Rapid Gate Sizing with Fewer Iterations of Lagrangian Relaxation

Ankur Sharma
Iowa State University
Ames, USA
ankur@iastate.edu

David Chinnery
Mentor Graphics
Fremont, USA
david_chinnery@mentor.com

Shrirang Dhamdhere
Mentor Graphics
Fremont, USA
shrirang_dhamdhere@mentor.com

Chris Chu
Iowa State University
Ames, USA
cnchu@iastate.edu

***Abstract*—Existing Lagrangian Relaxation (LR) based gate sizers take many iterations to converge to a competitive solution. In this paper, we propose a novel LR based gate sizer which dramatically reduces the number of iterations while achieving a similar reduction in leakage power and meeting the timing constraints. The decrease in the iteration count is enabled by an elegant Lagrange multiplier update strategy for rapid coarse-grained optimization as well as finer-grained timing and power recovery techniques, which allow the coarse-grained optimization to terminate early without compromising the solution quality. Since LR iterations dominate the total runtime, our gate sizer achieves an average speedup of 2.5x in runtime and saves 1% more power compared to the previous fastest work.**

TABLE I: Acronyms and their meanings.

Acronym	Meaning
LR	*L*agrangian *R*elaxation
LRS	*L*agrangian *R*elaxation *S*ubproblem
LDP	*L*agrangian *D*ual *P*roblem
RGS	*R*apid *G*ate *S*izer
CPS	*C*ritical *P*ath *S*izing
MGS	*M*ulti-*G*ate *S*izing
SGS	*S*ingle *G*ate *S*izing
TNS	*T*otal *N*egative *S*lack
STA	*S*tatic *T*iming *A*nalysis

I. Introduction

In modern chip design methodologies, circuit optimization via gate sizing is regarded as one of the key techniques that needs to be invoked at several design stages to trade off various metrics such as timing, area, and power. Due to the large number of gates in a design, gate sizing can be very time consuming. In a standard cell based design, each gate can be implemented by many different options which are characterized by a size and a threshold voltage (Vt). Each option trades off power, area, and delay. The task of gate sizing is to assign a suitable option to each gate such that the desired objective is optimized under the given design constraints. In this work, we focus on timing constrained leakage power (hereafter referred to as power) minimization.

The problem of gate sizing has been studied for over three decades. Earlier the gate sizes were assumed to be continuous and the timing models were either derived from RC Elmore delay models, which can be transformed into a convex function of sizes, or approximated by a convex function. Under such scenarios, an optimal solution could be obtained by applying techniques like Lagrangian Relaxation (LR) [1]. However, such delay models are too inaccurate to achieve a good solution with modern process technologies - non-convex lookup table-based delay models have been industry standard for more than a decade. With discrete size and Vt options, the gate sizing problem is NP hard [2] and thus no polynomial time optimal algorithm is known. With millions of gates in designs and tens to hundreds of discrete options for each gate size, it can take a day or more of runtime to deliver acceptable solution quality.

For discrete gate sizing, researchers have presented several heuristics based on dynamic programming [3], sensitivity guided greedy frameworks [4], network flow [5], and LR based techniques like [6], [7], [8], [9]. After the ISPD 2012 gate sizing contest [10], several publications established the superiority of the LR based gate sizers, showing both faster runtime and lower power. However, the proposed LR based gate sizers take many LR iterations, even more than 100 on some benchmarks. The high number of iterations can be very detrimental for runtime, especially with expensive timing updates to account for the RC parasitics on large designs.

A reason for the high iteration count of the LR based gate sizers is that they are effective only for coarse-grained timing and power recovery. As the total negative slack (TNS) and the total leakage power of the design reduce, the efficiency of each iteration also degrades. Although LR based gate sizers are usually equipped with greedy post-pass heuristics for finer-grained timing and power recovery, they cannot be invoked too early as they are very time consuming and get stuck in a local minimum. The LR iterations need to go on until TNS and power are sufficiently small. Otherwise, the outstanding timing violations and the remaining potential power savings would be too large to be effectively handled by those greedy techniques. Therefore, we need strategies that can reduce the runtime of coarse-grained optimization by reducing LR iterations, and techniques for finer-grained optimization.

In this paper, we develop an LR based rapid gate sizer (RGS). We propose an elegant Lagrange multiplier update strategy that makes the coarse-grained LR based sizing converge very rapidly to a solution with sufficiently low TNS and power. We propose two LR-based techniques, one for finer-grained timing recovery, and the other for finer-grained power recovery. For timing recovery, our proposed technique is called critical path sizing (CPS), which reduces the delay along critical paths. For power recovery, our proposed technique is called multi-gate sizing (MGS), which sizes several gates simultaneously, unlike typical sizing heuristics employed by LR based sizers which size one gate at a time. While CPS is able to efficiently fix the timing violations that may occur during power recovery, MGS allows coarse-grained optimization to terminate early, thereby reducing the expensive LR iterations without compromising on the final solution quality. MGS can also potentially take the design out of a local minimum, thus creating opportunities for further power recovery. With these three techniques, the number of LR iterations is significantly lower than those in previous works. Since LR iterations dominate the total runtime, RGS achieves an average speedup of 3x compared to the previously fastest work [8].

Our major contributions are summarized as follows:

- We propose an elegant Lagrange multiplier update strategy.
- We propose two LR-based techniques, MGS and CPS, for fine-grained power and timing recovery, respectively.
- We develop a rapid gate sizing flow, and empirically verify its effectiveness.

This paper is organized as follows: Section II formulates the problem. Section III presents the overall flow of RGS and briefly discusses some of its components. Section IV describes the core

solver of RGS in detail. There we discuss our proposed techniques: Lagrange multiplier update strategy, MGS, and CPS. Section V discusses the empirical results, and we conclude in Section VI.

