Fast Lagrangian Relaxation Based Gate Sizing using Multi-Threading

Ankur Sharma[†], David Chinnery[‡], Sarvesh Bhardwaj[‡], Chris Chu[†]
Iowa State University Computer Engineering[†], Mentor Graphics[‡]
Emails: {ankur,cnchu}@iastate.edu[†], {david_chinnery, sarvesh_bhardwaj}@mentor.com[‡]

***Abstract*—We propose techniques to achieve very fast multi-threaded gate-sizing and threshold-voltage swap for leakage power minimization. We focus on multi-threading Lagrangian Relaxation (LR) based gate sizing which has shown both better power savings and better runtime compared to other gate sizing approaches. Our techniques, mutual exclusion edge assignment and directed graph-based netlist traversal, maximize thread execution efficiency to take full advantage of the inherent parallelism when solving the LR subproblem, without compromising the leakage power savings.**

With 8 threads, our multi-threading techniques achieve on average 5.23x speedup versus our single-threaded (sequential) implementation. This compares well to the maximum achievable speedup of 5.93x by Amdahl's law due to 5% of the execution not being parallelizable. To highlight the problems with load imbalance and poor scheduling, we also propose a simpler approach based on clustering and topological level-by-level netlist traversal, which can achieve only 3.55x speedup.

We also propose three simple yet effective enhancements - fast optimal local resizing, early exit policy, and fast greedy timing recovery - to speed up single-threaded LR-based gate-sizing without degrading the leakage power. We test our gate sizer using the ISPD 2012 gate sizing contest benchmarks and guidelines. Compared to other researchers' state-of-the-art LR-based gate sizer, our approach is 1.03x (with 1-thread) and 5.40x (with 8-threads) faster and only 2.2% worse in leakage power.

I. Introduction

Power consumption of integrated circuits has increased substantially with much larger circuits integrated on a single chip with shrinking technology dimensions. Circuit performance is now limited by power due to higher power densities and device limits. Reducing power consumption is a high priority for circuit designers to allow higher performance, to reduce cooling and packaging costs, and to extend battery life in mobile devices.

In VLSI physical design, gate sizing is one of the most frequently used circuit optimizations. Each logic gate has several possible implementations in terms of size and threshold voltage (Vth) of cell alternatives in a standard cell library. Different implementations (cells) trade off area or power for delay. The task of a gate-sizer is to choose a suitable cell for every gate to minimize power while meeting the design timing constraints. With increasing design sizes of a million gates and larger, optimization tools must be very fast while not sacrificing the quality of results.

The discrete gate-sizing problem is NP-hard [1]. Also, cell delays do not vary in a convex manner with area or power, because for example internal capacitances vary with cell layout due to multiple transistor fingers or fins to implement greater drive strengths. This makes the gate-sizing problem very difficult to solve optimally. Researchers have applied various techniques such as greedy iterative sensitivity-based heuristics [2], [3], linear programming [4], [5], convex programming [6] [7], Lagrangian Relaxation (LR) [8]–[15], network flow [16] [17], dynamic programming (DP) [18]–[20], and logical effort [21] [22].

Two major drawbacks of most of these works are inaccurate delay models and the assumption that the gate sizes are continuous. In academia, researchers often assume simplified delay models like the Elmore delay model [8], or an input-slew independent delay-model [4]. Alternatively, they approximate with a convex delay model [7], such as a posynomial function [6]. Such delay models can be quite inaccurate versus SPICE models or library lookup tables, as reported in [5], [21]. Whereas, industry has mostly been using the table-lookup based non-linear delay models that do not have nice properties like convexity, but are fairly accurate. Continuous sizing followed by mapping to the discrete space may not be able to satisfy the timing constraints especially if the discrete gate sizes are of coarse granularity in size [18]. Moreover, in most of the previous works, the benchmark suites for evaluation either contained only small designs or were proprietary, making a fair comparison difficult.

Some of these concerns were addressed with the ISPD 2012 [23] and ISPD 2013 Discrete Gate Sizing Contests [24], which provided a common platform for fairly comparing different gate-sizing approaches. The objective is to minimize leakage power while meeting the timing constraints. Since then, several works have utilized the contest framework to compare their gate sizers [3], [13]–[15], [22]. We observe that some of the fastest approaches with competitive solution quality[1] use LR as the main technique [13]–[15]. LR achieves excellent results as it provides a global optimization avoiding local minima, and Karush-Kuhn-Tucker (KKT) optimality conditions [8] greatly prune the search space.

