Gate Sizing and Device Technology Selection Algorithms for High-Performance Industrial Designs

Muhammet Mustafa Ozdal
Intel Corporation
Hillsboro, OR 97124
mustafa.ozdal@intel.com

Steven Burns
Intel Corporation
Hillsboro, OR 97124
steven.m.burns@intel.com

Jiang Hu
Texas A&M University
College Station, TX 77843
jianghu@ece.tamu.edu

ABSTRACT

It is becoming more and more important to design high performance designs with as low power as possible. In this paper, we study the gate sizing and device technology selection problem for today's industrial designs. We first outline the typical practical problems that make it difficult to use the traditional algorithms on high-performance industrial designs. Then, we propose a Lagrangian Relaxation (LR) based formulation that decouples timing analysis from optimization without resulting in loss of accuracy. We also propose a graph model that accurately captures discrete cell type characteristics based on library data. We model the relaxed Lagrangian subproblem as a discrete graph problem, and propose algorithms to solve it. In our experiments, we demonstrate the importance of using the signoff timing engine to guide the optimization. Compared to a state-of-the art industrial optimization flow, we show that our algorithms can obtain up to 38% leakage power reductions and better overall timing for real high-performance microprocessor blocks.

1. INTRODUCTION

As smartphones and tablets are becoming more popular, the traditionally low power devices are starting to tackle increasingly compute-intensive tasks such as high-definition video decoding, 3D rendering, etc. On the other hand, power consumption of high performance microprocessors has become an important metric due to increasing energy costs of server farms, battery-life considerations of notebook computers, and environmental concerns. So, it is becoming more and more important to design high performance circuits with as low power as possible. In this paper, we focus on the gate sizing and device technology selection problem for designs with high performance requirements.

The gate sizing problem has been studied extensively in the literature, and there have been many techniques proposed to solve this problem. However, the new challenges in modern technologies make it hard to apply many of these techniques without incurring significant overheads. One main problem with existing approaches is the over simplification of the timing models, which can lead to suboptimal decisions in sizing, and can require large guardbands to avoid timing violations. Especially for high performance designs, significant power savings are possible by taking into account accurate timing information during gate sizing.

We can list the main optimization challenges in modern industrial designs as follows:

- Discrete cell sizes: Continuous optimization techniques (such as Lagrangian relaxation [3]) inherently assume that cell sizes are continuous, and the delay-power trade-off can be modeled as a convex curve. However, industrial cell libraries contain limited number of cell sizes. Furthermore, different technology parameters introduce discrete cell families with very few elements in between (e.g. only a few levels of threshold voltages exist for a typical library). For example, for Intel's 32nm technology, it was shown that varying Idsat (corresponding to drive strength) between 1.4 and $1.6 mA/\mu m$ leads to about 10x difference in Ioff (corresponding to leakage power) for an NMOS transistor [7]. A typical cell library contains different cell types, each corresponding to a different set of technology parameters, corresponding to different power-performance tradeoffs. Note that these technology parameters are orthogonal to gate sizes, which are traditionally easier to model as continuous curves. However, in the presence of discrete technology parameters, the circuit optimization problem is no longer only the well-known gate sizing problem, but also a discrete cell type selection problem among the available device technology parameters. Using the traditional continuous optimization techniques for this purpose would require defining continuous models over different device technologies, which would not capture the accurate characteristics of the discrete cells in the library. Furthermore, these continuous sizes would need to be snapped to discrete sizes at the end, which can be very sparse especially for different technology types.
- Cell timing models: Modern cell libraries utilize table lookup models to characterize cell delays. Furthermore, the different technology parameters used for different cells (such as gate length, doping, etc.) lead to different threshold voltages, and different tradeoffs between drive strengths and leakage powers. It is hard to represent all different cell types with a simple formula such as $delay = driver\_resistance \times load$. Even higher-order convex models are not accurate enough for modern cell libraries, because, in reality, delay is not a convex function of cell size and output load (due to transistor folding in the layout, etc.) [4]. As an example, consider Figure 1 for a timing arc of a cell from an industrial 32nm library. Here, normalized delay is plotted with respect to normalized size and output load. Note that the ratio of output load to size is constant in this graph. The simple delay models would assume constant delay when the cell size and the output load both change by the same amount. However, in reality, this is not the case as demonstrated in this figure. Furthermore, different technology parameters lead to different cell families, and hence different curves, two of which are shown in this figure.
- Complex timing constraints: There can be various timing constraints imposed on high performance designs such as timing overrides, multi-cycle paths, transparent paths, multiple clock events, false paths, etc. An algorithm that relies on simple timing models to compute slacks will not capture these constraints accurately.
- Interconnect timing models: Elmore model can be reasonable for early estimation, but is not accurate enough for final optimization. As the interconnect delays are becoming more and more dominant with device scaling, slack computation using simple interconnect delay models are not reliable anymore.
- Slew effects: The cell delay tables are typically defined with respect to input slews and output loads. The input slews can have significant impact on the gate delays. For example, downsizing a gate can lead to worse slew at its output pin, which also worsens the delays through other gates after it. So, slew changes and their impact on other cells should be considered during optimization.

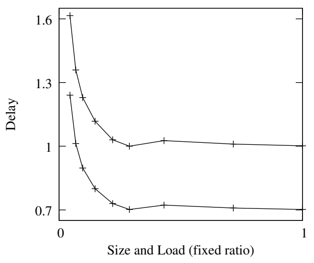

Figure 1: Delay as a function of cell size and output load for two cell families from a 32nm industrial library. In the x axis, the size and output load values are changed such that $load/size = k$, where k is kept constant. All values are normalized.

- Many near-critical paths: In a high performance design, a large portion of the cells may be on critical or near-critical paths. The algorithms that optimize only critical paths iteratively can have convergence problems. A near-critical path can become critical repeatedly depending on the size changes. Ideally, an optimization algorithm should consider all paths for high performance designs, not only the most critical ones.
- Large design sizes: Some blocks in modern designs can easily contain millions of cells, and the optimization algorithm should be scalable enough to handle such large designs. So, compute-intensive algorithms may be impractical for modern design. Also, the algorithms that iteratively upsize a few gates and do incremental timing updates can be too expensive, especially if an accurate timing engine is utilized.

As listed above, a gate sizing algorithm applicable to modern high performance designs should have an accurate view of timing. Modeling all the complicated timing constraints in the optimizer is not practical given all the sophistications in the modern timing engines. Ideally, the optimizer should be guided by the slacks computed by the signoff timing engine for accurate timing information. However, calling the signoff timer after every cell size change is not practical due to long runtimes even in the incremental mode.