II. Problem Formulation

With our gate sizing algorithm, we solve the following constrained optimization problem:

Given a gate-level netlist, a standard cell library with discrete choices for cell size and threshold voltage (Vt), timing constraints, and lumped parasitics, compute the size and Vt combination (hereafter referred to as 'option') for each combinational gate in the netlist such that the leakage power is minimized without violating the timing constraints.

This is the same formulation as used in the ISPD 2012 gate sizing contest. For our experiments, we use the same setup including the set of benchmarks as provided in the contest. Per the contest guidelines, there are two types of electrical constraints: load at the output of a gate cannot exceed a maximum value (max load constraint), and slew at the input of a gate cannot exceed a maximum value (max slew constraint). Since computing the max slew violations is computationally more expensive than computing the max load violations, we translate max slew constraints into max load constraints - this can be done because a lumped wire capacitance model is used for the ISPD 2012 gate sizing contest. Thus ensuring that there is zero max load violation guarantees zero max slew violation.

Before formally defining the problem, we detail notations commonly used in this work. Table I lists acronyms that are commonly used throughout this paper. T is the target clock period. For gate i, x_i denotes the size/Vt option, and a_i denotes the arrival time at its output. $d_{i\to j}$ is the delay of the timing arc $i \to j$ which is defined from the output of the gate i to the output of the gate j. Endpoint of a timing path can be either a primary output of the design or input of a sequential element (e.g. flip-flop). Note that sequential elements have a fixed size as per the contest.

Mathematically, the above problem is commonly formulated as a non-convex, discrete mathematical program as:

$$\begin{aligned} \underset{x,a}{\text{minimize}} \quad & \sum_i leakage_i \\ \text{subject to} \quad & a_i + d_{i\to j} \le a_j,\ \forall\ i \to j \\ & a_k \le T\ ,\ \forall \text{ endpoints } k \end{aligned} \tag{1}$$

We refer to Equation (1) as the *Primal Problem* (PP). To solve this NP-hard problem, as in previous work [1], we relax the constraints and derive its Lagrangian dual. Each constraint is associated with a non-negative Lagrange multiplier, λ, that acts as a penalty for violating the respective constraint. The Lagrangian function, $L(x, a, \lambda)$ is:

$$\begin{aligned} L(x,a,\lambda) = & \sum_i leakage_i + \sum_{i\to j} \lambda_{i\to j}\left(a_i + d_{i\to j} - a_j\right) \\ & + \sum_{k\in endpoints} \lambda_k \left(a_k - T\right) \end{aligned} \tag{2}$$

For a given set of Lagrange multipliers, the Lagrangian relaxation subproblem $LRS(\lambda)$ is:

$$LRS(\lambda) : \underset{x,a}{\text{minimize}}\ L(x,a,\lambda) \tag{3}$$

By applying the Karush-Kuhn-Tucker (KKT) conditions for optimality, and omitting the term $\sum_k \lambda_k T$ since it is a constant for a given set of λ_k, the LRS(λ) can be simplified to:

$$LRS^*(\lambda) : \underset{x}{\text{minimize}} \quad \sum_i leakage_i + \sum_{i\to j} \lambda_{i\to j} d_{i\to j} \tag{4}$$

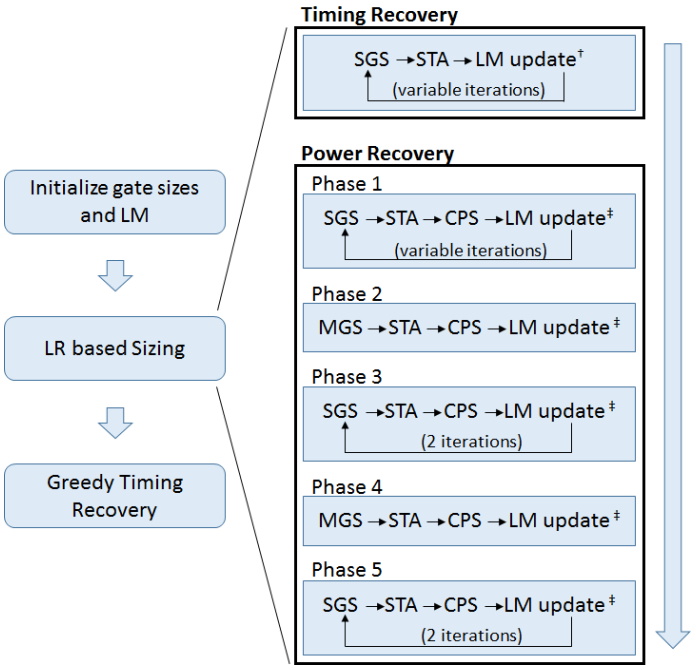

Fig. 1: RGS flow chart. Each iteration involving SGS and Lagrange multiplier (LM) update is referred to as an LR iteration. †: LM are updated to rapidly recover timing. ‡: LM are updated to rapidly recover power. During SGS of phase three and five, no gate is allowed to degrade power.

where Lagrange multipliers must satisfy the 'flow constraints':

$$\sum_{u\in fanin(i)} \lambda_{u\to i} = \sum_{v\in fanout(i)} \lambda_{i\to v},\ \forall i \tag{5}$$

The objective in Equation (4) is referred to as the *LRS cost*. Flow constraints in Equation (5) are obtained by setting the partial derivatives over the a variables to 0.

