291.42	0.21
voter	8877.67	7127.10	11516.02	9656.27	14.97
		−14.81%		**−6.16%**	62.53s

B. Comparison With State-of-the-Art

The algorithm most similar to ours is that of Panda et al. [2]. We adapt their method within our framework to ensure a fair comparison. Specifically, we implement their procedure as a window-based resynthesis. For each node v, we extract the node's function f and identify the permissible functions of the fan-in node q. We then use this set to rewire the fan-in to a node using the cost functions in Eq. 3 and Eq. 4.

The results, shown in Table II, compare both engines applied to netlists mapped after area-oriented optimization, AIG balancing, and aggressive power-oriented technology mapping, followed by resynthesis. The table is divided into two sections: arithmetic benchmarks and random logic. In both cases, the algorithm reduces dynamic power consumption by over 8%, though the source of optimization differs across benchmarks.

For the arithmetic benchmarks, glitch minimization plays a significant role in power reduction, which aligns with the motivation of this work. Interestingly, the engine also optimizes non-arithmetic designs. This is due to the cost function's high expressiveness, which simultaneously accounts for structural sources of glitching, functionally-dependent zero-delay switching, and the effect of large fanouts. As a result, the engine is flexible enough to address the dominant power-consumption contributors based on the circuit's properties.

During optimization we impose strict delay constraints to avoid performance degradation due to the increase in delay. Under this constraint, the average delay variation in all benchmarks is -2.74%, with an area reduction of -0.68%. The improvement over the state-of-the-art is expected, as input rewiring is a specific case of dependency-cut selection.

C. Correlation With OpenROAD

Our experiments demonstrate that local optimization of the cost function in Eq. 2 effectively reduces power. However, its impact on later design stages requires further verification. To examine this correlation, Table III presents results obtained using `OpenROAD` [20] on a subset of the EPFL and ISCAS benchmarks, selecting the smallest ones to enable fast design closure. Specifically, we analyze internal and switching power, showing that minimizing Eq. 2 reduces switching activity and encourages the use of smaller gates after sizing, thereby lowering internal power. Although these results are based on a default flow without circuit-specific tuning, both metrics improve in almost all benchmarks, validating our assumptions and demonstrating the effectiveness of our method.

V. CONCLUSIONS

This paper presents a resynthesis engine for optimizing dynamic power after technology mapping. By extending the Lazy Man's Synthesis paradigm to post-mapping power optimization, we demonstrate its effectiveness in reducing power consumption. Experimental results show an average reduction of 4.72% in glitching and 9.44% in dynamic power, with these improvements carrying over to later stages of the design flow.

ACKNOWLEDGEMENTS

We thank Raj B. Apte for making this publication possible, Alan Mishchenko and Chang Meng for the fruitful discussions and Robert O'Callahan for the invaluable technical support.

REFERENCES

[1] Shen, Ghosh, Devadas, and Keutzer, "On average power dissipation and random pattern testability of cmos combinational logic networks," in *1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 1992, pp. 402–407.

[2] R. Panda and F. N. Najm, "Post-mapping transformations for low-power synthesis," *VLSI Design*, vol. 7, no. 3, pp. 289–301, 1998.

[3] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 597–604.

[4] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[5] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 410–416.

[6] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, "Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 395–402.

[7] N. Miskov-Zivanov and D. Marculescu, "Modeling and analysis of ser in combinational circuits," in *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2010.

[8] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[9] L. Cheng, D. Chen, and M. D. Wong, "Glitchmap: An fpga technology mapper for low power considering glitches," in *Proceedings of the 44th annual design automation conference*, 2007, pp. 318–323.

[10] M. Hashimoto, H. Onodera, and K. Tamaru, "A power optimization method considering glitch reduction by gate sizing," in *Proceedings of the 1998 international symposium on Low power electronics and design*, 1998, pp. 221–226.