Significant further speedups in LR based gate-sizers can be achieved by smartly multi-threading different blocks of the LR framework. Liu et al. use a GPU to accelerate their DP-based gate-sizer [20]. DP-based gate sizing is optimal for tree topologies but it is known to exhibit suboptimal behavior due to path reconvergence. Li et al. very briefly talk about multi-threading their iterative LR framework [13]. Their multi-threading strategy in each iteration is to simultaneously resize the gates that are either in the same topological level or three levels apart. (We shall refer to topological levels as levels in the rest of the paper.) While three level separation might avoid inaccuracies in slew and capacitance computation, on the downside, only one-third of the gates are resized in each iteration which results in slow convergence and/or higher leakage. We propose techniques that consider all the gates in each iteration without compromising accuracy, and yet achieve high thread utilization. Compared to Li et al. [13], on ISPD2012 contest benchmarks, our sizer is 7.8x faster and, on average, saves 12% more leakage power.

In this paper, we focus on developing techniques that enable efficient multi-threading to realize a very fast gate sizer. With 8 threads, our multi-threading techniques achieve on average 5.23x speedup versus our sequential implementation, without degrading the leakage power. This compares well to the maximum achievable speedup of 5.93x by Amdahl's law due to only 5% of the execution not being parallelizable. To highlight the problems with load imbalance and poor scheduling, we also propose a simpler set of approaches based on clustering and topological level-by-level netlist traversal. Our major contributions are summarized below:

- We propose mutual-exclusion edge (MEE) assignment to avoid inaccuracies in capacitance computation. In contrast with clus-

[1]Up until now, [15] presents the best results for both leakage and runtime for all of the ISPD 2012 gate-sizing benchmarks.

tering, MEE greatly improves the load balancing.
- We use directed acyclic graph (DAG) netlist traversal (DNT) to systematically propagate the slew. In contrast with leveling, DNT does not require threads to synchronize at each level.
- We provide three enhancements to the sequential approach: a fast optimal local resizing (Fast-OLR) and an early exit policy to speedup the parallelized sequential runtime, and a fast greedy timing recovery (Fast-GTR) to reduce the non-parallelized sequential runtime. Due to Fast-GTR we are able to reduce the unparallelized sequential runtime to a mere 5%.

This paper is organized as follows. In Section II, we formulate the problem. Section III presents our sequential LR approach. There we describe the overall flow of our gate-sizer. In Section IV, we present MEE assignment and DAG-based netlist traversal. Section V details our enhancements to reduce sequential runtime. We discuss the experimental results in Section VI and conclude in Section VII.

II. Problem Formulation

Following the ISPD 2012 contest guidelines, we assume that 1) only combinational gates can be resized, whereas sequential gates have a fixed size, and 2) a lumped capacitance model is used for modeling interconnect capacitance. Consequently, all nodes of a net (one driver and one or more sinks) share the same timing information (slew, arrival and required times).

Throughout this work, T is the clock period; node i is the driver node of the net i; $i \rightarrow j$ denotes the timing arc from node i to node j; a_i and q_i are the actual and required arrival times (AAT and RAT) at node i, respectively; and $d_{i\rightarrow j}$ is the delay of the timing arc $i \rightarrow j$.

The objective is to choose suitable cells for every gate so that the total leakage power (sum of leakage powers of individual gates) is minimized under three types of timing constraints: 1) worst path delay $\leq T$; 2) maximum output capacitance $\leq maxcap$; and 3) maximum output slew $\leq maxslew$. This minimization is over all the available cells for every gate in the library. In our approach, we guarantee to satisfy the second and the third constraints throughout, so we do not formalize them mathematically below. We do model rise and fall timing constraints separately, but we have omitted them here for clear presentation. The original problem is formulated as follows:

$$\begin{aligned} \underset{cell,a}{\text{minimize}} \quad & \sum_{gates} leakage \\ \text{subject to} \quad & a_i + d_{i\rightarrow j} \leq a_j, \text{ for each timing arc } i \rightarrow j \\ & a_k \leq T \text{ for each primary output } k \end{aligned} \tag{1}$$