The Lagrangian dual problem (LDP) is then defined as:

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \quad & LRS^*(\lambda) \\ \text{subject to} \quad & KKT\ flow\ constraints \text{ and } \lambda \ge 0 \end{aligned} \tag{6}$$

Solving LRS*(λ) for a given set of λ gives a lower bound on the PP. And, solving LDP maximizes that lower bound. Optimal values of LDP and PP would match, if the duality gap is zero. A similar Lagrangian relaxation based formulation can be derived for other objective functions like area and dynamic power.

III. Overall Flow

Like several other LR-based gate sizers, our gate sizer RGS is also composed of initialization, LR based sizing which is equivalent to solving the LDP in Equation (6), and a greedy post-pass. The Overall flow chart for RGS is shown in Figure 1.

During the initialization, RGS initializes all gates to the least leakage power option (lowest size and highest Vt), followed by a reverse topological scan to remove the load violations. Lagrange multiplier values for all the arcs are initialized to 1. Initial Lagrange multiplier values would depend upon the leakage power of a typical gate and its timing arc delay. Therefore, for a different library, one-time tuning of the initial Lagrange multiplier value might improve the convergence. To satisfy the KKT flow constraints, Lagrange multipliers are then updated using the projection technique [11]. Reimann et. al. [9] proposed a strategy to estimate Lagrange multipliers to speedup the convergence on a pre-optimized design. Since the ISPD 2012 gate

sizing contest provided unoptimized designs, their strategy is not needed for these benchmarks.

After initialization, LR based sizing begins. LR based sizing can be broadly divided into two stages: timing recovery, and power recovery. The timing recovery stage is focused on fixing most of the setup timing violations. It is composed of iterations between single gate sizing (SGS) which is a typical heuristic for solving LRS* (described later in this section), static timing analysis (STA) and Lagrange multiplier update. Each such iteration is referred to as an LR iteration. The Lagrange multipliers are updated to facilitate quick timing recovery. The Lagrange multiplier update strategy will be discussed in Section IV-A. When TNS is below a threshold, tsh_{TNS}, power recovery can begin.

The second stage of LR based sizing is power recovery. It is composed of five phases. While phases one, three, and five use SGS to solve LRS*; phases two and four use MGS. Like timing recovery, phase one of power recovery is coarse-grained optimization. It iterates between SGS, STA, CPS, and Lagrange multiplier update. In each phase of power recovery, CPS is optionally invoked if timing violations exceed the threshold tsh_{TNS} to keep timing violations under control so that power recovery can continue unimpeded. Lagrange multipliers are updated to facilitate quick power recovery. Phase one achieves the bulk of the power recovery and is usually the most runtime expensive phase. It is terminated as soon as the improvement in power is less than a threshold, tsh_{pow}, compared to the previous iteration.

Phases two through five perform finer-grained power recovery. Phase two performs a single iteration of MGS followed by STA, CPS, and multiplier update. Since MGS is time consuming we perform only one iteration. In addition to power recovery, MGS can potentially perturb the design out of a local minimum, creating opportunities for more power reduction. Therefore, we invoke SGS based power recovery iterations in phase three. Empirically, we determined that beyond two iterations in this phase, power recovery diminishes significantly. Unlike the SGS during coarse-grained optimization, i.e., during timing recovery and phase one of power recovery, SGS in phase three does not allow any gate to increase its power. To recover more power, we repeat phases two and three again in phases four and five, respectively.

After LR based sizing, small timing violations can still remain. Therefore, we invoke greedy timing recovery to eliminate all the violations. In greedy timing recovery, gates with a larger number of critical paths passing through them are processed first. Each gate is upsized, and timing is propagated through its entire fanout cone. If timing degrades, the sizing is undone and the next gate is processed. This is a common greedy heuristic to recover small timing violations.

There are two main differences between our flow and previous flows. Firstly, we have an explicit timing recovery stage followed by a power recovery stage. This provides finer grained control on runtime and solution quality. Previous works did not make such distinction. Secondly, instead of a greedy power recovery, we employ MGS which is parallelizable and is LR compliant.

Solving LRS(λ):* In the context of LR based gate sizing, single gate sizing (SGS) is a common heuristic for solving LRS* in Equation (4). SGS is briefly described as follows. For a given set of Lagrange multipliers, assuming no other gate can change its option, SGS near-optimally minimizes the objective in Equation (4) for each gate separately. Gates are visited in the forward topological order. For each gate, several options are evaluated to compute their contribution to the LRS cost, and the option that minimizes the cost is assigned to that gate. Changing the option of a gate can potentially affect the delays in the entire fanout cone of that gate. However, in the interest of runtime, and without much loss of accuracy, delay changes only in the local neighborhood of the gate are computed and applied towards the LRS cost computation. We use the multi-threaded version of SGS as proposed in [8].

Strategy from [7]

// q_x: required time at the output of gate x
for timing arc $i \rightarrow j$ **do**
  if $a_j \geq q_j$ **then**
    $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(1 + \frac{a_j - q_j}{T}\right)^{1/k}$
  else
    $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(1 + \frac{q_j - a_j}{T}\right)^{-k}$
Projection to satisfy Equation (5).

(a)

Strategy from [8]

// WPD: Worst path delay
$cexp = 1$
$T' = r \times T$
if $WPD > T'$ **then**
  $cexp = cexp \times \frac{WPD}{T}$
else
  $cexp = cexp \times \left(1 + k \times \frac{WPD - T'}{T'}\right)$
for timing arc $i \rightarrow j$ **do**
  $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(\frac{a_j - q_j}{T}\right)^{cexp}$
Projection to satisfy Equation (5).

(b)

Fig. 2: Lagrange multiplier update heuristics from previous works.

