FastReplace: Efficient V_t Replacement Technique for Leakage Power Minimization

***Abstract*—This paper considers the timing-constrained discrete V_t replacement problem (DVRP), for leakage minimization in digital circuits. The problem is NP-complete. Earlier techniques reported for the DVRP employed iterative greedy or sensitivity-driven heuristics, that required incremental timing analysis after every iteration. The key observation reported in this paper is a good correlation between the slack distribution among gates in a given iteration and the order of gate replacements in subsequent iterations. This paper exploits the above observation to propose *FastReplace*, an iterative algorithm that uses *adaptive lazy timing analysis* to solve the DVRP. The proposed *FastReplace* technique, when applied to ISCAS and ITC benchmark circuits, produced solutions $9.8\times$ and $3.1\times$ faster as compared to the greedy technique and a commercial multi-V_t synthesis tool respectively, without impacting the solution quality.**

Keywords: V_t replacement, leakage minimization, incremental timing analysis, lazy timing analysis

I. INTRODUCTION

According to Dennard's law [1], power density should remain constant in spite of increase in MOS-device density with technology scaling. Conventionally, supply voltage scaling was used to reduce dynamic power consumption, and threshold voltage scaling to maintain/improve the critical path timing. In sub-$100nm$ regime, the exponential increase in subthreshold leakage with threshold voltage scaling caused leakage power to dominate total power consumption in microprocessors [2]. Since leakage power is dissipated in idle mode, it does not contribute to any useful computation. On the other hand, excessive leakage power dissipation can cause wastage of power resources and/or thermal runaway. There has been extensive research during the last decade to reduce leakage power at different levels of the VLSI design flow.

Gate-sizing and V_t-sizing are very efficient design-time techniques to reduce leakage power under timing constraints. Gate sizing is very effective in the early and middle stages of the physical synthesis flow. But in the post-route stage, applying gate sizing often necessitates incremental placement which may increase the turn-around-time [3]. On the other hand, V_t sizing will not impact the placement, while still providing room for significant optimization in power/timing. Thus, V_t sizing is employed to minimize leakage power at all stages of the physical synthesis flow. Any standard cell library shall have different versions of the same cell, one for each threshold voltage. However, the number of versions for a given cell is finite and limited to the discrete values of the threshold voltage as made available by the foundry. While solving the V_t sizing problem, the optimal value of V_t computed for each gate in the given circuit may not belong to the threshold voltages available in the given standard cell library, This necessitates mapping of the computed V_t sizes in continuous domain to the available (discrete) V_t values. Thus, the name *Discrete V_t Replacement Problem (DVRP)*. In this paper, we focus on the timing-constrained DVRP for leakage power minimization in digital circuits.

II. THE DISCRETE V_t REPLACEMENT PROBLEM

Consider a circuit formed using the gates $\{g_1, g_2, \ldots, g_{N_{gate}}\}$. Let $\{x_{g_i,j}\}$ denote a discrete variable defined as follows:

$$x_{g_i,j} = \begin{cases} 1, \ \textbf{if } gate\ g_i\ in\ the\ given\ circuit\ is\ realized \\ \quad with\ j^{th}\ choice\ of\ V_t \\ 0, \ \textbf{otherwise} \end{cases} \tag{1}$$

Let $D(g_i)$ denote the *arrival time*[1] at the output of gate g_i, and, $d_{g_{i,j}}$ and $p_{g_{i,j}}$ denote the delay and leakage-power respectively of gate g_i when realized with j^{th} choice of V_t. Let m denote the total number of V_t choices made available in the standard cell library. The optimization problem is to find $\{x_{g_i,j}\}$ for lowest leakage power, without violating the critical path timing. Let $fanin(g_i)$ be the set of all gates driving the gate g_i; $PO(C)$ and $PI(C)$ denote the set of primary outputs and primary inputs respectively of the given circuit C; and, T denote the timing budget assigned to the given circuit C. It can be formally stated as follows:

$$\text{Minimize} \sum_{i=1}^{N_{gate}} \sum_{j=1}^{m} x_{g_{i,j}} p_{g_{i,j}} \tag{2}$$

