Post-Layout Sizing for Leakage Power Optimization: A Comparative Study

Santiago Mok
Advisor: Puneet Gupta
Electrical Engineering Department, University of California Los Angeles
{smok,puneet}@ee.ucla.edu

1. Introduction

Sizing is a widely-used method to tune design parameters (i.e. gate width, threshold voltage) to meet timing, power, and signal integrity constraints. Compared to synthesis, placement, and routing, sizing can help to meet these constraints with a minimal effect on the overall design.

In library-based designs, the gate sizing problem amounts to choosing an appropriate size from the cell library for each of the gates in the design. This problem has been studied extensively, however as the problem is NP-hard [1], optimality has been difficult to show. Variety of approaches have been proposed to the sizing problem.

A rich literature [2, 3, 4, 5, 6, 7, 8, 9, 10] studies the sizing framework for area/power/delay optimization. [2, 3, 4] apply a greedy heuristic to optimize power/area/delay subject to delay/area constraints. [5] studies a sensitivity-based heuristic. The sizing problem has been formulated as a linear program problem in [6, 7]. [7] solves the problem as continous optimization and discretize the solutions. Slack allocation-based linear program is employed in [8]. Lagrangian relaxation based optimization is proposed in [9, 10].

Though many approaches have been proposed for the discrete sizing problem, there is no common delay model which makes comparison between methods hard. [2, 3, 9, 10] approximate timing using Elmore delay. [5] uses regionwise quadratic delay model in circuit timing and Elmore delay in estimating sensitivities. [6, 7] assume gate delay as a linear function of gate size or threshold voltage. [8] employs SPICE-characterized rise/fall delay as function of load.

The demand for low power design and aggresive time-to-market schedule require late-stage (post-layout) optimization with accurate design information and minimal design alterations. Gate sizing has been the desirable technique to achieve low power design but an accurate timing model that account for slew effects is indispensable, as designer cannot afford timing violation.

Many sizing work from the literature focus on circuit-level optimization and the effects from post-layout (post place and route) has not been carefully addressed. Additional constraints must be considered in post-layout stage optimization: 1) library-based design is the de facto standard where a set of discrete gate sizes is available; hence, no assumption on continuous sizing can be made. 2) As routing has not scaled so aggresively compare to devices, wire contribute significant capacitive loading; as a result, gate delay is impacted. Wire loading and wire delay have significant impact on overall design timing that cannot be ignored.

Due to these contraints, an accurate timer is an essential tool in the sizing problem. Gate delays are often taken as cost metric in many optimization framework. In certain delay-sensitive optimization, an inaccurate timing model could lead to significant error. In this paper, we will compare discrete gate sizing methods implemented with accurate delay model comparable to the commercial timer used for timing sign-off. We also propose a new incremental gate sizing framework which has not been studied in the sizing literature.

The remainder of the paper is structured as follows. Section 2 discusses commonly employed sizing optimization methodologies. We introduce a new incremental sizing framework in Section 3. Experimental results are discussed in Section 4 and we conclude our finding in Section 5.

2. Discrete Gate Sizing Methodologies

In this section, we discuss three widely studied gate sizing algorithms: greedy, linear programming (LP) and Lagrangian relaxation (LR). Greedy is widely used for the

simplicity of its implementation. The main drawback for any greedy heuristic is that they might be easily trapped in local minima. On the other hand, LP and LR are both mathematical-based algorithms that were proposed to avoid the pitfall of greedy optimization. The details of each algorithm is discussed in the next subsections.

2.1 Greedy Algorithm

The greedy algorithm is a problem solving heuristic that evaluates the most cost-effective solution at each stage of the optimization. Extensive gate sizing optimization framework has been centered around greedy heuristic and sensitivity-based greedy heuristic [2, 11, 4, 3, 12, 5]. An early work, TILOS [2] optimizes area by assuming a minimum-sized circuit is allowed and iteratively sizes transistors in the critical path until timing is met.

We implemented a greedy heuristic similar to [2] that optimizes leakage power based on $(\Delta\mathrm{Power}/\Delta\mathrm{Delay})$ sensitivity function. The sensitivity metric measure the improvement in power per unit of delay. In our implementation, the initial design is a timing feasible (timing optimized) design, as in [12]. The main optimization heuristic trades gate delay for leakage power in decreasing sensitivity values. For each gate size change, an incremental static timing analysis is carried out to ensure timing is met. This heuristic iterates until no further gate size changes is possible that would meet timing. The flow of the algorithm is shown in Figure 1 and the details are explained below.