Flach et. al. [7] suggested that while solving the LRS, in addition to minimizing the LRS cost, its important to prevent timing degradation to ensure solution stability for faster convergence. They proposed an approximate way of ensuring this by restricting the local slack degradation. We call it *local slack check* and apply it in SGS. Through our experiments, we verify that the local slack check indeed facilitates faster timing convergence and thereby, reduce the LR iterations, especially during the timing recovery stage.

IV. LR Based Gate Sizing

In this section we discuss various components of LR based gate sizing. We present our proposed Lagrange multiplier update strategy and describe how it differs in timing recovery and power recovery. Then, we discuss our proposed MGS and CPS techniques.

A. Lagrange Multiplier Update

The Lagrange multiplier update strategy is very crucial in reducing the number of iterations for faster convergence. The general strategy is to increase (decrease) the multipliers across timing critical (non-critical) arcs. The key is how much to change. Most of the previous works use a non-linear expression to direct the optimization. The strategies presented in previous works [7], [8] are reproduced in Figures 2a and 2b, respectively. Both of these strategies use an exponent to adjust the multipliers and then project them to satisfy Equation (5). While the projection heuristic has not changed since it was proposed in [11], the main difference is how the exponent is tuned. Flach et. al. [7] do not provide much detail on how to tune the exponent. Moreover, they use different expressions for critical and non-critical arcs (refer Figure 2a) without any insight. On the other hand, Sharma et. al. [8] complicate the tuning of the exponent by introducing two other parameters, namely r and k (refer Figure 2b). Both of the strategies have slow convergence.

We propose a single expression for the Lagrange multiplier update (refer Algorithm 1) along with a much simpler strategy to tune the exponent. $D_{i\rightarrow j}$, in Algorithm 1, is the **worst path delay through the arc** $i \rightarrow j$, and K is the 'acceleration' factor. The ratio $D_{i\rightarrow j}/T$ indicates the timing criticality of the arc $i \rightarrow j$. The ratio is more than one for an arc with a timing violation, so the Lagrange multiplier for such an arc is increased. For a non-critical arc, the ratio is less than one, therefore its multiplier is decreased. The acceleration factor determines how quickly the Lagrange multipliers increase or decrease. Larger acceleration factors can speedup the convergence but can also cause the solution to get stuck in a worse local minimum.

Algorithm 1 Our proposed Lagrange multiplier update algorithm

for timing arc $i \rightarrow j$ **do**

$\lambda_{i \rightarrow j} = \lambda_{i \rightarrow j} \times \left(\frac{D_{i \rightarrow j}}{T}\right)^K$

Projection to satisfy Equation (5). Refer [11]

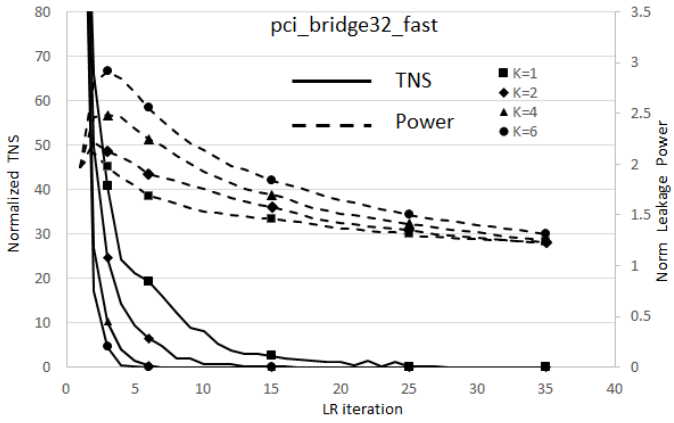

Fig. 3: TNS and power profiles for different values of K for critical arcs during the timing recovery stage are shown. TNS has been normalized with respect to T. As K increases, TNS reduces faster.

B. Timing Recovery

Since timing is a hard constraint that must be met, we first focus on fixing setup timing violations. To enable fast timing recovery, delay on the timing arcs with timing violations needs to be emphasized, so Lagrange multipliers for critical timing arcs need to increase faster. Therefore, acceleration factors of more than one for critical arcs during timing recovery improves the convergence of this phase. However, the larger the acceleration factor, the more the overshoot in the power. Figure 3 shows the TNS (solid lines) and power (dashed lines) profiles for different values of K for critical timing arcs. For non-critical timing arcs, we set $K = 1$. As K is increased, we observe faster convergence in TNS and larger overshoots in power. Note that, from $K = 4$ to $K = 6$, improvement in the TNS convergence is marginal but overshoot in the power is significant, as it may not always be recoverable due to the likelihood of the algorithm getting stuck in some bad local minimum. Also, due to the design being oversized for power, it is possible to simultaneously improve power and timing in the later iterations.

We use $K = 4$ for the critical arcs, and $K = 1$ for the non-critical arcs, during the timing recovery stage of LR based sizing. The small value of K for non-critical arcs means we gradually reduce their Lagrange multipliers and recover some power even during the timing recovery phase. In our case, timing recovery is said to converge when timing violations are less than a threshold tsh_{TNS}. With this setting, timing recovery on our experimental benchmark suite converges in four iterations on average.