In this paper, we have the following contributions: 1) We propose a Lagrangian Relaxation (LR) formulation (Section 4) that decouples timing analysis from the optimization engine without resulting in loss of accuracy. Existing LR based sizing algorithms incorporate static timing analysis modeling into the optimization formulation [3]. Although this is adequate for simple timing models, the complexity will significantly increase in the presence of realistic timing constraints (multiple clock domains, multi-cycle overrides, false paths, transparent latches, etc.). Our proposed formulation allows using the slack values computed by the signoff engine, rather than having to model all timing constraints in the optimizer. 2) We propose a graph model (Section 5) that captures the delay costs of discrete cells accurately based on the tables in the cell library. For multi-fanout nets, each edge cost can be modeled independently, while still capturing the fact that the load of one branch can affect the delay of another branch. The slew effects on delay are also taken into account as defined in the library. We show that minimizing the subnode and edge costs in this graph corresponds to optimizing the Lagrangian-relaxed subproblem (LRS). 3) We propose a delta-delay cost metric that alleviates the suboptimalities due to double counting in DAG optimization by combining fanin and fanout costs in the subnode costs. 4) We propose a dynamic programming (DP) algorithm based on critical tree extraction to solve the LRS optimization problem for discrete cells (Section 6). 5) In our experimental study (Section 7), we first use real microprocessor blocks to show the practical benefits of using the signoff timer to guide the LR iterations. Then, we compare our results with a state-of-the-art industrial tool, and show that up to 38% power savings are possible, while reducing the timing violations.

2. RELATED WORK

Many previous works on gate sizing are based on continuous optimization, such as linear programming [1, 4], convex programming [5, 13], iterative analytical solving [3, 15, 11], and fractional network flow [12, 16]. The prevalence of cell library-based designs requires rounding the continuous solutions to discrete options in a library. The rounding is not trivial at all, as it can easily result in infeasible solutions. As such, the rounding can be as complicated as a full-fledged combinatorial optimization like dynamic programming [8] and branch-and-bound [11]. Moreover, the continuous optimizations are often restricted to simple delay models such as piece-wise linear model [1] and RC switch gate model [3, 15, 12, 16].

Discrete gate sizing is also directly tackled by combinatorial optimizations. Not surprisingly, the discrete sizing problem is proved to be NP-hard [9]. An early work on discrete gate sizing [2] is an exhaustive search, which is hardly affordable even on medium-sized circuits. For simultaneous gate sizing and threshold voltage assignment, sensitivity-based heuristics are reported in [17, 14]. The greedy nature of these heuristics often leads to ad hoc solutions. A recent breakthrough is [10], which allows a dynamic programming-like systematic solution search despite the notorious fanout reconvergence problem. Although combinatorial approaches are capable of addressing the aforementioned practical issues, such a capability has not been demonstrated in these previous works. A multi-direction search method is developed in [6]. Similar to simulated annealing, this is a flexible framework that can accommodate many practical concerns such as accurate delay models and slew rate. However, its computation runtime is difficult to scale with modern chip designs.

Instead of directly solving the gate sizing problem, Lagrangian relaxation (LR) changes the problem to a subproblem, which is easier to solve, and a companion dual problem. An elegant and perhaps the most famous LR-based work is [3]. With the Elmore delay model, it provides the optimal continuous solutions. Later on, speedup techniques for LR-based approaches are proposed in [15]. LR is also integrated with convex programming [5], network flow [16] and dynamic programming-based algorithm [10]. However, all these works assume simple timing models and more practical issues are ignored.

3. PRELIMINARIES

Let $\mathcal{D}$ be the given design that contains a set of standard cells $\mathcal{C}$, a set of pins $\mathcal{P}$ on these cells, and a set of nets $\mathcal{N}$ that define the connectivity over these pins. For each standard cell $c \in \mathcal{C}$, a set of cell types S_c is defined, where S_c can contain cell types with different sizes and device technology parameters[1]. Let $power(s)$ denote the leakage power of cell type $s \in S_c$. For simplicity, we will focus on leakage power optimization, but it is straightforward to extend our models to consider dynamic power optimization if the activity factors are specified.

Furthermore, assume that an arbitrary set of timing constraints are enforced on design $\mathcal{D}$. Let $slack(p)$ be defined as the timing slack of pin $p \in \mathcal{P}$, which is computed by a timing engine. Obviously, negative slacks indicate timing violations. Let TNS denote the absolute value of the total negative slack of all the primary outputs in $\mathcal{D}$. Although our implementation can handle other constraints such as max transition time, max capacitance, etc., we will not present them here due to page limitations.

Based on these definitions, we can define the basic problem as follows: Given a design $\mathcal{D}$ with timing constraints, determine the cell type $s \in S_c$ for each cell $c \in \mathcal{C}$ such that the following objective function is minimized: $\alpha \sum_{c \in \mathcal{C}} power(c) + TNS$. Here α is a constant that determines the relative importance of power minimization with respect to timing violations. Eventually, the TNS value needs to

[1]In the rest of the paper, *cell type* will refer to both the gate size and the set of technology parameters used.

be reduced to 0 before tapeout. However, during the design process, some timing violations can be allowed, assuming some changes will be made in the design to fix these violations later. Note that this objective function is chosen for practical reasons as opposed to enforcing TNS to be zero, so that our techniques are also applicable to designs that are not converged yet.

In a typical standard cell library today, cell delays and slews are defined using delay tables (DT) and slew tables (ST). The timing arcs are defined from input pins of the cell (rising and falling) to the output pin[2] (rising and falling). For each timing arc through the cell, a table of the form $DT[input\_slew, output\_load]$ exists to define the timing arc delay for the given input slew and output load. A similar slew table $ST[input\_slew, output\_load]$ exists for output slews. If a given slew or load value does not exactly match the entries in the table, linear interpolation is performed in-between the nearest table entries. So, in this paper, we assume that two functions $delay(input\_slew, output\_load)$ and $output\_slew(input\_slew, output\_load)$ are available in the cell library for each timing arc of each cell type. Note that although delay and slew have some linear dependence to load within an interval (due to linear interpolation between table entries), in general, the behavior can be nonlinear if the load and input slew values vary a lot.