1. **Create Sensitivity List:** For each gate in the circuit, compute the $(\Delta\mathrm{Power}/\Delta\mathrm{Delay})$ sensitivity value with respect to each of the downsizes available.
2. **Evaluate:** For each sensitivity in descending order, change the current gate to the new gate's size. Perform incremental timing analysis and verify timing feasibility. If it does not meet timing, reverse gate to previous sizing; otherwise retain new sizing.
3. **Update and Iterate:** Note that during the evaluate phase, for any sizing modification in the circuit, the affected sensitivity values are not updated for performance purposes. Sizing a gate affects its fanin loading and its output transition time. The evaluate phase loops through all power-improving moves until the list is exhausted. If any sizing changes were performed, the sensitivity list is updated to capture the new configuration from previous iteration and the evaluate phase is re-iterated. Othewrise, the heuristic terminates as no further downsize is possible.
4. **Single Iteration Greedy:** A variant of this greedy heuristic is to terminate after evaluating through the sensitivity list once. This provide speed-up when no significant benefit is gained from additional iteration such as in Vt assignment.

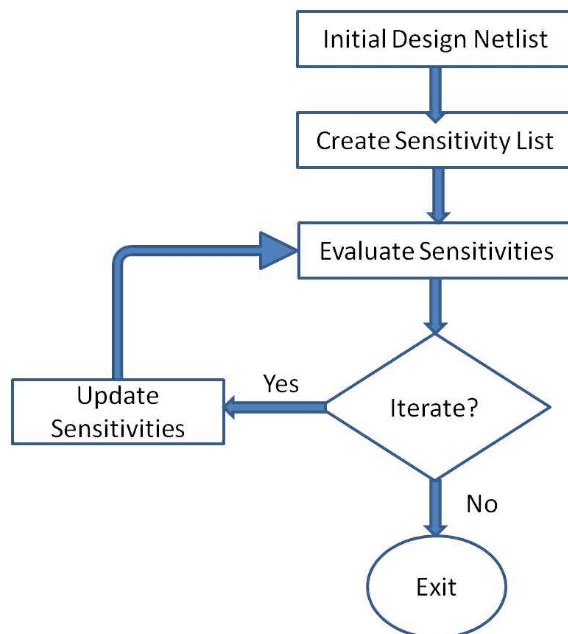

Figure 1. Greedy algorithm

2.2 Linear Programming

The linear programming (LP) problem, in general, is formulated as maximizing a linear function subject to linear constraints. In academia, LP has been widely studied in gate sizing and Vth assignment context [6, 8, 7, 13, 14].

Gate sizing technique in [8] has claimed remarkable power improvement using a slack allocation-based LP. Similar to our greedy approach, $(\Delta\mathrm{Power}/\Delta\mathrm{Delay})$ sensitivity value is assigned for each library size available from each gate in the circuit. Based on the power sensitivity value, the linear program smartly distributes slack to each individual gate with the objective of minimizing leakage power. In contrast to the greedy method where maximum delay on highly sensitive gates are traded for power, this method may assign more delay to those gates that are bottlenecks and common to multiple critical paths.

In this comparative study, we implemented the slack allocation-based LP algorithm; the flow of the algorithm is shown in Figure 2 and the details are explained below.

1. **Create Sensitivity List:** For each gate in the circuit, compute the $(\Delta\mathrm{Power}/\Delta\mathrm{Delay})$ sensitivity value with respect to each possible downsize cell that quantifies maximum power savings per unit delay cost.
2. **Slack Allocation:** Taking the measured sensitivity into consideration, LP is solved to distribute slack

among gates in the circuit. The LP formulation is given as follow for a circuit $\mathcal{G}$ with *n* gates:

$$\begin{array}{lll} \text{minimize} & \sum\limits_{n} d_n^{+} \frac{\Delta P_n}{\Delta D_n}, & \forall n \epsilon \mathcal{G} \\ \text{subject to} & a_i + d_i + d_n^{+} \leq a_n, & \forall n \epsilon \mathcal{G}, \forall i \epsilon fanin \\ & 0 \leq d_n^{+} \leq T_{max}, & \forall n \epsilon \mathcal{G} \\ & a_j \leq T_{target}, & \forall j \epsilon PO \end{array} \quad (1)$$

where d_n^+ is the additional delay assigned to gate n, a_i is the arrival time and d_i is the gate+wire delay with respect to fanin i, a_n is the arrival time at output of gate n, a_j is arrival time at the primary output and T_{target} is the target delay.

3. **Sizing Assignment:** Once the slack is allocated, the best configuration (minimum power) is assigned such that the allocated slack is not violated. This is done by choosing the lowest power size that changes the slack to a difference that is within the slack allocated to the gate. As new sizes alter slew and load in the fanin and fanout cone, an incremental static timing is triggered for each gate size change to ensure that circuit timing constraint is met.
4. **Update and Iterate:** Due to the interactions between gates, slack may remain after an initial assignment. Steps 1, 2, and 3 are re-iterated with updated sensitivity list that capture changes in the new configuration. The heuristic iterate until the power objective converges and no further power improvement is possible.