such that

$$\sum_{j=1}^{m} x_{g_{i,j}} = 1, \forall i, 1 \leq i \leq N_{gate} \tag{3}$$

$$x_{g_{i,j}} \in \{0,1\}, \forall i, 1 \leq i \leq N_{gate}; \forall j, 1 \leq j \leq m \tag{4}$$

$$D(g_i) + \sum_{j=1}^{m} d_{g_{i,j}} x_{g_{i,j}} \leq D(g_k), \forall g_i \in fanin(g_k) \tag{5}$$

$$D(O) \leq T, \forall O \in PO(C) \tag{6}$$

$$D(I) \geq 0, \forall I \in PI(C) \tag{7}$$

[1]maximum propagation time of any event in the primary inputs of the given circuit to a given wire

In the above equations 6 and 7, $D(O)$ and $D(I)$ denote the arrival-times at primary output wires and primary input wires respectively. Equation 2 presents the objective function that minimizes the leakage power of the given circuit. Equations 3 and 4 ensure that exactly one version of gate g_i is used among the available V_t choices. Equations 5, 6 and 7 ensure that the arrival-time constraints are met for all the gates, primary inputs and primary outputs of the circuit.

III. RELATED WORK

As mentioned earlier, a solution to the DVRP involves repeated timing analysis of the given circuit. Thus, fast solution to the DVRP requires a fast timing analyzer. Early Static Timing Analysis (STA) engines always processed an entire design, which is impractically expensive for evaluating every V_t replacement [4]. It is to be noted that replacing V_t of a gate g_i, changes the arrival times of gates only in the fan-in and fan-out cones of g_i, while the arrival times of other gates remain unaffected. By performing STA for entire design, we may end up computing known values repeatedly. To improve the efficiency of the STA engine, several *incremental* STA techniques have been proposed in the past [5]–[10].

In [5], *incremental* STA is performed by solving the *incremental longest path problem*, with a novel algorithm which is linear in the number of edges in the *dominance fan-out cone*. In [6], leftmost and rightmost frontiers of change in relative timing values are recorded. Based on a request, incremental timing analysis is performed on the modified design employing the recorded frontiers of change to limit the timing analysis to the *affected* regions of the circuit alone. An input based path sensitization approach was used for incremental STA in [7]. In [8], timing queries were modeled using temporal logic and an efficient algorithm was proposed to answer those queries. In [9], an efficient incremental STA algorithm which exploits the circuit structure was proposed. In all these works, path based algorithms were proposed to increase the performance of incremental STA, but none of them except [6] looked at reducing the number of times the incremental STA needs to be performed. In addition, none of the above mentioned techniques addressed *timing-constrained leakage power optimization*. The *timing-constrained leakage power optimization* problem was addressed in [3], [12], [13]. In [3], a polynomial time approximation scheme was proposed, but this scheme does not scale for large circuits, since the running time grows quadratically with the size of the circuit. In [12] and [13], iterative greedy heuristics were shown to be fastest and most effective for solving the *timing-constrained leakage power optimization* problem. However, it is stated in [12] that even their proposed technique may take several hours when applied to large circuits ($> 100K$ gates). In this paper, we show by experimentation that the greedy techniques indeed take several hours, when applied to large circuits.

This paper uses the *lazy evaluation* paradigm to arrive at a fast algorithm for the DVRP. *Lazy evaluation* is a popular paradigm in improving the computational efficiency of iterative algorithms. The line sweep algorithm used in the VLSI routing checkers is based on this technique [11]. Incremental STA for *timing optimization* based on lazy updates was first proposed in [6]. Later, in [4], it was shown that lazy STA when employed for *timing optimization* can result in more than $2\times$ speed-up, when combined with transactional timing analysis. *It should be noted that lazy updates may not be acceptable for all optimization problems*. An *optimistic* approach with *lazy timing updates*, may lead to unnoticed timing violations that might have occurred on the gates that have been marked *dirty*. This demands backtracking that actually ends up increasing the running time of the algorithm, defeating the whole purpose. Interestingly, *lazy updates* do speed up the iterative algorithms that solve the *timing optimization* problem.

This paper proposes the *FastReplace* algorithm for the DVRP that uses the *lazy update* paradigm to reduce significantly the number of STA runs during the optimization, thereby speeding up the time required for solving the problem without compromising on the quality of the solution.