C. Power Recovery

To quickly reduce power, Lagrange multipliers need to rapidly reduce along the non-critical timing arcs. Therefore, a large acceleration factor for the non-critical timing arcs is crucial. Figure 4 shows TNS and power profiles for different values of K applied to such arcs. For critical timing arcs, we set $K = 1$. We can observe that with larger K, power reduces more rapidly. We note that the gain diminishes from $K = 4$ to $K = 6$. On the other hand, TNS does not seem to be affected by K. This is because timing degradation is discouraged

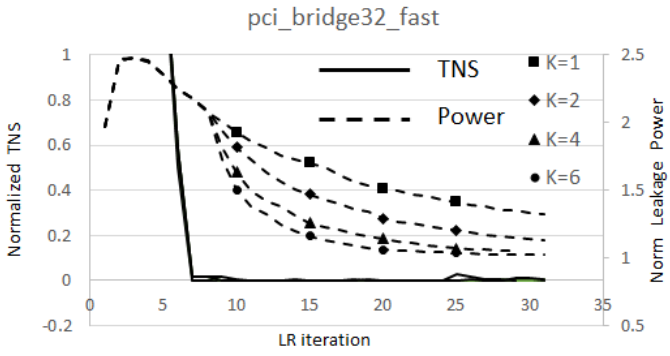

Fig. 4: TNS and power profiles for different values of K for non-critical arcs are shown. Around iteration 10, timing recovery ends and power recovery phase one begins. Note: TNS profiles for all K indistinguishably overlap. Therefore, markers are not used for them.

by the local slack check. Even if, on account of reduced Lagrange multipliers, LRS cost favors a smaller size (or larger Vt) for a gate, that size is not applied if local slack would degrade. Additionally, we invoke CPS for sizing critical paths whose timing violations exceed the threshold.

During all phases of power recovery, we set K to 1 and 6 for critical and non-critical timing arcs, respectively. If we use a larger K for non-critical arcs, we notice that the final power is 1 to 2% worse, because the design gets stuck in a worse local minimum. In order to terminate power recovery phase one, we propose an aggressive early exit strategy. We do not want to wait until phase one yields the best power that it can, because it can be very runtime inefficient in recovering power at later LR iterations. Therefore, we terminate this phase as soon as reduction in power is less than a threshold, tsh_{pow}, compared to the previous iteration, and invoke MGS which can size multiple gates simultaneously to recover power in larger chunks. We empirically verify that our aggressive early exit strategy can significantly reduce the number of iterations without compromising on the final power.

Algorithm 2 Multi-gate sizing (MGS)

```
1:  for each gate g ∈ S in forward topological order do
2:      status = false
3:      resize_g = g.downsize()
4:      resizes = {resize_g}
5:      if resize_g.valid() then
6:          for each fo ∈ g.fanout() do
7:              resize_fo = single_gate_sizing(fo)
8:              resizes = resizes ∪ {resize_fo}
9:          if change_in_power(resizes) < 0 then
10:             △negSlack = local_neg_slack_change(resizes)
11:             if △negSlack ≤ 0 then
12:                 status = true
13:     if status ≠ true then
14:         undo(resizes)
15:     STA every four topological levels
```

D. Multi-Gate Sizing (MGS)

As briefly discussed in Section III, a typical way of solving LRS is SGS which processes one gate at a time assuming that the other gates do not change their options. Such an approach restricts the solution space exploration, and increases the likelihood of the sizing solution

getting stuck in a local minimum. We propose MGS to alleviate this drawback by allowing multiple gates to simultaneously change their sizes. In favor of runtime, we do not change Vt and thereby, restrict the number of sizing combinations.

We'll briefly describe the MGS algorithm in reference to its pseudo code shown in Algorithm 2. MGS processes gates in forward topological order. Gate g is downsized and, if the new size of g is valid (lines 3-5) - in other words, no new load violations are created - then all the fanouts of g are sized in the same way as in SGS (lines 6-8). However unlike SGS, in favor of runtime and also since we do not anticipate large perturbations in the size of a gate at this stage, for each fanout only three options are evaluated: the current option, the option with the next bigger size, and the option with the next smaller size. The least LRS cost option is assigned to each fanout. New options for the gate g and all its fanouts are referred to as a set of *resizes*. Lines 9 and 11 describe the conditions to accept the resizes. The first condition is that the total power must decrease. If the first condition holds true, then the change in negative slack of the neighboring gates is computed (line 11). If the negative slack has not degraded then the resizes are accepted. If either condition fails, the resizes are undone (line 14). Since local slack degradation is only a rough indicator of the impact of the resizes on circuit timing, to prevent large timing violations from accumulating, we update timing after every four topological levels (line 15).

Since MGS can be more time consuming than SGS, we refrain from applying it during phase one of power recovery. We apply it only twice, during phases two and four of power recovery.

Algorithm 3 Critical Path Sizing (CPS)

1: $tsh = tsh_{TNS}/vend$ ▷ $vend$: #endpoints that violate timing
2: $S = \phi$
3: **for** each timing endpoint, end **do**
4: **if** end.slack $< -tsh$ **then**
5: $S = S \cup \{end\}$
6: Sort elements of S in the ascending order of slack
7: **for** each $end \in S$ **do**
8: $P = critical\_path(end)$
9: $minDeltaLM =$ some arbitrarily large value
10: **for** each arc $i \rightarrow j \in P$ **do**
11: $\triangle\lambda_{ij} = \lambda_{ij}\left[\left(\frac{D_{ij}}{T}\right)^K - 1\right]$
12: **if** $\triangle\lambda_{ij} < minDeltaLM$ **then**
13: $minDeltaLM = \triangle\lambda_{ij}$
14: **for** each arc $i \rightarrow j \in P$ **do** ▷ Update Lagrange multiplier along the path
15: $\lambda_{ij} = \lambda_{ij} + minDeltaLM$
16: **for** each gate $g \in P$ **do** ▷ LRS along the path
17: $single\_gate\_sizing(g)$
18: increamental_STA()

E. Critical Path Sizing (CPS)

The timing recovery stage can reduce the bulk of the timing violations in a few LR iterations. During various phases of power recovery, despite the local slack check, a few paths may become timing critical and TNS may exceed the threshold, tsh_{TNS}. In such cases, it is an overkill to run LR iterations to recover timing, because each LR iteration scans the entire design several times and is quite expensive. Moreover, LR iterations are not as effective in recovering finer-grained timing violations as they are during the coarse-grained optimization.