2.3 Lagrangian Relaxation

Lagrangian Relaxation is another mathematical based nonlinear optimization method that has been adapted to solve the sizing problem [10, 9]. For power optimization, [9] define the lagrangian problem for circuit $\mathcal{G}$ with *n* gates:

$$\begin{array}{ll} \text{minimize} & \sum\limits_{x \epsilon \mathcal{G}} p(x) + \sum\limits_{I\ \epsilon\ PI} \lambda_i (a_i - q_i) \\ & + \sum\limits_{\forall (x_j, x_i)\ \epsilon\ \mathcal{G}} \lambda_{ji} (q_j + D(x_{ji}) - q_i) \\ \text{subject to} & X_{min} \leq x_n \leq X_{max} \\ & q_i \geq a_i \\ & q_i \geq D(x_{ji}) + q_j \end{array} \quad (2)$$

where:

- $p(x)$ is the leakage power for gate *x* in the design $\mathcal{G}$
- $D(x_{ji})$ is the delay of gate *x* through fanin(*i*)
- λ_{ji} is the lagrangian multiplier through fanin(*i*)
- (x_i, x_j) is the interconnect between gate x_i and x_j
- x_n is the vector of gate sizes
- a_i is the arrival time at fanin(*i*)

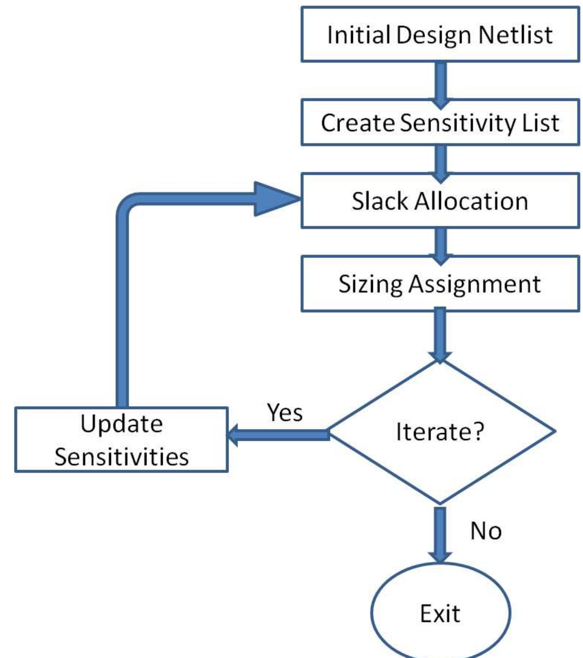

Figure 2. Slack Allocation-based LP

- q_i is the require arrival time at input of gate(*i*)

The lagrangian relaxation method reduces the number of timing constraints in (2) by embedding them as cost function into the lagrangian multipliers (λ). The reduction is made possible by deriving the "Kuhn-Tucker" optimality conditions on λ (proved in [10]) that lead to the flow-like requirement in (3). The condition on λ is indepedent of the arrival and require arrival time at the gate nodes. With (3), the lagrangian function in (2) reduces to the weighted sum of power and delay in (4).

$$\sum_{i\ \epsilon\ fanins(x_k)} \lambda_i = \sum_{j\ \epsilon\ fanouts(x_k)} \lambda_j \quad (3)$$

$$\begin{array}{lll} \text{minimize} & \sum\limits_{x\ \epsilon\ G} p(x) + \sum\limits_{i\ \epsilon\ fanin} \lambda_{ij} D(x_{ij}) & \\ \text{subject to} & X_{min} \leq x_n \leq X_{max}, & \forall n \end{array} \quad (4)$$

The lagrangian-based heuristics that we will compare is based on "Timing-Constrained Power Optimization" in [9] which is a Dynamic Programming (DP) guided lagrangian relaxation heuristic. Lagrangian relaxation method does not rely on sensitivity metrics which some claim that sensitivity model leads to local minima. This algorithm offer two main advantages over other continuous lagrangian-based optimization: (1) This algorithm can be easily adapted to look-up table timing model which avoid fitting data to continuous timing model that is subject to inaccurate delay; (2)

It avoids snapping solution to a discrete size which is also prone to error.

In brief, the algorithm solves two problems: 1) the Lagrangian relaxation subproblem (4) guided by DP, and 2) the Lagrangian dual problem which tune the Lagrangian multipliers (λ) to optimize the Lagrangian function. The Lagrangian function serves as the objective for DP. The DP perform a reverse topological propagation from primary outputs (PO) to primary inputs (PI) to compute the partial lagrangian function; which is then followed by a forward topological search from PI to PO that assigns the gate size that optimizes the Lagrangian function.