IV. THE FASTREPLACE ALGORITHM

A. Motivation

The greedy techniques reported earlier in the literature are iterative. In each iteration a gate with positive slack is assigned a higher V_t version so as to reduce the leakage power, provided the replacement does not violate the timing budget. At the end of each iteration, incremental STA is performed and arrival times/slacks are updated for all gates[2]. It is stated in [13] that the greedy heuristic which replaces the gate with the largest slack at every iteration, is the fastest. Hence, all comparisons done in this paper are with respect to the greedy heuristic mentioned in [13]. Let $\pi_i = \{\pi_i(1), \pi_i(2), \dots\}$ be the list of gates in decreasing order of slack, in the i^{th} iteration. For *c432* circuit, the first 5 iterations are summarized below. The integers in the lists below denote the gates (Gate ID) and not the slack time. The gates *underlined* are the ones which get replaced in the respective iterations.

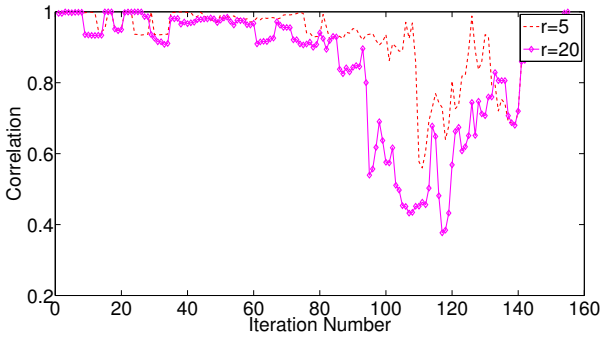
π_1: <u>51</u> 52 59 70 67 72 71 58 74 65 56...
π_2: <u>51</u> 52 59 70 67 72 71 58 74 65 56...
π_3: <u>59</u> 70 67 72 71 58 74 65 56 52 ...
π_4: <u>70</u> 67 59 72 71 58 74 65 56 52 ...
π_5: <u>67</u> 59 72 71 58 70 74 65 56 52 ...

These lists provide the following insight into the working of the greedy heuristic:

Observation 1: *Gates higher in the ordering at the beginning of an iteration, get replaced successively* with a high probability *in the next few iterations. For e.g. the gates* $51, 59, 70$ *and* 67 *that were in the top* 5 *before iteration* 1*, got replaced in successive iterations in the same order*[3]*. Therefore, instead of replacing only one gate (the topmost in the list) during an*

[2]Usually arrival times are associated with a net. In this paper, whenever we mention about the arrival time of a gate, we intend to indicate about the arrival time at the output of the gate

[3]It can be noted that Gate 51 is at the top of the list for both π_1 and π_2, as it had large amount of slack even after replacement in iteration 1 and that the standard cell library used has more than two V_t versions.



(a) c432

Fig. 1. X axis: i, Y axis: $\rho(i, i+r)$, where
i: iteration number and r : number of gate replacements per each iteration

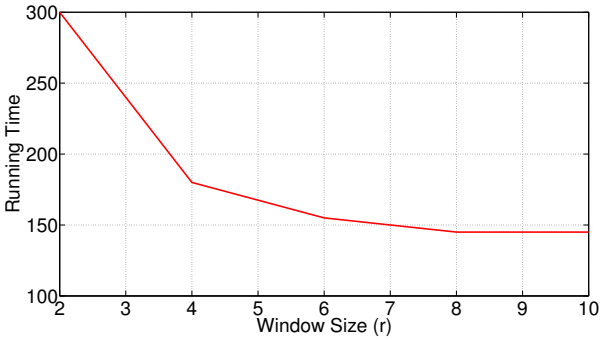


Fig. 2. Saturation of Running time with Window Sizing for *b14*

iteration, we can replace many gates that are among the top of the list .

Observation 1 implies that the STA can be performed after *multiple* gate replacements (*Lazy timing evaluation*) in contrast to performing the same after every single gate replacement. This in turn, significantly reduces the number of STA runs and thereby, the running time of the entire algorithm. Since incremental STA takes time, which is linear in circuit size, in the worst case, it is understood that the computational efficiency of the gate replacement algorithm can be improved by reducing the number of the incremental STA runs.