To reduce the delay of a timing critical path, usually either a gate along the path is upsized or its load is reduced. (Reducing the Vt at this stage of the algorithm is generally avoided due to the large increase in the leakage power.) Typically, to upsize a gate via SGS, Lagrange multipliers of the gate's timing arcs need to be large enough to justify trading power for reduced delay. However, at this stage in the algorithm when timing violations are not very big, for most of the timing arcs $i \rightarrow j$, $D_{i\rightarrow j}$ would be either close to T or significantly smaller than T. Consequently, it may require several LR iterations before the gate will be upsized by SGS. The delay of a timing critical gate can also be reduced by reducing its load, but that may induce timing violations on near-critical paths. Thus, critical and near-critical paths may compete with each other, thereby slowing down the convergence. Another strategy to reduce the small timing violations is to uniformly scale up all the Lagrange multipliers. This strategy is similar to applying the 'power weighting factor' of [6]. Although it can eliminate all the timing violations in one to two iterations, it tends to upsize even the non-critical gates, which causes unnecessary increase in the leakage power.

The CPS is designed to reduce the timing violations of the critical paths, while minimally affecting the total design power, and it is also very fast as it works on only a small sub-circuit. We describe the CPS algorithm with pseudo-code shown in Algorithm 3. In line 1, we compute a threshold, tsh, to identify timing critical endpoints. tsh is derived from tsh_{TNS} which is the allowed total timing violation during the LR based sizing stage. tsh is simply the average violation allowed per endpoint. Critical endpoints are then sorted by their slack (line 6). More timing critical endpoints, with more negative slack, are processed first. For each endpoint, its critical path, P, is computed (line 8). Then, lines 9 through 13 compute how much to increase the Lagrange multiplier along P. We use the Lagrange multiplier update expression from Algorithm 1 to compute the potential change in the Lagrange multiplier of each timing arc along the path (line 11), and track the minimum value, $minDeltaLM$. In order to emphasize the delay along the critical path, we would want substantial increase in the Lagrange multipliers. Therefore, we set $K = 10$ during the CPS. Then, the Lagrange multipliers of all the arcs along P are increased by $minDeltaLM$ (lines 14-15), followed by resizing all the gates along P. Lastly, the timing is incrementally updated in line 18 before processing the next primary output, so that the critical path of the next endpoint is computed based on the updated timing.

V. Experimental Results

We implemented our gate-sizer in C++. Experiments are performed on an 8-node cluster made up of two quad-core Intel(R) Xeon(R) E3-1240 v5 CPU @ 3.67GHz with an aggregate memory of 16GB. For multi-threading, OpenMP [12] is used. We use 8 threads. All the results reported in this work are averaged over 10 runs to minimize the bias due to non-determinism caused by multi-threading. The experimental set up including the benchmark suite is identical with the ISPD 2012 gate sizing contest. In our flow, we set $tsh_{TNS} = 0.1 \times T$ and $tsh_{pow} = 0.1\%$.

A. A Comparison With Previous Works

We use the algorithm proposed by Sharma et. al. [8] as our baseline. To the best of our knowledge, among all the published results so far, they have reported the best runtime on the ISPD 2012 contest benchmarks with 2.5% degradation in the average leakage power compared to the best quality published results [7]. For fair comparison against Sharma et. al., we executed their binary with 8 threads on our cluster, and we are using those results as the baseline in Table II. Since our cluster uses faster CPUs, the runtimes shown in column two of Table II are on average 18% smaller than the runtimes published by Sharma et. al. [8]. Powers in column five are on average

TABLE II: Comparison of overall runtime and power of RGS versus the baseline ([8]). Slow refers to the loose timing constraints, and fast refers to the tighter timing constraint. Benchmarks are listed in order of ascending number of combinational cells. DMA through netcard approximate combinational cell count is 23K, 30K, 102K, 148K, 213K, 540K and 861K, respectively.

Benchmark	Total runtime (min)			Leakage Power (W)		
	[8]	RGS	[8]/RGS	[8]	RGS	RGS/[8]
DMA_slow	0.11	0.07	1.47	0.135	0.135	0.997
pci_b32_slow	0.27	0.09	3.02	0.099	0.098	0.995
des_perf_slow	0.43	0.32	1.33	0.597	0.583	0.977
vga_lcd_slow	1.43	0.44	3.27	0.331	0.329	0.995
b19_slow	3.03	0.83	3.66	0.568	0.569	1.001
leon3mp_slow	3.91	2.52	1.55	1.335	1.335	1.000
netcard_slow	5.48	2.35	2.33	1.763	1.763	1.000
DMA_fast	0.26	0.08	3.05	0.251	0.245	0.979
pci_b32_fast	0.31	0.10	2.95	0.142	0.141	0.993
des_perf_fast	1.16	0.40	2.91	1.455	1.436	0.987
vga_lcd_fast	1.82	0.56	3.28	0.433	0.417	0.963
b19_fast	3.19	1.13	2.82	0.733	0.729	0.995
leon3mp_fast	4.88	3.13	1.56	1.443	1.449	1.004
netcard_fast	7.05	3.33	2.12	1.848	1.846	0.999
Average			**2.52**			**0.992**

TABLE III: Catalog of flows referred for different analysis.

Name	Flow
v1	RGS with the early exit policy adapted from the baseline [8]
v2	v1 with Lagrange multiplier update strategy adapted from the baseline
v3	v2 without the local slack check in the timing recovery phase
v4	v3 with CPS in the timing recovery phase

0.001% more than the published results. We also compare against Flach et. al. [7]. Their results are obtained from single threaded runs executed on 3.40GHz Intel(R) Core(TM) i7-3770 CPU.