The Lagrangian relaxation subproblem is solved by recursively computing the partial weighted Lagrangian function (5) at each gate for each size option. At each gate's output node, the candidate solutions from its fanouts are merged and pruned based on the criteria in (7). Once all candidate solutions are generated for each gate, a forward topological search is performed to assign the gate size that minimizes the weighted objective function (6).

$$L(x_i^n) = \sum_{\mathrm{x_j}\,\epsilon\,\mathrm{fanout(x_i)}} \min_{m\,\epsilon\,vec(x_j)} \{p(x_i) + \lambda_{ij} D(x_{ij}^m) + L(x_j^m)\} \tag{5}$$

$$solution(x_i) = \min_{m\,\epsilon\,vec(x_i)} [L(x_i^m) + \sum_{x_j\,\epsilon\,fanin(x_i)} \{\lambda_{ji} D(x_{ji}^m) + p(x_h)\}] \tag{6}$$

$$if\ c(x_i^n) \geq c(x_j^m) \text{ and } L(x_i^n) \geq L(x_j^m) \tag{7}$$

The Lagrangian dual problem updates the Lagrangian multipliers in equation (8) simlar to the subgradient method based on the input slack and α factor that facilitate convergence. The overall algorithm is shown in Algorithm (1)

$$\lambda_i^+ = \lambda_i + \alpha(-\mathrm{slack(x_i)}), \quad \mathrm{i}\,\epsilon\,\mathrm{fanin(x_i)} \tag{8}$$

3. Peephole

One area that deserves attention is incremental gate sizing. This has not been studied in prior literarture. In this section, we present a method to perform incremental gate sizing. This method focuses on optimizing portions of the design, which we call *peepholes*, which can unlock additional leakage power improvement of an optimized design

Definition 1. *A* peephole *is a collection of a gate (the* root*) and an arbitrary collection of its fanouts.*

Algorithm 1 DP_LR()

```
1:  λ ← 0
2:  for all x ϵ G in reverse topological order do
3:      for all size option(n) in vec(x_n) do
4:          for all x_j ϵ fanout(x) do
5:              obj(x_i^n) ← L(x_i^n) fromequation (5)
6:          end for
7:      end for
8:      Prune candidates based on 7
9:  end for
10: for all x ϵ G in topological order do
11:     if x_i ϵ PI then
12:         solution(x) ← min_{k ϵ option(x^n)} obj(x_i^k)
13:     else
14:         solution(x) ← equation (6)
15:     end if
16: end for
17: Update λ
```

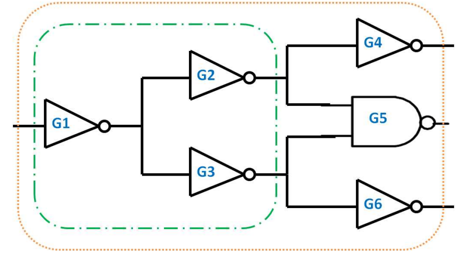

Figure 3. Peephole Example

For the example in Figure 3 there are many peepholes associated with "G1". One peephole could be $\mathcal{P} = \{G1, G6\}$, and another could be $\mathcal{P} = \{G1, G3, G6\}$. From the graph, there are also natural peepholes that can be made, for example, the root gate and its immediate fanouts ($\mathcal{P} = \{G1, G2, G3\}$), and the root gate and its second level fanouts ($\mathcal{P} = \{G1, G4, G5, G6\}$). These peepholes play a key part in the optimization, and we formalize these natural peepholes below:

Definition 2. *An nth* degree peephole *consists of the peephole formed by the root, and the fanouts that are at least n levels from the root.*

Note that the nth degree peephole excludes the gates that are less than n levels from the root. For example, in Figure 3 the 2nd-degree peephole is $\mathcal{P} = \{G1, G4, G5, G6\}$ and excludes gates $\mathcal{P} = \{G1, G2, G3\}$.

The peephole optimization focuses on upsizing the root gate, and downsizing the fanouts in the peephole. The main idea is to create slack by upsizing the root gate, which is used downstream to downsize *multiple* gates. This improves leakage power by downsizing *multiple* gates at the expense

of increasing the leakage power of the single root gate, and yields a net leakage power improvement. This allows leakage power improvement in cases where it is not possible when only a single gate is evaluated.

n^{th}-degree peepholes with $n > 1$ are considered because the slack from upsizing a root propagates through the path but its immediate fanouts may not be able to take advantage of these slacks. Thus, in the algorithm in this section, we focus on n^{th}-degree peepholes for $n \in \{1, 2, 3\}$.

An overview of the peephole optimization algorithm is given in the next section and the rest of this section we proceed to discuss in details each steps of the algorithm.