4. LAGRANGIAN RELAXATION BASED OPTIMIZATION

Existing work on Lagrangian Relaxation (LR) based sizing [3] incorporates static timing analysis modeling directly into the formulation of the optimization problem. Although this is adequate for simple timing models, the complexity will increase significantly in the presence of complex timing constraints (multiple clock domains, multi-cyle overrides, false paths, transparent latches, etc.). In this section, we propose an LR formulation that decouples slack computation from optimization. This formulation allows the optimizer to use the slack values computed by the signoff timer directly, instead of having to model them explicitly. This way the complexity of the timing analysis is encapsulated in the signoff timing engine (including timing constraints, interconnect timing models, etc.). Also the signoff timer needs to be called only once per iteration, after all the cell sizes and types are chosen to optimize the Lagrangian relaxed subproblem (LRS), as will be described in Sections 5 and 6. This allows our approach to scale well to handle large industrial designs, and makes it affordable to process all cells in the design instead of only the most critical ones.

The problem of gate sizing is traditionally formulated as minimize either area or power subject to the constraints that 1) the design should meet static timing constraints, 2) the design should use only legal gate sizes, and 3) the design should meet additional constraints such as max transition, max capacitance, etc. For simplicity, we will only derive relaxations associated with the timing constraints. Although our implementation can handle other constraints, they will not be presented here due to page limitations.

Let us formulate the objective function as a trade-off between power and total negative slack:

$$\alpha power + \sum_{po} \max(0, a_{po} - r_{po}), \tag{1}$$

where a_{po} and r_{po} are the arrival and required times of transition po, and po ranges over all transitions (up and down) of the timing endpoints (primary outputs or sequential inputs) in the design. Next, we define the dummy variables $\hat{m}_{po}$ (negative margin) for representing the term inside the summation in the cost function above:

$$\begin{aligned} 0 &\le \hat{m}_{po} \\ a_{po} - r_{po} = m_{po} &\le \hat{m}_{po} \\ a_u + d_{u\to v} &\le a_v \end{aligned} \tag{2}$$

where $d_{u\to v}$ is the delay of the timing arc from u to v. Putting the constraints into a form conducive to relaxation into the objective function and introducing a Lagrangian multiplier for each constraint, we add these constraints times their respective multipliers to the original objective function, obtaining the Lagrangian-Relaxed Subproblem (LRS):

$$\alpha power + \sum_{po} \hat{m}_{po} + \sum_{po} \mu'_{po}(-\hat{m}_{po}) + \sum_{po} \mu_{po}(a_{po} - r_{po} - \hat{m}_{po}) + \sum_{u\to v} \mu_{u\to v}(a_u + d_{u\to v} - a_v) \tag{3}$$

Note that Eqn (3) contains the terms margins and the arrival times. These are the quantities that are hard to compute during sizing, and we want to rely on the signoff timer to compute these quantities. Fortunately, we can eliminate these quantities by maintaining the flow constraints among the introduced multipliers. Here, for example, the terms $\hat{m}_{po}$ cancel out if $\mu'_{po} + \mu_{po} = 1$:

$$\alpha power + \sum_{po} \mu_{po}(a_{po} - r_{po}) + \sum_{u\to v} \mu_{u\to v}(a_u + d_{u\to v} - a_v) \tag{4}$$

Here, each timing arc is associated with a Lagrangian multiplier (LM) $\mu_{u\to v}$. Based on Kuhn-Tucker conditions, it is derived in [3] that the sum of multipliers on incoming arcs of a node must be equal to the sum of multipliers on its outgoing arcs. If the multipliers are treated like flow, this conclusion is equivalent to the flow conservation in network flow model.

By introducing further constraints conserving the multiplier flow, we get the following functional form as our LRS formula:

$$\alpha power + \sum_{u\to v} \mu_{u\to v} d_{u\to v} + \sum_{po} \mu_{po}(-r_{po}) + \sum_{pi} \mu_{pi} a_{pi} \tag{5}$$

Observe that the variables in this LRS formula are only the cell power and the timing arc delay through each cell, whereas r_{po} and a_{pi} are fixed. This equation can be optimized (Sections 5 and 6) using the data from the cell library only, without having to compute the arrival times and slacks explicitly.

It is known that for any fixed set of LMs, the optimal result to the LRS problem will be no greater than the optimal result to the original problem [3]. So, the Lagrangian dual problem is to maximize the minimum value obtained for the LRS problem by updating LMs accordingly. If arrival times were available, then Eqn (4) would be the appropriate form for updating the multipliers. However, in the presence of complex timing constraints, we want to use the slack values computed by the signoff timer directly. Using the definitions $m_v = a_v - r_v$ and $m_{u\to v} = a_u + d_{u\to v} - r_v$, we can rewrite (4) as:

$$\alpha power + \sum_{po} \mu_{po}(a_{po} - r_{po}) + \sum_{u\to v} \mu_{u\to v}(m_{u\to v} - m_v) \tag{6}$$

Now, we can base the multiplier update on two types of margin values: the margin value across a timing arc, $m_{u\to v}$, and the margin value at a node, m_v. These values are readily calculated by most static timing engines, and can be used directly in our LR framework.

In summary, we perform a number of LR iterations. In each iteration, we size all the cells to minimize the LRS formula in Eqn (5)

[2]For simplicity, let us assume a single output pin for each cell. It is straightforward to generalize our models for multi-output cells.

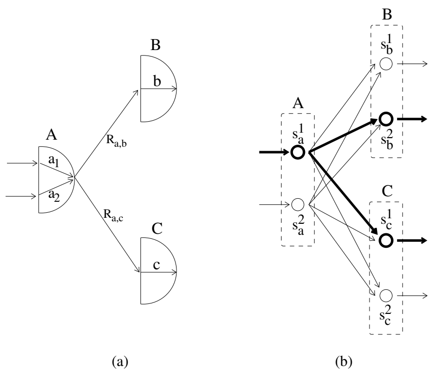

Figure 2: (a) A sample cell type selection problem with cells A, B, and C. (b) The corresponding graph model, where a subnode exists for each available cell type.

based on the current set of LMs. Then, we use the signoff timer to compute the new slack values. Based on the computed slack values, we use a subgradient-based algorithm to update the LMs to maximize Eqn (6). Further details are omitted due to page limitations.

5. GRAPH MODEL

In this section, we propose a graph model that captures the Lagrangian relaxed subproblem (LRS) formulation in Eqn (5). Here, our objective is to represent the delay costs as accurately as possible using the delay functions in the cell library.