TABLE II
COMPARISON WITH STATE-OF-THE-ART GLITCHING-AWARE OPTIMIZATION [2]

	Benchmark	Area	Delay	Glitching activity			Power			Time	
				Baseline	Rewire [2]	LMR	Baseline	Rewire [2]	LMR	Rewire [2]	LMR
ARITHMETIC	c6288	144.17	1548.57	3380.15	3379.98	2020.08	6953.69	6951.21	3958.30	1.06	3.12
	c7552	92.75	514.81	495.22	492.97	459.88	1457.58	1428.74	1279.89	0.50	1.88
	adder	57.40	3548.84	77.63	77.63	77.63	336.32	336.32	336.32	0.20	0.46
	bar	126.25	199.39	774.17	774.17	774.17	3036.63	3036.63	3036.63	0.59	1.53
	div	2400.65	50292.99	78.17	76.79	82.83	21469.69	21465.12	19838.23	13.45	66.94
	hyp	14047.20	271933.19	75133.67	75136.57	75297.84	250064.86	250056.84	243140.20	39.53	326.52
	log2	1590.39	5072.06	16584.74	16632.04	15003.88	53456.91	53316.73	45348.70	19.23	62.34
	max	147.42	2930.65	76.12	76.12	76.12	1534.51	1534.51	1525.28	0.67	1.53
	multiplier	1332.67	3262.85	17135.79	17135.69	15109.43	45140.03	45136.96	36523.45	9.34	34.23
	sin	292.71	2366.35	2703.66	2597.31	2607.13	9630.71	9325.95	7989.34	4.64	10.20
	sqrt	1412.52	86160.67	407.74	394.81	377.64	16463.02	16352.69	16035.93	9.41	19.09
	square	1103.93	3337.42	7615.44	7618.67	7958.61	22412.85	22404.89	19314.82	7.25	33.67
Averages for the **arithmetic** benchmarks					−0.75%	**−5.75%**		−0.51%	**−11.16%**	8.82s	46.79s
CONTROL / RANDOM LOGIC	arbiter	592.62	1004.36	253.70	370.94	323.38	7218.25	7020.82	6834.13	2.95	5.45
	cavlc	38.98	221.78	83.49	83.49	84.42	407.01	407.01	376.85	0.92	1.62
	ctrl	8.60	131.19	16.55	16.23	15.39	80.00	79.41	65.76	0.20	0.30
	dec	31.20	86.33	8.61	9.21	17.74	258.25	265.72	167.66	2.21	2.75
	i2c	74.26	227.55	99.06	97.74	94.35	598.80	588.21	544.40	0.58	3.14
	int2float	11.75	216.64	20.25	20.25	21.35	109.07	109.07	100.21	0.11	0.22
	mem_ctrl	2721.72	1503.31	3742.30	3352.71	2857.73	27993.97	27045.18	25090.17	33.70	1722.47
	priority	57.23	2891.73	254.82	188.45	156.27	879.37	748.07	612.76	0.43	1.08
	router	15.42	468.09	14.80	13.60	13.06	89.07	77.15	89.57	0.13	0.23
	voter	674.98	1034.38	8916.78	8317.08	7893.67	21361.01	19491.03	17020.20	7.22	24.39
	c432	10.96	433.75	23.81	23.23	22.50	122.09	118.46	105.36	0.05	0.14
	c880	18.15	371.76	48.31	48.31	44.07	163.04	163.04	137.43	0.11	0.29
	c499	23.03	363.30	93.82	93.82	96.82	282.89	282.89	322.24	0.11	0.65
	c1355	23.02	326.78	81.25	81.25	82.45	257.94	257.94	259.68	0.11	0.74
	c1908	18.75	417.03	76.64	76.74	67.84	208.44	207.90	177.35	0.09	0.39
	c2670	31.01	322.61	122.20	122.03	123.44	434.16	432.87	408.35	0.14	0.62
	c3540	48.75	573.00	223.40	223.30	200.74	799.42	772.24	701.48	0.42	1.07
	c5315	78.35	589.41	430.64	423.90	347.08	1353.42	1314.98	1126.22	0.47	1.98
Averages for the **control/random logic** benchmarks					−1.33%	**−4.08%**		−2.66%	**−8.35%**	2.10s	212.73s
Averages for all the **EPFL** and **ISCAS** benchmarks					−1.11%	**−4.72%**		−1.83%	**−9.44%**	4.70s	148.49s

TABLE III
CORRELATION WITH OPENROAD [20]