3.1 Peephole Optimization Algorithm

The overall flow for the peephole is:

1. Start with a list of all peepholes for the circuit.
2. Prune the peephole list
3. Rank peepholes according to their size-ability
4. Optimize the peepholes in decreasing order of rank
5. Recover the additional slacks created by step (4) using a greedy sizing method.

3.2. Pruning the peepholes

The number of nth degree peepholes in a circuit is prohibitively large. It is approximately the number of gates times the depth of the circuit. As the number of peepholes is large, pruning is necessary to manage runtime and avoid effort wasted in peepholes that are unlikely to give a power reduction. The following conditions are verified before any peephole is accepted for sizing optimization.

- **Restriction to** $n \in \{1, 2, 3\}$: Substantial amount of time is taken and no considerable improvement is achieved when 4th and higher degree fanouts are considered. Thus, we restrict the peepholes up until 3rd degree fanouts.

- **Up-Sizeable Root**: Since our optimization relies on upsizing the root $r \in \mathcal{P}$ to create slack for its fanout, any peephole that contains a root with maximum size is removed.

- **Timing-Infeasible Root**: It is not always the case that upsizing a gate will create additional slacks on the path. When a gate is upsized, the driver of this gate encounters a larger load at its output. The larger load can slow down the driver and eventually slow down the path. Consequently, if the output arrival time is worsen when the root $r \in \mathcal{P}$ is upsized, then any peephole with such root is removed.

- **Down-Sizeable Fanout**: Another peephole constraint is the fanout gates are downsized only. Each fanout $j \in \mathcal{P}$ that is already minimum size is removed from the peephole for optimization.

In addition, we prune peepholes during the optimization process:

- **Optimized Peephole**: From our extended peephole definition, up to 3 peepholes can be associated with a gate in the circuit. Recall that the extended peephole definition takes into account transitive fanouts in case that immediate fanouts were not able to be optimized; if any 1 of 3 peephole is optimized, not enough slack is available to optimize another peephole with the same root. As the algorithm proceed, once $\mathcal{P}(r, j)$, peephole with root r and fanouts j is optimized, further peephole $\mathcal{P}(r, k)$ down in the ranked-list are removed.

3.3 Peepholes Ranking

After the pruning process, the peephole are ranked in order of the likelihood that it will provide power savings after optimization. This is done using two metrics – timing feasibility and sizeability.

The timing feasibility measures the likelihood that the root upsize – fanout downsize combination will be timing feasible:

Definition 3. *The timing Feasibility of a root, r and fanout j is measured as:*

$$\text{Feasibility}(r, j) = \Delta^{(+)}\text{delay}(r) - \Delta^{(-)}\text{delay}(j) + \text{slack}(j) \tag{9}$$

where $\Delta^{(+)}\text{delay}(r)$ is the change in the delay of gate r when it is upsized (to the next available size), $\Delta^{(-)}\text{delay}(j)$ is the change in the delay of gate j when it is downsized (to the next available size), and $\text{slack}(j)$ is the output slack of gate j. The gate delay is estimated as the time required to drive the fanout load with the current input slew and output load configuration.

The size-ability of a peephole $\mathcal{P}$ can then be defined using the timing Feasibility of its members:

$$\text{Sizeability}(\mathcal{P}) = \text{slew}(r) \cdot \sum_{i \in \mathcal{P}} \mathcal{W}(\text{Feasibility}(r, i)) \tag{10}$$

where

$$\mathcal{W}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

A weight $\mathcal{W} = 1$ is added for each fanout $j \in \mathcal{P}$ that is has a positive feasibility. The weighted sum is then multiplied with output slew of root $r \in \mathcal{P}$, which accounts for propagated arrival time effects. This is because when a root

with a high (slow) slew is upsized, the downstream slew will decrease (speed up), improving the probability that the peephole optimization will be successful.

3.4 Peephole Evaluation

The peepholes are evaluated in the ranking order in Section 3.3. The evaluation checks if a peephole $\mathcal{P}(r, j)$ yields leakage power improvement without violating timing. The pseudo-code for the evaluation process is shown in (2).

In the evaluation process:

1. The root r is upsized to the next available size.

2. Fanouts $j \in \mathcal{P}$ are sorted according to its timing feasibility estimates using (9).

3. For each $j \in \mathcal{P}$, j is downsized until it cannot be downsized without violating the timing constraint.

4. If the net leakage power has improved, the current size is retained. Otherwise, the sizes of the fanouts are reverted and goes to step 2.