In the rest of the section, we will make use of the example shown in Figure 2(a). In this example, we have 3 cells A, B, and C, where the input pins of B and C are both connected to the output pin of A. Let a_1 and a_2 denote the timing arcs through cell A. Similarly, let b and c denote the single timing arcs through cells B and C. Finally, let $R_{a,b}$ and $R_{a,c}$ denote the interconnect resistances of the net from A to B and from A to C, respectively. For ease of presentation, we assume that the routes from A to B and from A to C do not have common segments. However, it is straightforward to extend our models to handle that case.

In our graph model, we define a subnode for each possible cell type of each cell. Let s_i^j denote the j^{th} cell type corresponding to cell i. In Figure 2(b), we have subnodes $\{s_a^1, s_a^2\}$, $\{s_b^1, s_b^2\}$, $\{s_c^1, s_c^2\}$ corresponding to cells A, B, and C, respectively. To keep the example small, we assume that there are only 2 cell types available for each cell, although in reality, one can expect many more available cell types, corresponding to different sizes and technology parameters.

Once the graph is defined this way, the problem is how to define the node and edge weights such that the original LRS objective function is captured accurately. One difficulty here is how to define edge costs of multi-fanout nets independent of each other. In the example of Figure 2, the delay of timing arcs a_1 and a_2 depend on both types of cells B and C, but we want to model the edge costs independent of each other. Furthermore, the delays through timing arcs b and c depend on the slew at the inputs of B and C, which in turn depend on both the type of A and its output load.

Ideally, when exactly one subnode is chosen for each cell, then the sum of the selected subnode weights and the weight of the edges between them should directly correspond to our LRS cost metric defined in Section 4. For example, in Figure 2, if cell types s_a^1, s_b^2, and s_c^1 are chosen, then the LRS cost should be equal to the total weights of the highlighted subnodes and edges in Figure 2(b).

In the rest of this section, we discuss how to model the subnode and edge weights in our graph model. For simplicity of presentation, we will ignore interconnect delays and slew dependency in the next two subsections. Then, in Sections 5.3 and 5.4, we describe how to incorporate interconnect delays and slew effects into our cost metrics.

5.1 Simple Delay Cost Model

We first show a rather straightforward way to model edge weights based on absolute delay costs in this subsection. Then, we discuss potential problems with this approach, and will propose a better model in the next subsection to avoid these issues.

The delay cost of an edge (e.g. the edge from s_a^1 to s_b^1 in Figure 2) can be modeled independent of other edges (e.g. the edge from s_a^1 to s_c^1) if delay is assumed to be linear with respect to the output load, i.e. if $delay(cap_B + cap_C) = delay(cap_B) + delay(cap_C)$. With this assumption, we can model our graph as follows:

- The weight of subnode s_i^j is set as: $weight(s_i^j) = \alpha.power(s_i^j)$, where α is the constant in the original LRS formulation, and $power(s_i^j)$ is the leakage power of the j^{th} type of cell i.
- The weight of edge from s_i^j to s_m^n is set as $weight(s_i^j \rightarrow s_m^n) = \sum_k \mu_k.delay_k(cap(s_m^n))$, where μ_k and $delay_k$ are the Lagrangian multiplier and the delay of the k^{th} timing arc of cell i, respectively. Here, delay is represented as a function of $cap(s_m^n)$, which is the input capacitance of n^{th} type of cell m. The summation is over all timing arcs through cell i, because the delay through each arc depends on the input cap of s_m^n.

Lemma 5.1. *Under the assumption that delay is a linear function of load, and ignoring interconnect delays and slew effects, the cost metric above models the LRS cost exactly. In other words, the LRS cost in Section 4 will be minimized if we choose exactly one subnode for each cell such that the total cost of all selected subnodes and edges between them is minimized.*

PROOF. The variable portion of the LRS cost from Eqn (5) can be rewritten as: $\sum_i(\alpha.power(s_i^j)+\sum_k(\mu_k.delay_k(\sum_m(cap(s_m^n)))))$, where i is over all cells; k is over all timing arcs through cell i; and m is over all cells connected to the output of cell i. Assume cell type s_i^j is selected for cell i and the type s_m^n is selected for each cell m connected to the output of i. By definition, subnode cost of s_i^j is $\alpha.power(s_i^j)$, and the sum of edge costs connected to s_i^j will be $\sum_m \sum_k \mu_k.delay_k(cap(s_m^n))$. Due to linearity assumption, this expression can be rewritten as $\sum_k \mu_k.delay_k(\sum_m cap(s_m^n))$. Hence the lemma follows. □

There are two problems with this type of formulation. First, the linear delay model is inaccurate for modern cell libraries, which define delays in look-up tables. The second problem is with the difficulty in the optimization algorithms. Although the algorithms will be described in detail in Section 6, we will briefly mention the difficulties here. If the proposed graph was a tree structure (e.g. no nets have more than one fanout, and there are no re-convergent paths), then a dynamic programming (DP) formulation would solve the LRS optimization problem optimally. However, as also mentioned in [10], reconvergent paths in DAGs make optimization more difficult, leading to heuristic algorithms that don't guarantee optimality. We claim that a graph model that uses absolute delay values as its edge costs make DAG optimization even more difficult, and the heuristics more prone to suboptimalities.

As an example, consider the DAG optimization problem in Figure 3, where there are 4 cells A, B, C (with two subnodes each), and D (with a single subnode). The edge costs are as shown in the figure. For simplicity of the example, let us assume that the subnode costs are zero. The optimization objective is to choose exactly one subnode for each cell such that sum of edge costs between the selected subnodes is minimum. This figure also shows how a DP-based heuristic would operate on this problem. A typical heuristic would process the nodes in topological order, and compute the minimum accumulated cost at each subnode. For example, for the first subnode of cell B, the accumulated cost is the minimum of $1 + 8$ or $10 + 1$, depending on

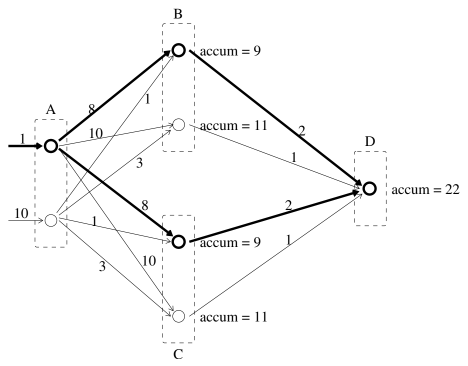

Figure 3: A sample optimization problem with edge weights as marked. The solution of a commonly used heuristic is highlighted, where the solution cost is equal to the total weights of the edges highlighted.

the previous subnode selected. Note that there are edges from cells B and C converging at the input of D. This can happen if output pins of B and C are connected to different input pins of D. In this case, the minimum accumulated cost of D is chosen to be the sum of min costs at its input pins, i.e. min(9+2, 11+1) + min(9+2, 11+1) = 22. Backtracking from D leads to the solution highlighted in Figure 3.