To empirically justify our claim the following experiment was conducted. The greedy heuristics in [13] was employed on the *c432* circuit for many iterations. For each iteration i, sequence of gates denoted by $\pi_i^{'}$ was computed such that $\pi_i^{'} = (gr_1, gr_2, gr_{i-1}, g_i, g_{i+1}....)$, where gr_j, $1 \leq j < i$, denote the gate replaced in iteration j, and g_k, $k \geq i$ are the gates in decreasing slack values at iteration i. Then, $\pi^{'} = \{\pi_1^{'}; \pi_2^{'}; \dots\}$ is a matrix. It is straightforward to see that a good correlation between the rows of $\pi^{'}$ justifies observation 1.

The *autocor* command in *GNU Octave* software is used to find the correlations between the rows in $\pi_i^{'}$. The autocorrelation function used in this experiment is shown in in Equation 8, where i stands for the iteration number and r for number of gate replacements.

$$\rho(i, i+r) = corr(\pi_i, \pi_{i+r}) \tag{8}$$

Figure 1 shows the autocorrelation plot of rows of $\pi^{'}$. The Y-axis shows the correlation and X-axis the iteration number. Each waveform on this plot shows how the correlation changes in successive iterations. Figure 1 shows that even for $r = 20$, the correlation is very good, providing the opportunity for performing *lazy timing updates* between gate replacement iterations. If correlation is good for a certain value of r, it indicates that we can replace r gates at a time, without violating timing and without majorly disturbing the solution quality. We call this variable r as *gate replacement window*. Additionally Figure 1 shows how the correlation varies with *iteration number*. Thus, using a fixed *gate replacement window* in all iterations, may lead to suboptimal results. It would be desirable to change *gate replacement window* in successive iterations, so as to adapt the algorithm to the iterative process, effectively pruning the solution quality and the algorithm running time.

B. Gate Replacement Windows

From the last section, we infer that in a given iteration i, if $\rho(i, i+r)$ is high for a certain r, r gates can be replaced in iteration i itself, with little loss in solution quality compared to greedy iterative gate replacement. There are two major challenges in implementing this concept.

1) The $\rho(i, i+r)$ function is not available beforehand for all values of i. Computing the same takes prohibitively large time and also results in solving the DVRP itself.
2) The *window size* r cannot be a constant, as we see from Figure 1 that for a given r the correlation decreases for higher iteration numbers. The decrease in correlation not only implies suboptimal results but also that the replacement done may cause *timing violations* in subsequent STA run forcing a backtrack (undo the replacement). Large number of such backtracks shall increase the execution time of the algorithm. Thus, the window size r needs to be *adaptive* and change across iterations.

Figure 2 shows the plot of running time versus varying r. It can be seen here that the running time decreases with an increase in r. After certain value of r (r_{opt}), it saturates and does not improve further. This is because

- with increase in r, there is a decreasing in the running time of the algorithm.
- a very high value of r causes too many undo operations, thereby saturating the running time of the algorithm.

The value of r_{opt} was 8 for *b14* circuit. This behavior was observed for other circuits also, although the value of r_{opt} was different in each case. In general, the value of r_{opt} increases with the size of the circuit as shown in Figure 3.

The *FastReplace* algorithm addresses the challenges of adaptively sizing r during optimization as follows: The value of r is varied within a range $[W_{low}, W_{high}]$. At the start, r is

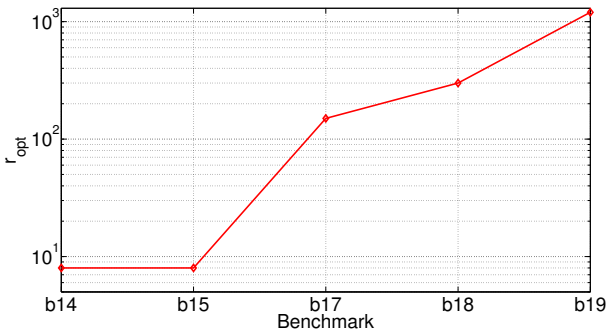


Fig. 3. r_{opt} for different benchmarks

initialized to W_{low} and incremented in each iteration until it is equal to W_{high}. After r reaches W_{high}, it is maintained at that value and is not increased further. At any stage, if there is a timing violation, the previous window of replacements is undone, and the value of r is reset to 1, so as to gracefully recover from the impact of the violation.