Algorithm 2 Evaluate($\mathcal{P}(r, j)$)

```
1:  while Up − Sizeable(r) do
2:    UpSize(r)
3:    for all j ∈ P do
4:      j ← minimum size that meet timing
5:    end for
6:    if NetLeakagePowerImprove(r,j) then
7:      retain current size; mark(r); break
8:    else
9:      reverse size; continue
10:   end if
11: end while
```

3.5 Algorithm Summary and Slacks Recovery

An overview of the peephole optimization is given in Algorithm 3. It begins by storing all the peepholes in a list L. In lines 2-3, pruning is performed, size-ability is computed and the peepholes are ranked (Sections 3.3 and 3.2). In lines 5-10, the peephole list is evaluated in decreasing rank order (Section 3.4) and the Evaluate algorithm (Algorithm 2) is executed.

After peephole optimization ends, excessive slack is leftover. Sensitivity-based greedy heuristic is applied after peephole optimization to take advantage of the slacks.

Algorithm 3 OptimizePeephole()

```
1:  L ← 1st, 2nd, and3rd degree peephole
2:  ComputeRankAndPrune(L)
3:  Sort(L)
4:  for all P ∈ L do
5:    r ← P(root); j ← P(fanouts)
6:    if r is optimized then
7:      continue
8:    end if
9:    EVALUATE(r, j)
10: end for
11: Do Sensitivity-based Greedy()
```

4. Experimental Setup and Results

We tested the algorithms on ISCAS'85 benchmark circuits. All benchmark circuits are synthesized with Cadence RTL Compiler. The design were optimized for leakage power and timing during synthesis. We used Cadence SoC Encounter to place, route, and extract interconnect RC (.spef). All design optimizations were disabled during placement and routing.

The benchmark circuits is synthesized to Nangate 45nm Open Cell Library [15] and a commercial 65nm ST library. In the Nangate library, there are 6 sizes for inverter and 3 sizes for other standard cells. In ST library, there are at least 6 sizes for all standard cells. In addition to gate sizing, we synthesized the circuits to different Vt from the ST library for Vt assignment. There are 3 Vt variants: high Vt, standard Vt, and low Vt.

($\Delta\text{Power}/\Delta\text{Delay}$) sensitivity-based greedy heuristic, slack allocation-based linear programming, Lagrangian relaxation, and Peephole algorithm are implemented in C++ using OpenAccess API [16] from Si2. We used an open-source OpenAccess-based timer from OAGEAR [17] package for static timing analysis (STA). The timer STA engine is comparable to commercial timer; the timer propagate both rise/fall transistion and timing, and it uses 2D-table lookup (slew,load) for delay model.

As a starting point, the benchmark circuits were optimized for timing using the dynamic program employed in [9] for maximizing slacks. The target timing constraint for all the benchmarks is 30% from minimum delay with respect to delay with minimum size configuration.

The results for the four algorithms are plotted in figure (4) and figure (6) for Nangate and ST libraries, respectively. In general, none of the three discrete sizing methods perform better than another under Nangate library even though the percentage difference among them is quite small. The % power improvement among the algorithms is more clearly plotted in figure (5) which is normalized with respect to the greedy algorithm. LP performs better on c432

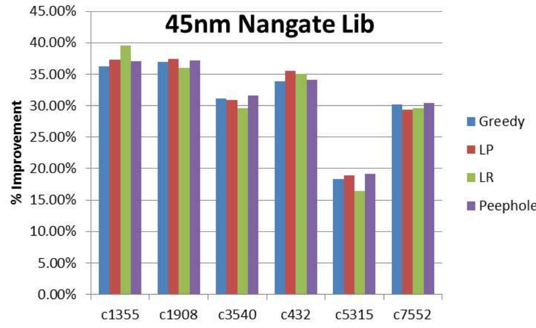

Figure 4. Gate Sizing with 45nm Nangate library

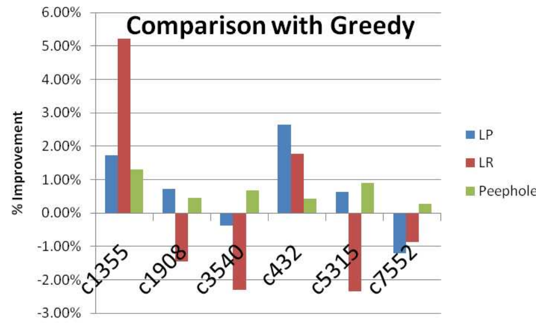

Figure 5. Power improvement compare to greedy (Nangate lib)

and c1908 while LR outperfom other methods by more than 3% on c1355. Greedy leads in c3540 and c7552 by a mere 1% difference.

Gate sizing with ST library indicate more interesting results. LR algorithm yields the best leakage power in all six benchmark circuits in Figure (7). Normalized with respect to greedy, LR leads by as high as 11% in c1355 and 4% on average over the benchmark suite . On the other hand, LP's results are the least among most of the circuits. For the designs synthesized with ST library, there is a wider spread in delay between the minimum timing and minimum size configuration. The wider delay spread along with more discrete sizes in the ST lib facilitate convergence with better solution quality in the LR algorithm.