Many existing heuristics focus on the "historical inconsistency" problems in DAG optimization [10]. For example, in Figure 3, backtracking from D could end up choosing different subnodes of cell A. Algorithms such as [10] try to avoid this problem through some heuristics. However, in Figure 3, the solution obtained is *consistent*, i.e. the paths from D through B and through C both end at the same subnode of A. Despite this, the solution obtained is suboptimal. The total weight of the subgraph highlighted in Figure 3 is 1+8+8+2+2 = 21. However, if the second subnode of A was chosen instead of the first, then the total weight would be 10+1+1+2+2=16. This example shows that we can obtain suboptimal results using common heuristics even when the solution does not have the "historical inconsistency" problem. We will show in the next subsection that we can make DAG optimization easier by using a different graph model.

5.2 Delta Delay Based Modeling

In this section, we make use of the iterative nature of LR optimization. Especially in the later iterations of LR, we can expect the cell sizes start to converge, and the size changes in every iteration to be small with respect to the previous iteration. Also, the accuracy of the delay model becomes more important as the iterations start to converge. Based on this, we define our graph model using the sizes from the previous iteration as reference, and modeling the delta delay values in the current iteration. This approach both improves the accuracy of the delay model for multi fanout nets, and makes DAG optimization easier.

Let us consider a multi-fanout cell i with a particular type s_i^j, and let us define the reference delay as the delay value when all cells at the output of i have their reference cell types (i.e. same types as they had in the previous iteration). We can compute the reference delay through timing arc k of this cell using the delay function defined in the lookup table of s_i^j:

$$delay\_ref_k(s_i^j) = delay_k\Big(\sum_{t \in fanout(i)} cap(s_t^{ref})\Big) \qquad (7)$$

Here, s_t^{ref} is the reference size of cell t at the fanout of cell i. As before, $cap(s_t^{ref})$ refers to the input capacitance of s_t^{ref} connected to the output of cell i.

Now, assume that the cell type of one of the fanout cells (e.g. cell m) is changed, leading to a change in its input capacitance by $\Delta cap(s_m^n)$, i.e. $\Delta cap(s_m^n) = cap(s_m^n) - cap(s_m^{ref})$, where s_m^n is the

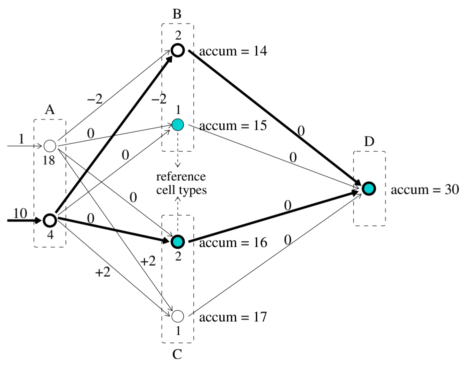

Figure 4: The graph model based on delta delay model. The reference cell types (from the previous iteration) are shown as shaded circles. The result of a DP-based heuristic is highlighted.

new cell type of m. We can use first order approximation to compute the new delay value through timing arc k of driver cell i as:

$$delay(s_i^j) = delay\_ref_k(s_i^j) + \Delta cap(m) \cdot \left.\frac{\partial T_k}{\partial cap}\right|_{ref} \qquad (8)$$

Here, $\frac{\partial T_k}{\partial cap}|_{ref}$ is the derivative of delay through arc k with respect to output load, and it is computed at the reference load. In other words, we linearize the delay function at around point (ref_cap, ref_delay). Since the cell delay models typically use linear interpolation for delay between lookup table entries, we expect the accuracy of this model to be reasonable if $\Delta cap(m)$ is not too large. As mentioned above, as the LR iterations converge, we expect the accuracy of this model to improve more and more.

Based on this delay model, we can re-define the costs in our graph model as follows.

- The weight of a subnode s_i^j is computed as:

$$weight(s_i^j) = power(s_i^j) + \sum_{k \in arcs(s_i^j)} \mu_k delay\_ref_k(s_i^j) \qquad (9)$$

- The weight of an edge from s_i^j to s_m^n is computed as:

$$weight(s_i^j \rightarrow s_m^n) = \Delta cap(s_m^n) \cdot \sum_{k \in arcs(s_i^j)} \mu_k \left.\frac{\partial T_k}{\partial cap}\right|_{ref} \qquad (10)$$

Compared to the cost model of Section 5.1, the main difference here is that we do not assume delay is linear in capacitance, but we do linearization around the point (cap_ref, delay_ref), which is computed based on the delay tables of the cell from the library. Also, we include the reference delay cost in the subnode cost, and let the edge costs reflect delay differences only. This model also simplifies the DAG optimization, as described in the following example.

Let us consider the example of Figure 3 again, and assume that the reference sizes of cells B and C are as marked in Figure 4. The reference size of A is irrelevant for this example. Using Eqns (9) and (10), we obtain the values shown in Figure 4.

For example, the cost of the first subnode of A is set to 18, which is the sum of old edge weights (in Figure 3) from this subnode to the reference subnodes of B and C (i.e. 10 and 8). Similarly, the edge weights from the second subnode of A to the reference sizes of B and C were 3 and 1 (in Figure 3); hence the subnode cost is 4. By definition, the edge weights to the reference subnodes are always zero, because the subnode weights are defined with respect to these reference cell types. The other edge costs are defined based on the differences in their cost values with respect to the reference costs. For example, in Figure 3, the edge cost from the first subnode of A to the first subnode of B was 8, and to the second subnode of B was 10, which is the reference cost now. So, the corresponding edge costs in Figure 4 are -2 and 0, respectively, both with respect to the reference cost 10.

If the same DP-based heuristic (as in Section 5.1) is used for this model, the minimum accumulated cost found at node D becomes 30, and backtracking leads to the selection of the highlighted edges in Figure 4. If we compare the highlighted subgraphs of Figure 3 and Figure 4, we can see that the total cost is less in the latter figure.