Table II compares the total runtime and the power between RGS and the baseline. Both the flows yield timing violation free designs on all the benchmarks. On average, RGS is 2.52x faster than the baseline with 0.8% extra power savings. Compared with [7], which is single threaded, RGS is 19x faster and 1.5% worse in power. Authors believe that the 1.5% degradation in power can be attributed to the differences in tuning of the acceleration factor, K. We are able to optimize the biggest design in the suite, *netcard* which has around 861K combinational gates, in 3.33 min for the 'fast' and 2.35 min for the 'slow' timing constraints. The main contributor towards the speedup is significant reduction in LR iteration count. In Section V-B, we analyze various factors that contributed towards reducing the LR iteration count.

Average runtime breakdown of our flow is as follows: LR iterations dominate the runtime by accounting for 78% of the total runtime; followed by MGS (5%); greedy timing recovery (3%); and lastly, the CPS (1%). 14% of the total runtime is consumed in parsing the input verilog, spef, library files, and pre-processing. Note that although MGS can be parallelized like SGS, currently it is sequential.

Compared to the power reported by the baseline (fifth column in Table II), power reported by RGS after the coarse grained optimization, i.e., after phase one of power recovery, is the same as the baseline. Then phases two through five perform finer-grained power recovery and achieve a further 1.2% reduction in power, followed by the greedy timing recovery which increases power by 0.2%.

B. Factors Contributing to the Reduction in LR Iterations

Most of the speedup in RGS is due to the reduction in the LR iterations. In this section, we empirically analyze the contribution of different factors towards the reduction in the total LR iteration count. We identify three main factors, namely, (1) the aggressive early exit from the first phase of power recovery, (2) the proposed Lagrange multiplier update strategy, and (3) restricting the timing degradation in the timing recovery phase via local slack check. To evaluate the impact of each one of these factors individually we derive three different versions of gate sizers from RGS: v1, v2 and v3. They are summarized in Table III. In v1, we replace our aggressive early exit strategy by the early exit policy of the baseline. Baseline runs LR iterations as long as power is improving, whereas RGS terminates phase one of power recovery as soon as improvement in power is less than tsh_{pow}. v2 is built on top of v1 by replacing our proposed Lagrange multiplier update strategy with the corresponding strategy from the baseline. Lastly, v3 is built on top of v2 by disabling the local slack check in the timing recovery phase. On average, v3 has similar runtime as the baseline and it yields designs with 1% better power. So it is a good comparison point for our further analysis.

Table IV shows the LR iteration count during timing recovery stage, LR iteration count during phase one of power recovery stage, and power after phase one of power recovery for v3 and the RGS

TABLE IV: We compare the following for v3 and RGS: LR iteration count for the timing recovery stage (TR); LR iteration count for power recovery phase one (PR); and power after the power recovery phase one (LRpow). Power numbers are normalized with respect to the baseline (fifth column of Table II). At the bottom we also append the average results for v1 and v2.

Benchmark	v3			RGS		
	TR	PR	LRpow	TR	PR	LRpow
DMA_slow	10	42	0.999	3	14	1.007
pci_bridge32_slow	64	68	0.987	4	14	0.990
des_perf_slow	9	54	0.984	3	16	0.992
vga_lcd_slow	53	61	0.999	4	10	1.003
b19_slow	17	111	0.996	4	16	1.007
leon3mp_slow	8	34	1.001	3	10	1.003
netcard_slow	1	45	1.000	1	5	1.000
DMA_fast	72	57	0.982	5	15	1.000
pci_bridge32_fast	66	76	0.956	6	17	0.996
des_perf_fast	71	54	1.004	5	24	0.999
vga_lcd_fast	61	69	0.990	8	13	0.988
b19_fast	71	59	0.998	6	30	1.004
leon3mp_fast	10	37	1.003	3	18	1.012
netcard_fast	5	30	0.999	2	11	1.000
Average	**37**	**57**	**0.993**	**4**	**15**	**1.000**
v2 **Average**	**20**	**48**	**0.993**			
v1 **Average**	**4**	**44**	**1.002**			

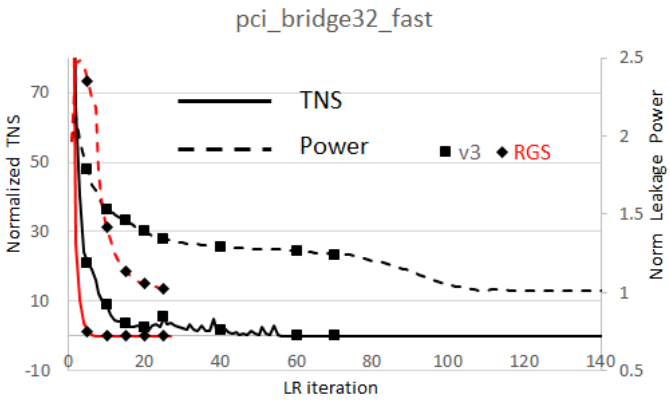

Fig. 5: Comparison of TNS and power profiles for v3 and RGS runs on the pci_bridge32_fast. LR iterations shown on the x-axis are from the timing recovery stage and phase one of power recovery. For RGS, by iteration 6, timing recovery ends and power recovery phase one begins. For v3, timing recovery extends until around iteration 60, followed by the power recovery phase one until iteration 140.