In Vt assignment, benchmarks using greedy show much better power results over LP and LR shown in Figure (9). The suboptimality of LP in exponential power tradeoff is quite high which confirm the result in [18]. From the results in Nangate library and Vt assignment, LR require larger set of discrete sizes to yield significant power improvement.

With respect to Peephole algorithm, the power improvement shown in Figure (11) over greedy is under the 3%

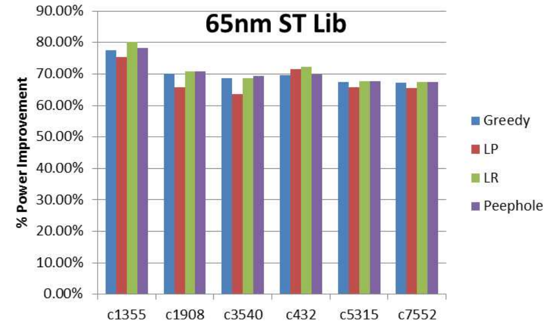

Figure 6. Gate Sizing with 65nm ST library

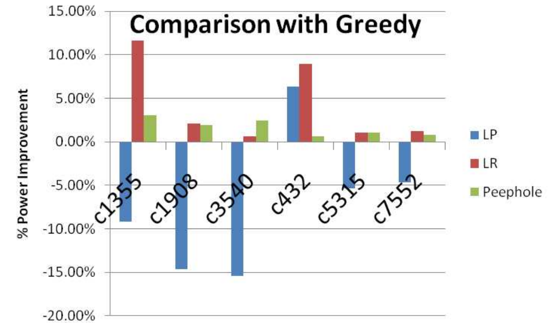

Figure 7. Power improvement compare to greedy (ST lib)

	Nangate (45nm)					ST (65nm) Library					ST (65nm) Vt Assignment				
		Runtime (s)					Runtime (s)					Runtime (s)			
iscas85	Gates Count	Greedy	LP	LR	Peephole	Gates Count	Greedy	LP	LR	Peephole	Gates Count	Greedy	LP	LR	Peephole
c1355	601	39	438	58	62	411	62	358	69	196	572	18	233	68	61
c1908	549	25	246	43	53	478	104	510	106	346	577	20	360	82	91
c3540	1086	71	1379	122	156	1006	273	1485	157	917	1377	82	1322	159	309
c432	183	3	27	25	6	134	9	24	22	21	198	4	47	27	14
c5315	1237	45	544	148	87	1094	157	792	188	383	1684	51	694	119	115
c7552	1982	92	1430	242	73	1756	318	1659	269	794	2515	113	1637	208	186

Figure 8. Algorithms Runtime in seconds

bound. Compare to Nangate library, larger improvement are seen on ST library with more discrete gate sizes and in ST for Vt assignment with exponential power tradeoff. However, the quality of the improvement in peephole is difficult to compare as the optimum delay is unknown and difference in best leakage power results among the other sizing method is quite small.

The runtime comparison is shown in Figure (8). LP algorithm has the worst runtime among the three sizing methods. In LP, the slack allocation is quite sensitive to the rough estimate of delta delay value; hence, static timing analysis (STA) is carried out for the sensitivity computation. Also after slack allocation, STA is triggered to search for the library gate assignment that fit the given allocated delay. In contrast, an estimate of the sensitivity value is computed in greedy and STA is carried out only in evaluating the gate size change. Similarly in LR, DP is solved with an estimate

delay and STA is performed for each sizing solution.

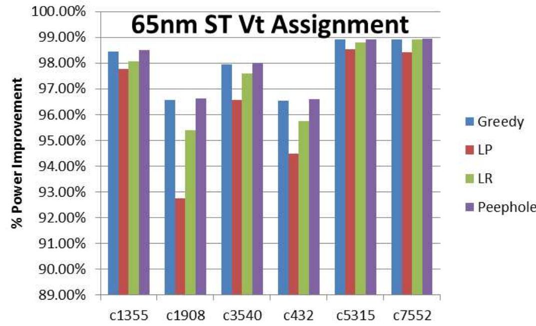

Figure 9. Vth assignment with 65nm ST library

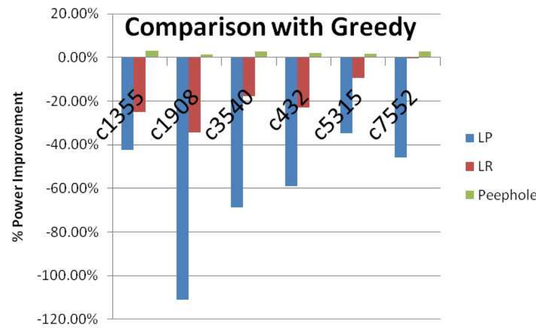

Figure 10. Power improvement compare to greedy (ST Lib-Vt)