Intuitively, by collecting the reference costs of all edges originating from a subnode in the actual subnode cost, we can capture the real cost of the subnode in a better way in the DAG optimization. For example, in Figure 3, the first subnode of A has high fanout cost but small fanin cost, and the second subnode has the opposite. If a typical DP-based heuristic is used, the computed accumulated cost at node D double-counts the fanin cost of A, but does not double-count its fanout costs. Since the first subnode of A has lower fanin cost, it is chosen over the second subnode in Figure 3, leading to a suboptimal solution. However, in Figure 4, the reference edge costs are collected in the subnode costs, and the subnode costs reflect the actual costs more accurately. For example, one can see that the first subnode of A is more expensive overall, because most of its fanout costs are captured in its subnode cost. The edge costs are smaller, and they only reflect the delta delays with respect to the reference size. This makes optimization easier in general. This is also true for the critical tree extraction based algorithm we propose in Section 6.

Furthermore, this model allows us to capture the delay values more accurately from the delay lookup tables for the reference load. The only potential inaccuracies are in the delta delay values, which are expected to be small as the LR iterations start to converge.

5.3 Modeling Interconnect Costs

In the previous two subsections, we have ignored the interconnect effects for simplicity of presentation. In this section, we show how to model interconnect effects in our graph model.

Basically, two changes are needed to the model described in Section 5.2. First, the reference delay cost through timing arc k of cell type s_i^j needs to be computed considering the interconnect capacitance at its output. For this, we can rewrite Eqn (7) as:

$$delay\_ref_k(s_i^j) = delay_k(interconnect\_cap + \sum_{t \in fanout(i)} cap(s_t^{ref})) \tag{11}$$

Again, $delay_k()$ is the delay function defined in the library for timing arc k. We simply add the effective capacitance of interconnect at the output pin of s_i^j. In our current implementation, we use a lumped capacitance model, ignoring effects such as resistive shielding. As mentioned before, we rely on the actual timing engine to perform accurate slack computations, and those slack values guide our LR optimization framework. However, by using a simplified lumped capacitance model here, we take into account how the delay of the driver cell is impacted due to interconnect.

Our second change in the model is to consider how input capacitances of different cell types affect the interconnect delays. We can capture this effect by adding the interconnect delay costs to the subnode cost of the corresponding cell type. Let us define the interconnect cost corresponding to cell s_i^j as follows:

$$interconnect\_cost(s_i^j) = \sum_{p \in input\_pins(s_i^j)} \mu_p \cdot R_p \cdot cap(p) \tag{12}$$

Here, μ_p is the Lagrangian multiplier at the input pin p of s_i^j, R_p is the effective resistance of the interconnect connected to pin p, and $cap(p)$ is the input pin capacitance. Obviously, if we choose a cell type with a larger input capacitance, the cost of the preceding interconnect will increase. Based on this, we can add $interconnect\_cost(s_i^j)$ to the subnode cost of s_i^j.

5.4 Modeling Slew Dependencies

It is important to model slew dependencies, because the cell delays depend not only on cell types, but also the slew rates at their input pins. As mentioned before, in a typical library, the delay and output slew tables are defined as a function of output loads and input slew values. In the previous subsections, we have assumed cell delays as functions of output loads only. In this section, we show how to incorporate slew dependencies as well.

If the slew value at each input pin was given, we could simply use these values during cell delay computations. However, the input slew rates depend on the cell types of the preceding cells. In the example of Figure 2, the slew rate at the input pin of cell B depends on all the cell types before B. So, the delay through arc b, and hence the subnode and edge costs of B all depend on the predecessor cells. This necessitates the traversal of our graph in forward topological order.

Starting from the primary inputs, we can compute the slew rate at the output of a subnode using the output slew tables from the library. Then, we can propagate these slew values to successor nodes. In other words, as long as we process the nodes in topological order, we can compute the input slews at each subnode, and can compute the subnode and edge costs defined before using the library delay tables. Note that this means our graph model is not static, because the edge weights depend on the path preceding them.

Other than this, we also need to take into account the slew effects at multi-fanout nets. Consider the example of Figure 2 again. If we upsize cell C, then the load at the output of A increases, and the slew rate at the output of A and at the inputs of B and C all get worse. This increases the delay of not only arc c (through cell C), but also arc b (through cell B). Although our dynamic graph model described above captures the impact on arc c, the impact on arc b is not captured. The reason is that cell C is not before B in topological order, and it can be processed before or after cell B. So, we need to capture this effect for multi-fanout nets explicitly.

We have observed in practice that the change in slew rates have the highest impact on the first level cells only, and significantly less impact on the second or third level successors. For the example of Figure 2, the slew change at the input pin of B (due to load change of C) will affect the delay through cell B and the output slew of B. This change in output slew of B can also affect other cells driven by B, but not as much as the arc through B. So, for simplicity, we model the slew impact on first-level cells only.

For simplicity of presentation, we will derive our formulas using the example in Figure 2, and then generalize them at the end of this subsection. Let us now rewrite the delay through timing arc b for cell type s_b^j as:

$$delay_b(s_b^j) = delay_b(input\_slew(b), output\_cap(b)) \tag{13}$$

First-order approximation around reference delay (i.e. when the size of C is the same as in the previous iteration) leads to:

$$delay_b(s_b^j) = delay\_ref(s_b^j) + \Delta input\_slew_b \cdot \left.\frac{\partial T_b}{\partial slw}\right|_{ref} \tag{14}$$

Here, $\frac{\partial T_b}{\partial slw}|_{ref}$ denotes the delay sensitivity of timing arc b with respect to input slew at around the reference input slew and output load of arc b, which can be obtained directly from the cell delay table. This sensitivity value is multiplied by $\Delta input\_slew_b$, which is the input slew change due to size change of cell C. We can approximate this as:

$$\Delta input\_slew_b = \Delta cap_C \cdot \max\left\{ \left.\frac{\partial slw_{a1}}{\partial cap}\right|_{ref}, \left.\frac{\partial slw_{a2}}{\partial cap}\right|_{ref} \right\} \tag{15}$$

Here, Δcap_C is the change in the input capacitance of cell C with respect to the reference type of C; $\frac{\partial slw_{a1}}{\partial cap}|_{ref}$ and $\frac{\partial slw_{a2}}{\partial cap}|_{ref}$ are the output slew sensitivity values of arcs $a1$ and $a2$ with respect to output load. These sensitivity values are computed at around the reference output loads based on the sizes from the previous iteration. Here, we take the maximum slew sensitivity to consider the worst degradation in the slew rate at the output of A.