C. The Proposed Algorithm

Algorithm 1 shows the proposed algorithm, wherein, the $window\_size$ is initialized to W_{low} (line 1) to start with. All gates are assigned to their LV_t (lower V_t) versions that are fast but consume large leakage power. An initial STA run is done in line 3 to compute the slacks. Each execution of the **while** loop in line 5 corresponds to an iteration. During each iteration, the top $window\_size$ number of gates in decreasing order of their slacks are replaced as follows: if a gate is a LV_t cell then it is replaced by its SV_t (Standard V_t) version, and if it is a SV_t cell then it is replaced by its HV_t (High V_t) version (lines 5 - 15). The STA run is performed in line 16. Based on the results, the new $window\_size$ value is computed as described in previous section (lines 17 - 34).

V. Experimental Setup and Results

A. Experimental Setup

The objective is to minimize leakage power of a given digital circuit without degrading the performance. To achieve this, the netlist is synthesized using 90 nm LV_t library which is a high performance library. We also set the timing constraint option $set\ max\_delay$ to 0.0 in the synthesis script, which will yield a highly timing optimized LV_t netlist. We perform mixed V_t synthesis for leakage minimization (using a $90nm$ mixed V_t library from the same vendor), by setting the leakage constraint option $set\ max\_leakage$ to 0.0 in the synthesis script. Additionally we set the timing constraint option $set\ max\_delay$ to the circuit delay obtained after the LV_t synthesis. The resultant netlist is thus timing constrained and power optimized.

The *FastReplace* tool (written in $C++$) takes the synthesized LV_t netlist as input and represents the netlist as a graph, using the *Boost Graph Library 1.53.0*. Each node in the graph will have the following properties

- *type* (PI, PO, cell type),
- *fanins* (array of all *gate-id*s that are fanins of that node),
- *fanouts* (array of all *gate-id*s that are fanouts of that node),
- *delay* (average delay),
- *leakage* (average leakage),
- *nature* (LV_t, HV_t, SV_t), *output arrival time* and *slack*.

The values of delay and leakage power are assigned to each node, based on the *nature* parameter. For timing evaluation, we use a full blown STA engine, because an incremental STA engine is not feasible for multiple replacements with *lazy*

Algorithm 1: V_t Replacement based on Adaptive Window Sizing

Input: Nestlist C represented as a DAG, lower bound on window size (W_{low}), upper bound on window size (W_{high})
Input: Set of V_ts and corresponding delay, power values
1 $window\_size = W_{low}$;
2 $n \leftarrow gate\ count$;
3 Run STA after assigning LV_t delays to all gates, and compute slack for each gate;
4 A $\leftarrow$ list of gates, sorted in descending order of slacks;
5 **while** $A.size < window\_size$ **do**
6 **for** i *in* $1 \rightarrow window\_size$ **do**
7 Δ_1 = $A_i.SV_t$ delay - $A_i.LV_t$ delay;
8 Δ_2 = $A_i.HV_t$ delay - $A_i.SV_t$ delay;
9 **if** $(A_i\ is\ LV_t)\ and\ (A_i.slack > \Delta_1)$ **then**
10 Replace A_i with its SV_t equivalent;
11 **end**
12 **else if** $(A_i\ is\ SV_t)\ and\ (A_i.slack > \Delta_2)$ **then**
13 Replace A_i with its HV_t equivalent;
14 **end**
15 **end**
16 Check for delay violation by updating arrival times;
17 **if** *delay violation* **then**
18 **if** $window\_size > 1$ **then**
19 undo replacements;
20 $window\_size \leftarrow 1$;
21 **end**
22 **if** $window\_size = 1$ **then**
23 undo replacement;
24 mark A_i as critical;
25 discard A_i;
26 **end**
27 **end**
28 **else**
29 update slacks by calculating required time ;
30 update A;
31 **if** $window\_size < W_{high}$ **then**
32 $window\_size \leftarrow window\_size + 1$;
33 **end**
34 **end**
35 **end**
Result: V_t assignment to each gate