	Peephole Improvement over Greedy		
iscas85	Nangate	ST	ST - Vt Assignment
c1355	1.31%	3.06%	3.15%
c1908	0.45%	1.93%	1.41%
c3540	0.67%	2.42%	2.55%
c432	0.43%	0.64%	2.09%
c5315	0.90%	1.10%	1.73%
c7552	0.28%	0.78%	2.79%

Figure 11. Peephole power improvement over greedy

5. Conclusion

We have implemented three distinct flavors of discrete sizing method commonly studied in the literature. A common timer that employs 2D-table lookup is used among the algorithms to make comparison fair. No algorithm perform better than other under the Nangate library. However, LR yields the best power improvement with gate sizing in ST library and greedy outperform in the Vt assignment.

Though we have not shown significant improvement with our peephole method, incremental gate sizing has not been studied in the literature and it should be an interesting area of research for late-stage optimization.

References

[1] W. Li, “Strongly np-hard discrete gate sizing problems,” *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 468–471, Oct 1993.

[2] J. Fishburn and A. Dunlop, “TILOS: A Posynomial Approach to Transistor Sizing,” *Proceedings of the 1985 International Conference on Computer-aided Design, Nov*, 1985.

[3] L. Wei, Z. Chen, K. Roy, and V. De, “Design and Optimization of Dual Threshold Circuits for Low Voltage Low Power Applications,” in *IEEE Trans. on Very Large Scale Integration Systems*, pp. 16–24, March 1999.

[4] O. Coudert, “Gate sizing for constrained delay/power/area optimization,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 5, pp. 465–472, Dec 1997.

[5] S. Sirichotiyakul, T. Edwards, C. Oh, R. Panda, and D. Blaauw, “Duet: an accurate leakage estimation and optimization tool for dual-Vt circuits,” in *IEEE Trans. on Very Large Scale Integration Systems*, pp. 70–90, 2002.

[6] M. R. C. M. Berkelaar and J. A. G. Jess, “Gate sizing in mos digital circuits with linear programming,” in *EURO-DAC ’90: Proceedings of the conference on European design automation*, (Los Alamitos, CA, USA), pp. 217–221, IEEE Computer Society Press, 1990.

[7] A. Srivastava, “Simultaneous vt selection and assignment for leakage optimization,” in *Proc. Int. Conf.*

Low Power Electronics and Design, pp. 146–151, 2003.

[8] D. Nguyen, A. Davare, M. Orshansky, D. Chinnery, B. Thompson, and K. Keutzer, “Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization,” in *ISLPED ’03: Proceedings of the 2003 international symposium on Low power electronics and design*, (New York, NY, USA), pp. 158–163, ACM, 2003.

[9] Y. Liu and J. Hu, “A new algorithm for simultaneous gate sizing and threshold voltage assignment,” in *ISPD ’09: Proceedings of the 2009 international symposium on Physical design*, (New York, NY, USA), pp. 27–34, ACM, 2009.

[10] C. Chen, C. Chu, and D. Wong, “Fast and exact simultaneous gate and wire sizing by lagrangian relaxation,” *IEEE Trans. on Computer-Aided Design*, vol. 18, no. 7, pp. 1014–1025, 1999.

[11] O. Coudert, R. Haddad, and S. Manne, “New algorithms for gate sizing: A comparative study,” *Design Automation Conference*, vol. 33, pp. 734–739, Dec 1996.

[12] S. Sirichotiyakul, T. Edwards, C. Oh, J. Zuo, A. Dharchoudhury, R. Panda, and D. Blaauw, “Stand-by power minimization through simultaneous threshold voltage selection and circuit sizing,” in *Proc. Design Automation Conference*, pp. 436–441, 1999.

[13] D. G. Chinnery and K. Keutzer, “Linear programming for sizing, vth and vdd assignment,” in *Proc. Int. Conf. Low Power Electronics and Design*, pp. 149–154, 2005.

[14] K. Jeong, A. Kahng, and H. Yao, “Revisiting the linear programming framework for leakage power vs performance optimization,” in *IEEE International Symposium on Quality Electronic Design*, pp. 127–134, 2009.

[15] “Nangate Open Cell Library v1.2.” Available from http://www.si2.org/openeda.si2.org/projects/nangatelib.

[16] Available from http://www.opencores.org.

[17] OAGear v0.98 available from http://www.si2.org/openeda.si2.org/projects/oagear.

[18] P. Gupta, A. Kahng, A. Kasibhatla, and P. Sharma, “Eyecharts: Constructive benchmarking of gate sizing heuristics,” in *DAC ’10: Proceedings of the 47th annual conference on Design automation*, 2010.