Substituting Eqn (15) into Eqn (14), we can obtain:

$$delay_b(s_b^j) = delay\_ref(s_b^j) + \Delta cap_C \cdot \max\left\{ \left.\frac{\partial slw_{a1}}{\partial cap}\right|_{ref}, \left.\frac{\partial slw_{a2}}{\partial cap}\right|_{ref} \right\} \cdot \left.\frac{\partial T_b}{\partial slw}\right|_{ref} \quad (16)$$

Intuitively, the input capacitance change of C affects the delay through timing arc b by the delta term in Eqn (16), so this delta delay value needs to be accounted for during selection of the cell type of C. Note that this term depends on the input capacitance of cell C and the output slew sensitivity of cell A to output load. Hence, it depends on both sizes of A and C. So, we add the following cost term to each edge weight from s_a^j to s_c^n:

$$slew\_impact(s_a^j \rightarrow s_c^n) = \mu_b \cdot \Delta cap(s_c^n) \cdot \max\left\{ \left.\frac{\partial slw_{a1}}{\partial cap}\right|_{ref}, \left.\frac{\partial slw_{a2}}{\partial cap}\right|_{ref} \right\} \cdot \left.\frac{\partial T_b}{\partial slw}\right|_{ref} \quad (17)$$

We can generalize this formula for any edge from subnode s_i^j to s_m^n as follows:

$$slew\_impact(s_i^j \rightarrow s_m^n) = \sum_{t \in fanout(i) \backslash m} \left(\mu_t \cdot \Delta cap(s_m^n) \cdot \max_{k \in arcs(s_i^j)} \left\{ \frac{\partial slw_k}{\partial cap} \right\} \cdot \frac{\partial T_t}{\partial slw} \right) \quad (18)$$

This way we can capture the slew impact of cell m on the other cells (if any) connected to the output of cell i.

6. ALGORITHMS

In the previous section, we have modeled the LRS optimization as a graph $\mathcal{G}$, taking into account cell delay tables, interconnect costs, and slew effects. In this section, we describe our algorithms to solve the following problem: Choose a subnode for each cell node in $\mathcal{G}$ such that the total cost of the selected subnodes and edges between them is minimized. For this purpose, we use a dynamic programming (DP) based algorithm. As mentioned before, $\mathcal{G}$ can have reconvergent paths, so we first need to extract the critical trees from $\mathcal{G}$ (Section 6.1), and optimize each tree independently (Section 6.2). Note that this approach is different than [10], where heuristics are used to expand beyond multi-fanout cells in DP. Such an approach has the disadvantage that the delay values computed can be inaccurate even in the late iterations of LR, because the sizes estimated during DP expansion can be incorrect due to double counting of costs after multi-fanout expansion. In contrast, our method is expected to get more and more accurate as the LR iterations converge, because our estimation error at multi-fanout nets is bounded by the change in the current iteration only.

6.1 Critical Tree Extraction

For a forward traversal of $\mathcal{G}$, if all nets had single fanout, then $\mathcal{G}$ would be a tree, and we could solve the LRS optimization problem using DP directly. For a cell c with multiple fanouts, we have a choice between: 1) making c the root of the current subtree (e.g. cell H of Figure 5), or 2) pre-determining the cell types of all receivers of c except one (e.g. cell G of Figure 5). We will show how to make this choice based on the relative criticalities of the receivers.

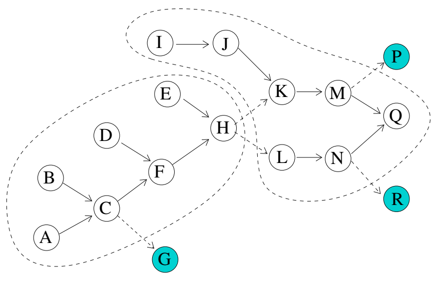

Figure 5: A sample graph model for LRS optimization. The non-critical cells with pre-determined types are shown with shaded circles. The two trees extracted are shown with dashed boundaries.

As mentioned before, as the LR iterations start to converge, the cell sizes do not change dramatically. So, we can pre-determine the type of a non-critical cell based on the sizes of its neighbors in the previous iteration. We expect this approximation to be good enough for non-critical cells. In Figure 5, if the size of cell G is predetermined before DP optimization, then the driver cell C can assume G has fixed input capacitance during DP optimization.

Note that the larger the extracted trees, the more accurate our LRS optimization will be. At the boundaries between different trees, we will rely on approximations for the input capacitances of the receivers. In the example of Figure 5, the output load of H will be computed using the reference sizes of K and L from the previous iteration. The LR framework is supposed to compensate for such inaccuracies by choosing the multipliers accordingly [3].

To solve this problem, we first define a criticality metric for each cell based on the sum of Lagrangian multipliers (LMs) of its timing arcs. It is known that the LMs are expected to be large for critical paths. Then, we process each multi-fanout net, and compare the criticality weights of the receivers based on a parameter we define as *relative criticality threshold* β, which is determined empirically. If the criticality weight of one receiver is at least β times larger than the other receivers of the same net, then we *fix* all the receivers of this net except the most critical one. This way, the tree can continue to be expanded through the most critical receiver.

After this step, we process the cells in topological order. When we encounter a net with multiple receivers that have not been fixed before, we extract a subtree rooted at that net. We continue this process until all the primary outputs are processed. Figure 5 shows an example where 2 subtrees are extracted: {A, B, C, D, E, F, H} and {I, J, K, L, M, N, Q} after 3 non-critical cells were pre-determined: {G}, {P}, {R}.

6.2 Dynamic Programming Based Optimization

We process the trees extracted in Section 6.1 in topological order, and use dynamic programming (DP) based algorithms to choose the best subnode for each node such that the sum of edge and subnode costs is minimized.

For the neighboring cells outside the tree, we consider a single cell type only: For the receiver cells connected to the root node, the reference sizes are considered from the previous iteration (e.g. cells K and L connected to H in Figure 5). For all others, the cell types have already been determined in this iteration, so accurate computations are possible.

In our DP algorithm, we keep track of the accumulated cost up to each subnode while processing nodes in topological order. Once we reach the root node, we choose the best cell type and backtrack from there. Further details are omitted due to page limitations.

7. EXPERIMENTAL RESULTS

In our experiments, we first show the importance of using an accurate timing engine rather than approximate timing models, as com-

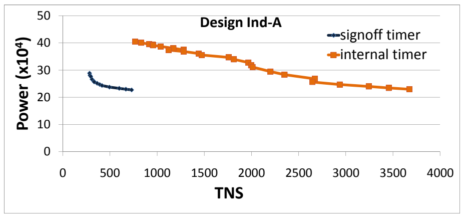

Figure 6: Experimental results on block ind-A, comparing two different timing engines guiding LR optimization.

monly used by many existing works. For this, we make use of two different timing engines: 1) the signoff timing engine, 2) an in-house static timing analyzer. Note that the in-house implementation is significantly more sophisticated than the simple timing models used by many of the academic gate sizing works. It uses the cell delay and slew tables from the cell library, it can handle transparent latches, multiple clock domains, and multiple clock events per domain. However, it cannot handle false paths and multi-cycle timing overrides, and its interconnect model is based on Elmore.