TABLE I
LEAKAGE POWER AND RUNNING TIME COMPARISONS

	LV_t Synthesis	Mixed V_t Synthesis using Commercial Tool		Mixed V_t Synthesis using Greedy [13]		Mixed V_t Synthesis using FastReplace with r=5		
Circuit	Leakage	Leakage	Runtime	Leakage	Runtime	Leakage	Runtime	Speed-Up over Greedy [13] (S_1)
c3540	82.11μW	18.60μW	62s	24.00μW	9.78s	21.46μW	2.50s	3.9×
c5315	82.52μW	9.15μW	57s	26.81μW	12.13s	24.87μW	3.27s	3.7×
c6288	214.42μW	121.42μW	121s	76.40μW	4m 20.19s	74.96μW	1m 9.94s	3.7×
c7552	136.04μW	14.40μW	68s	36.75μW	27.63s	34.52μW	7.26s	3.8×
b01	8.08μW	1.94μW	40s	3.64μW	0.021s	4.34μW	0.014s	1.5×
b02	5.43μW	1.75μW	41s	2.91μW	0.014s	2.10μW	0.011s	1.3×
b03	4.84μW	48.35nW	40s	2.40μW	0.075s	2.72μW	0.036s	2.1×
b04	50.92μW	9.36μW	47s	19.00μW	2.43s	14.57μW	0.755s	3.2×
b05	51.26μW	2.28μW	45s	9.30μW	4.20s	9.05μW	1.117s	3.8×
b06	11.63μW	1.19μW	40s	3.75μW	0.028s	3.75μW	0.016s	1.8×
b07	28.81μW	4.95μW	46s	8.49μW	1.553s	8.41μW	0.482s	3.2×
b08	26.94μW	3.94μW	36s	1.56μW	0.224s	1.57μW	0.076s	3.0×
b09	20.63μW	6.82μW	43	5.05μW	0.191s	5.70μW	0.068s	2.8×
b10	13.72μW	1.17μW	42	4.01μW	0.197s	3.99μW	0.071s	2.8×
b11	78.24μW	18.20μW	43s	9.20μW	3.70s	11.77μW	1.089s	3.4×
b12	96.90μW	27.21μW	46s	13.69μW	12.579s	14.68μW	3.484s	3.6×
b13	26.94μW	3.06μW	42s	4.84μW	0.76s	3.87μW	0.251s	3.0×
b14	426.87μW	173.75μW	144s	114.58μW	7m 33.16s	114.04μW	2m 2.35s	3.7×
b15	436.57μW	92.65μW	166s	86.13μW	14m 2.73s	85.07μW	3m 27.42s	4.1×
b17	1.04mW	265.87μW	384s	131.59μW	141m 37s	132.66μW	31m 58.38s	4.4×
b18	2.14mW	897.23μW	859s	442.40μW	23h 12m 23s	435.89μW	5h 13m 11.18s	4.5×
b19	4.12mW	1.80mW	1280s	687.23μW	126h 22m 43s	800.09μW	12h	10.5×
b20	703.20μW	109.73μW	209s	91.36μW	30m 14.92s	88.12μW	7m 9.69s	4.2×
b21	690.23μW	64.47μW	216s	120.46μW	29m 14.16s	107.92μW	7m 8.75s	4.1×
b22	1.02mW	302.57μW	328s	153.13μW	63m 20.0s	146.70μW	15m 7.55s	4.2×
Average	-	-	-	-	-	-	-	3.5×

timing updates. All programs are single-threaded and were run on an Intel core i7 64-bit machine, with a 16 GB RAM and running at 3.4 GHz.