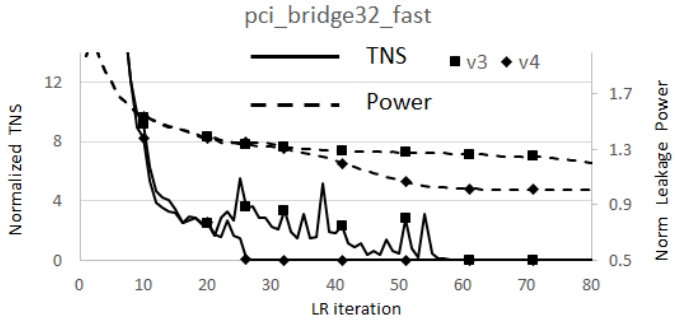

Fig. 6: TNS and power profiles for v3 and v4 runs on the pci_bridge32_fast are plotted. Only 80 LR iterations are shown. v4 invokes CPS around iteration 22 when it is still in its timing recovery stage, and TNS starts to degrade. Within 3 calls, CPS recovers the timing from the critical paths, and power recovery phase one begins at around iteration 26. On the other hand, v3 could not converge until 60 iterations.

flows. For the sake of comparison, average results for the same metrics are shown for v1 and v2 as well. Comparison of v1 and RGS shows that, by exiting early from phase one of power recovery, RGS could save 29 more iterations with only 0.2% higher power after phase one. There are slight fluctuations due to randomness on account of multi-threading. Compared to v1, v2 which uses the Lagrange multiplier update strategy of the baseline, takes 5x as many iterations to recover timing. v2's power after power recovery is around 1% lower than v1's power. Our Lagrange multiplier update uses large acceleration factors for quick timing and power recovery. That may cause the solution to get stuck in a worse local minimum, so power is slightly worse after phase one of power recovery. However, later power recovery phases involving MGS are able to recover it. Compared to v2, v3 which disables the local slack check during its timing recovery stage, spends on average 17 more iterations to converge the timing, and still yields the same power.

In summary, while our Lagrange multiplier update strategy is extremely effective in improving the convergence of timing recovery, the local slack check also helps to a smaller extent. The Lagrange multiplier update strategy also enables fast power recovery during phase one. Additionally, by exiting early from phase one, and relying on the later phases for finer-grained power recovery, the iteration count in phase one significantly reduced. Figure 5 compares TNS and power profiles for v3 and RGS runs on pci_bridge32_fast. As seen in the TNS profile of v3, timing convergence is initially quite slow due to poor Lagrange multiplier updates, and after iteration 22 due to lack of local slack check. As seen in the power profile of v3 between iterations 60 and 110, the power recovery in phase one is slow. This is due to the poor multiplier update strategy. After iteration 110, in an attempt to recover finer-grained power, phase one does not exit. Consequently, v3 takes about 140 iterations in total to complete timing recovery and phase one of power recovery, whereas RGS completes in only about 25 iterations. Overall, compared to v3, RGS reduces the total LR iterations of timing recovery and phase one of power recovery by 80%. The average power of RGS after phase one of power recovery is only 0.8% worse.

C. Impact of CPS on Timing Convergence

In order to evaluate the impact of CPS on timing convergence, we build a flow version, v4, by adding CPS in the timing recovery stage of v3. CPS has the ability to quickly recover timing from the critical paths. Therefore, once the TNS falls below a threshold ($2T$, in v4), and then if it degrades, we invoke CPS **instead of** SGS to converge the timing. Figure 6 shows pci_bridge32_fast as an example. In the figure, around iteration 20, TNS falls below $2T$. Then, TNS starts to degrade (increase) around iteration 22 at which point the CPS is invoked. CPS is able to recover the timing in just 3 calls. On the other hand, v3 is not able to converge the timing until iteration 60.

Overall results show that v4 on average, can cut down the number of LR iterations in timing recovery from 37 to 25. Moreover, each iteration of CPS is around 10x faster than an iteration of LRS. In fixing critical path timing violations, CPS causes only marginal increase in the total design power.

VI. Conclusion

In modern VLSI physical design flows, gate sizing is a time consuming optimization. LR-based gate sizers provide good quality results, but can take significant runtime due to the need to update timing after each iteration, as they can take many LR iterations to converge. In this work, we propose several techniques to enable rapid gate sizing by reducing the number of LR iterations. We utilize an elegant Lagrange multiplier update strategy to speed up the coarse-grained timing and power recovery. We also propose two LR-based techniques, MGS and CPS, for finer-grained power and timing refinement. These techniques allow the coarse-grained optimization to terminate early, further cutting down the number of iterations. Since LR iterations dominate the total runtime, our proposed gate sizer, RGS, is 3x faster than the previous fastest LR-based gate sizer, while still achieving state-of-the-art reduction in leakage power.

References

[1] C.-P. Chen *et. al.* Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):1014–1025, 1999.

[2] W. Ning. Strongly NP-hard discrete gate-sizing problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1045–1051, 1994.

[3] Y. Liu and J. Hu. A new algorithm for simultaneous gate sizing and threshold voltage assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(2):223–234, 2010.

[4] J. Hu *et al.* Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 233–239, 2012.

[5] L. Li *et al.* An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–232. IEEE, 2012.

[6] V. S. Livramento *et al.* A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4):40, 2014.

[7] G. Flach *et. al.* Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557, 2014.

[8] A. Sharma *et al.* Fast Lagrangian relaxation based gate sizing using multi-threading. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 426–433. IEEE, 2015.

[9] T. J. Reimann *et. al.* Cell selection for high-performance designs in an industrial design flow. In *ACM International Symposium on Physical Design*, pages 65–72. ACM, 2016.

[10] M.M. Ozdal *et al.* The ISPD-2012 discrete cell sizing contest and benchmark suite. In *ACM International Symposium on Physical Design*, pages 161–164. ACM, 2012.

[11] H. Tennakoon and C. Sechen. Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 395–402. ACM, 2002.

[12] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.