In the first set of our experiments, we use a small microprocessor block ind-A (27K cells) with high performance constraints, and we use one of the two timing engines described above to guide our LR-based optimization. After the LR iterations end, we pick the iteration that has the minimum value for our objective metric. Then, we use the signoff timer to evaluate the final results. Figure 6 shows the results of our experiments where we vary the α value in our objective metric (1) to obtain results with different power performance tradeoffs. Here, the leakage power is plotted against the total negative slack (TNS). Note that all values in this figure are normalized with respect to the power and delay of the smallest inverter size in the library (due to IP reasons). As can be seen in Figure 6, using the signoff timer during optimization leads to significantly better results, because the paths with timing violations are targeted exactly during LR iterations. If the signoff timer is not used, the timing violations are significantly larger for the same power levels. Also, reducing timing violations (through guardbanding) requires significantly more extra power, as can be seen from this figure.

In the second set of experiments, we evaluate the effectiveness of our model on a set of high-performance industrial microprocessor blocks. Here, we make comparisons between 1) our proposed algorithm, and 2) a state-of-the-art industrial optimization flow. For (2), we obtained blocks from microprocessor designers, who have run the industrial flow on these blocks over a period of time. Hence, the runtimes of the industrial flow cannot be reported. Note that for all of the designs, the detailed parasitics information is available, and utilized by the timing engine during optimization. Our experiments are performed on Linux systems with dual 3GHz quad-core Intel Xeon CPU and 32GB memory.

Our results are listed in Table 1, where all values are normalized with respect to the smallest inverter power and delay values. Observe that our algorithm leads to 29% average reduction in leakage power (38% reduction in the best case), while reducing the total negative slacks (TNS) significantly. Our runtimes are reasonable for such large blocks, and about 85% of the runtime is spent by the signoff timer during LR iterations. Our LRS optimization takes about 15% of the runtime on average.

8. CONCLUSIONS

We have proposed models and algorithms for the gate sizing and device technology selection problem. We have focused on the problems encountered in high performance industrial designs, which are typically ignored by the existing gate sizing works in the literature. Our experiments show significant improvements with respect to a state-of-the-art industrial flow.

Table 1: Experimental results on real microprocessor blocks

Design	#sizeable cells	INDUSTRIAL FLOW Power (norm)	TNS (norm)	WNS (norm)	OUR ALGORITHM Power (norm)	TNS (norm)	WNS (norm)	cpu (h)	Power impr
ind-B	81K	391K	5.8	0.8	294K	0.0	0.0	2.8	25%
ind-C	60K	271K	4.2	1.7	194K	0.8	0.8	1.6	28%
ind-D	68K	209K	85.8	11.7	191K	27.5	11.7	1.1	9%
ind-E	111K	420K	96.7	5.0	401K	30.8	5.0	1.2	5%
ind-F	361K	1961K	436.7	3.3	1214K	8.3	1.7	13.2	38%
Avg		650K	125.8	4.5	459K	13.5	3.8	4.0	29%

9. ACKNOWLEDGEMENTS

The authors would like to thank Bun Tse (Intel Corp.) for help with the experimental study.

10. REFERENCES

[1] M. R. C. M. Berkelaar and J. A. G. Jess. Gate sizing in MOS digital circuits with linear programming. In *Proc. of DATE*, pages 217–221, 1990.

[2] P. K. Chan. Algorithms for library-specific sizing of combinational logic. In *Proc. of DAC*, pages 353–356, 1990.

[3] C. P. Chen, C. C.-N. Chu, and D. F. Wong. Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Trans. on Computer-Aided Design*, 18(7):1014–1025, July 1999.

[4] D. Chinnery and K. Keutzer. Linear programming for sizing, vth and vdd assignment. In *Proc. of ISLPED*, pages 149–154, 2005.

[5] H. Chou, Y.-H. Wang, and C. C.-P. Chen. Fast and effective gate sizing with multiple-Vt assignment using generalized Lagrangian relaxation. In *Proc. of ASPDAC*, pages 381–386, 2005.

[6] O. Coudert. Gate sizing for constrained delay/power/area optimization. *IEEE Trans. on VLSI Systems*, 5(4):465–472, 1997.

[7] S. N. et.al. A 32nm logic tech. featuring 2nd-generation high-k + metal-gate transistors, enhanced channel strain and 0.171m2 sram cell size in a 291mb array. In *Proc. of IEDM*, 2008.

[8] S. Hu, M. Ketkar, and J. Hu. Gate sizing for cell-library-based designs. *IEEE Trans. on Computer-Aided Design*, 28(6):818–825, June 2009.

[9] W.-N. Li. Strongly NP-hard discrete gate sizing problems. In *Proc. of ICCD*, pages 468–471, 1993.

[10] Y. Liu and J. Hu. A new algorithm for simultaneous gate sizing and threshold voltage assignment. In *Proc. of ISPD*, 2009.

[11] M. Rahman, H. Tennakoon, and C. Sechen. Power reduction via near-optimal library-based cell-size selection. In *Proc. of DATE*, 2011.

[12] H. Ren and S. Dutt. A network-flow based cell sizing algorithm. In *Workshop Notes, Int'l Workshop on Logic Synthesis*, 2008.

[13] S. Roy, W. Chen, C. C.-P. Chen, and Y. H. Hu. Numerically convex forms and their application in gate sizing. *IEEE Trans. on Computer-Aided Design*, 26(9):1637–1647, Sept. 2007.

[14] A. Srivastava, D. Sylvester, and D. Blaauw. Power minimization using simultaneous gate sizing, dual-Vdd and dual-Vth assignment. In *Proc. of DAC*, pages 783–787, 2004.

[15] H. Tennakoon and C. Sechen. Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *Proc. of ICCAD*, pages 395–402, 2002.

[16] J. Wang, D. Das, and H. Zhou. Gate sizing by Lagrangian relaxation revisited. *IEEE Trans. on Computer-Aided Design*, 28(7):1071–1084, July 2009.

[17] L. Wei, K. Roy, and C.-K. Koh. Power minimization by simultaneous dual-Vth assignment and gate sizing. In *Proc. of CICC*, pages 413–416, 2000.