B. Results

Table I shows the results for the experiment conducted using a fixed window size for *FastReplace*. The solution produced by the *FastReplace* algorithm is compared with the solutions provided by the greedy technique [13] and the mixed V_t synthesis performed using the Synopsys Design Compiler (Commercial) tool, from leakage power and running time perspective. We see that for most of the benchmark circuits, *FastReplace* provides a faster solution when compared to the greedy technique, without degrading the solution quality. *It can be seen that the method is most effective for larger circuits*. The method may not always improve the leakage power, as seen in the case of some smaller benchmark circuits. This is because for smaller benchmark circuits, the room for optimization is already less, which can further be degraded by replacing multiple gates with *lazy evaluation*. However, this phenomenon is not seen for larger benchmark circuits, due to the higher scope for optimization available in those cases, compared to the smaller benchmark circuits. The running time however improves consistently across all the benchmark circuits.

As mentioned in section IV-B, since the slack distribution keeps varying across the iterations, it is desirable to adapt the gate replacement window size across the iterations. Thus, using a fixed window size across all the iterations, may not always provide the best speed-up possible. To overcome this limitation, we scale up/down the window size using timing violation as a measure of confidence. Table II shows the additional speed-up obtained by implementing this adaptive sizing policy. It can be noted in Table I that for larger benchmark circuits, the running time of *FastReplace* is slightly higher when compared to the commercial tool. However, this extra time is spent in reducing the leakage power. To allow for a fair comparison of running time of *FastReplace* with the commercial tool, we analyze the time taken by *FastReplace* to provide the same leakage value. The results for the same are presented in Table III. It can be seen here that *FastReplace* performs faster than the commercial tool, under iso-power condition.

TABLE III
RUNNING TIME COMPARISON WITH COMMERCIAL TOOL, UNDER ISO-POWER CONDITION

	Running Time for Tool (Mixed V_t Synthesis)	FastReplace with Adaptive Window Sizing	
Ckt		Running Time	Speed-Up
b15	166s	71.0	2.3×
b17	384s	102.0	3.8×
b18	859s	374.0	2.3×
b19	1280s	810.0	1.6×
b20	209s	58.0	3.6×
b21	216s	117.0	1.9×
b22	328s	54.0	6.1×
Average	-	-	3.1×

TABLE II
TOTAL SPEED-UP OF FASTREPLACE WITH ADAPTIVE WINDOW SIZING, AS COMPARED TO ITERATIVE GREEDY REPLACEMENT

	FastReplace with r = 5		FastReplace adaptive sizing		Additional Speed-Up due to adaptive sizing (S_2)	Total Speed-Up over Greedy ($S_1 \times S_2$)
Ckt	Leakage	Runtime	Leakage	Runtime		
c3540	21.46μW	2.496s	22.29μW	1.82s	1.4×	5.4×
c5315	24.87μW	3.274s	24.55μW	2.369s	1.4×	5.1×
c6288	74.96μW	1m 9.94s	75.88μW	46.503s	1.5×	5.6×
c7552	34.52μW	7.262s	34.91μW	5.652s	1.3×	4.9×
b01	4.34μW	0.014s	3.96μW	0.025s	0.6×	0.8×
b02	2.10μW	0.011s	2.11μW	0.011s	1.0×	1.3×
b03	2.72μW	0.036s	2.53μW	0.035s	1.0×	2.1×
b04	14.57μW	0.755s	16.05μW	0.604s	1.3×	4.0×
b05	9.05μW	1.117s	7.94μW	0.760s	1.5×	5.5×
b06	3.75μW	0.016s	4.02μW	0.025s	0.6×	1.1×
b07	8.41μW	0.482s	9.45μW	0.388s	1.2×	4.0×
b08	1.57μW	0.076s	1.53μW	0.062s	1.2×	3.6×
b09	5.70μW	0.068s	6.04μW	0.060s	1.1×	3.2×
b10	3.99μW	0.071s	3.63μW	0.062s	1.2×	3.2×
b11	11.77μW	1.089s	10.10μW	0.782s	1.4×	4.7×
b12	14.68μW	3.484s	12.81μW	2.627s	1.3×	4.7×
b13	3.87μW	0.251s	3.28μW	0.236s	1.1×	3.2×
b14	114.04μW	2m 2.35s	115.14μW	76s	1.6×	6.0×
b15	85.07μW	3m 27.42s	85.77μW	1m 56s	1.8×	7.3×
b17	132.66μW	31m 58.38s	133.36μW	4m 56s	6.5×	28.7×
b18	435.89μW	5h 13m 11.18s	437.87μW	40m 19s	7.8×	34.5×
b19	800.09μW	12h	740μW	2h 49m 26s	4.3×	44.8×
b20	88.12μW	7m 9.69s	83.02μW	1m 26s	5.0×	21.1×
b21	107.92μW	7m 8.75s	110.22μW	1m 58s	3.6×	14.9×
b22	146.70μW	15m 7.55	154.85μW	3m 34s	4.2×	17.8×
Average	-	-	-	-	2.0×	9.8×