Benchmark	Internal Power [W]		Switching Power [W]	
	Baseline	LMR	Baseline	LMR
c432	$5.77 \cdot 10^{-13}$	$5.70 \cdot 10^{-13}$	$7.17 \cdot 10^{-13}$	$6.87 \cdot 10^{-13}$
c880	$1.21 \cdot 10^{-12}$	$1.22 \cdot 10^{-12}$	$1.21 \cdot 10^{-12}$	$1.19 \cdot 10^{-12}$
c1908	$1.91 \cdot 10^{-12}$	$1.85 \cdot 10^{-12}$	$1.76 \cdot 10^{-12}$	$1.60 \cdot 10^{-12}$
c2670	$2.62 \cdot 10^{-12}$	$2.78 \cdot 10^{-12}$	$2.76 \cdot 10^{-12}$	$3.05 \cdot 10^{-12}$
c3540	$5.17 \cdot 10^{-12}$	$5.94 \cdot 10^{-12}$	$5.18 \cdot 10^{-12}$	$5.67 \cdot 10^{-12}$
c5315	$6.20 \cdot 10^{-12}$	$5.70 \cdot 10^{-12}$	$8.26. \cdot 10^{-12}$	$7.51 \cdot 10^{-12}$
c7552	$8.89 \cdot 10^{-12}$	$4.57 \cdot 10^{-12}$	$9.96 \cdot 10^{-12}$	$5.07 \cdot 10^{-12}$
c6288	$2.06 \cdot 10^{-9}$	$0.56 \cdot 10^{-9}$	$2.18 \cdot 10^{-9}$	$0.59 \cdot 10^{-9}$
max	$1.60 \cdot 10^{-11}$	$1.58 \cdot 10^{-11}$	$1.83 \cdot 10^{-11}$	$1.87 \cdot 10^{-11}$
multiplier	$1.15 \cdot 10^{-6}$	$0.93 \cdot 10^{-6}$	$1.63 \cdot 10^{-6}$	$1.18 \cdot 10^{-6}$
sin	$2.55 \cdot 10^{-8}$	$1.94 \cdot 10^{-8}$	$2.32 \cdot 10^{-8}$	$2.69 \cdot 10^{-8}$
square	$7.42 \cdot 10^{-10}$	$7.35 \cdot 10^{-10}$	$1.18 \cdot 10^{-9}$	$1.01 \cdot 10^{-9}$
arbiter	$2.55 \cdot 10^{-11}$	$2.49 \cdot 10^{-11}$	$3.11 \cdot 10^{-11}$	$3.00 \cdot 10^{-11}$
cavlc	$1.75 \cdot 10^{-12}$	$1.78 \cdot 10^{-12}$	$1.83 \cdot 10^{-12}$	$1.80 \cdot 10^{-12}$
ctrl	$4.38 \cdot 10^{-13}$	$4.00 \cdot 10^{-13}$	$4.36 \cdot 10^{-13}$	$3.95 \cdot 10^{-13}$
dec	$3.95 \cdot 10^{-13}$	$3.71 \cdot 10^{-13}$	$1.05 \cdot 10^{-12}$	$0.77 \cdot 10^{-12}$
i2c	$3.47 \cdot 10^{-12}$	$3.25 \cdot 10^{-12}$	$4.85 \cdot 10^{-12}$	$4.42 \cdot 10^{-12}$
int2float	$5.12 \cdot 10^{-13}$	$5.49 \cdot 10^{-13}$	$5.29 \cdot 10^{-13}$	$5.41 \cdot 10^{-13}$
priority	$3.24 \cdot 10^{-12}$	$2.89 \cdot 10^{-12}$	$3.65 \cdot 10^{-12}$	$3.23 \cdot 10^{-12}$
voter	$4.41 \cdot 10^{-9}$	$4.30 \cdot 10^{-9}$	$4.60 \cdot 10^{-9}$	$4.43 \cdot 10^{-9}$
Average improvement		**−9.61%**		**−11.02%**

[11] L. Benini, G. De Micheli, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Glitch power minimization by selective gate freezing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 287–298, 2000.

[12] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006.

[13] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–23, 2011.

[14] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De 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, 2021.

[15] Y. Li, M. Liu, M. Ren, A. Mishchenko, and C. Yu, "Dag-aware synthesis orchestration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[16] A. Costamagna, A. T. Calvino, A. Mishchenko, and G. De Micheli, "Area-oriented optimization after standard-cell mapping," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 1112–1119.

[17] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[18] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 352–359.

[19] V. Vashishtha, M. Vangala, and L. T. Clark, "Asap7 predictive design kit development and cell design technology co-optimization," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 992–998.

[20] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça *et al.*, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," *Proc. GOMACTECH*, pp. 1105–1110, 2019.