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed *FastReplace*, an efficient V_t replacement algorithm for leakage power minimization, using adaptive lazy timing analysis. The proposed algorithm provides 9.8× speed-up, without impacting the solution quality, as compared to the iterative greedy replacement technique. Additionally, under iso-power condition, we observe that the proposed algorithm performs 3.1× faster than the commercial tool. The following is the ongoing work, based on the results we have obtained for *FastReplace*

- In our algorithm, the undo procedure is highly pessimistic. This limits the speed-up that is achievable even with an adaptive window sizing mechanism. An interesting future work, is to reduce the backtracking involved in undoing an entire window, by efficiently identifying the *first violating gate* within the window.
- Our algorithm does not address within-die process variation. Statistical V_t replacement using adaptive lazy timing analysis is also under progress.
- There is scope for parallelization in *FastReplace*, by simultaneously exploring mutually exclusive fanin and fanout cones, for gate replacements and timing updates. Existing works like [12] explored parallelization using multithreading. We are exploring the scope for both parallel and concurrent realizations of the proposed algorithm.
- We would like to extend *FastReplace* for improving the runtimes of sensitivity driven metaheuristics, without affecting the solution quality.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions", IEEE JSSC ,vol.9 ,no.5 ,1974, pp.256-268.

[2] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir and V. Narayanan, V, "Leakage current: Moore's law meets static power", IEEE Computer , vol.36, no.12, 2003, pp.68-75.

[3] Y. Feng and S. Hu,"The epsilon-approximation to discrete VT assignment for leakage power minimization", International Conference on Computer Aided Design, IEEE/ACM, 2009, pp.281-287.

[4] D. A. Papa, M. D. Moffitt, C. J. Alpert and I. L. Markov, "Speeding Up Physical Synthesis with Transactional Timing Analysis", IEEE Design and Test of Computers, vol.27, no.5, 2010, pp.14-25.

[5] J. Lee and D.T. Tang, "An Algorithm for Incremental Timing Analysis", Design Automation Conference, IEEE/ACM, 1995, pp.696-701.

[6] R.P. Abato et al., "Incremental Timing Analysis", US patent, 5,508,937, to IBM Corp., Patent and Trademark Office, 1996.

[7] S.S. Sapatnekar, "Efficient Calculation of All-Pairs Input-to-Output Delays in Synchronous Sequential Circuits", International Symposium on Circuits and Systems, IEEE, 1996, pp.724-727.

[8] A. Mondal and C.A. Mandal, "A New Approach to Timing Analysis Using Event Propagation and Temporal Logic", Design, Automation and Test in Europe ,IEEE, 2004, pp.1198-1203.

[9] D. Das et al., "FA-STAC: A Framework for Fast and Accurate Static Timing Analysis with Coupling", International Conference on Computer Design, IEEE, 2006, pp.43-49.

[10] D. A. Papa et al., "Rumble: An Incremental, Timing-Driven, Physical-Synthesis Optimization Algorithm", International Symposium on Physical Design, ACM, 2008, pp.2-9.

[11] T. H. Cormen, C. Stein, R. L. Rivest and C. E. Leiserson, "Introduction to Algorithms", 2^{nd} Edition, 2001, McGraw-Hill Higher Education.

[12] J. Hu, A. B. Kahng, S. Kang, M.C. Kim and I. L. Markov, "Sensitivity-guided metaheuristics for accurate discrete gate sizing", International Conference on Computer Aided Design, IEEE/ACM, 2012, pp.233-239.i

[13] S. Mok, J. Lee and P. Gupta, "Discrete sizing for leakage power optimization in physical design: A comparative study", ACM Transactions on Design Automation of Electronic Systems, Vol. 18, No. 1, 15 (2012).