We apply the Lagrangian Relaxation technique to Equation (1) to include all the constraints into the objective. To avoid constraint violations, a positive penalty term is introduced per constraint, called the Lagrangian multiplier (LM, λ). This gives the Lagrangian Relaxation Subproblem (LRS):

$$\begin{aligned} \underset{cell,a}{\text{minimize}} \quad & \sum_{gates} leakage + \sum_{i\rightarrow j} \lambda_{i\rightarrow j} \left(a_i + d_{i\rightarrow j} - a_j\right) \\ & + \sum_{k} \lambda_k \left(a_k - T\right) \end{aligned} \tag{2}$$

By applying KKT conditions [8], the LRS can be simplified to

$$\underset{cell}{\text{minimize}} \quad \sum_{gates} leakage + \sum_{i\rightarrow j} \lambda_{i\rightarrow j} d_{i\rightarrow j} \tag{3}$$

The Lagrangian Dual Problem (LDP) is shown in (4).

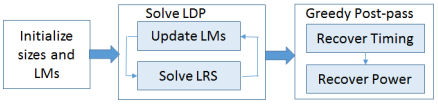

Figure 1: Gate-sizing algorithm flowchart

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \quad & \left(\underset{cell}{\text{minimize}} \sum_{gates} leakage + \sum_{i\rightarrow j} \lambda_{i\rightarrow j} d_{i\rightarrow j} \right) \\ \text{subject to} \quad & \sum_{u \in fanin(i)} \lambda_{u\rightarrow i} = \sum_{v \in fanout(i)} \lambda_{i\rightarrow v}, \text{ for each node } i \end{aligned} \tag{4}$$

The term $\sum \lambda_{i\rightarrow j} d_{i\rightarrow j}$ is referred to as the *lambda-delay* [15].

III. Sequential Approach

In this section we present a brief overview of our sequential gate-sizer. As shown in Figure 1, the algorithm has three stages: the initialization; the LDP solver; and the final greedy stage.

A. Initialization

In the initialization stage, each gate is assigned its minimum leakage library cell. However, that might violate all three types of timing constraints. Gates are minimally upsized to satisfy the maximum load capacitance and maximum slew constraints. This is done by traversing the netlist in reverse topological order and upsizing the gates that have capacitance violations; followed by a forward topological traversal to upsize the gates to satisfy the slew constraints, where necessary. The LMs are all initialized to 1.

B. LDP solver

This is an iterative stage to approximately solve (4). In each iteration, the LM update and the LRS solver alternate to update the LMs for the given cells and update the cells for the given LMs, respectively.

1) LM update: The LDP solver begins with the static timing analysis (STA) and updating of the LMs according to the criticality of the corresponding timing arc. We developed our own static timer and verified its accuracy against Synopsys PrimeTime, as per the contest guidelines. For updating the LMs, a common strategy is to increase the LMs in proportion to their timing criticality [9]. We use an exponential factor *cexp* to emphasize the LMs (see the pseudo code in Figure 2). Although the use of *cexp* was originally presented in [15], details were not provided therein on how to update it, except for a few hints. In our experience, the update method for *cexp* is crucial to the final solution quality, so we explicitly show it in the pseudo code. As long as the design violates the relaxed delay target ($r * T$), we increase *cexp*. By the time the design satisfies the relaxed delay target, *cexp* is usually very large because we started out with the minimum power solution instead of a minimum delay solution. Therefore, in order to accelerate the power recovery, we rapidly reduce *cexp* by using the factor k. KKT projection ensures that LMs satisfy the KKT conditions [9]. Both STA and LM update are highly parallelizable sub-blocks (discussed in Section IV), though they contribute only a small fraction of the total runtime.

2) LRS solver: The LRS solver approximately solves (3) by optimal local resizing (OLR) of one gate at a time, assuming all the other gates are fixed. Gates are traversed in forward topological order from the primary inputs (PIs) to the primary outputs (POs), i.e., OLR of a gate begins after all its fanin gates have been processed [14]. Such an order can be precomputed and stored. To optimally resize a

```
1   Algorithm: LM-update
2   // WPD: Worst path delay
3   cexp = 1
4   if (WPD > r * T)      // r = 1.01
5       cexp *= WPD / T
6   else
7       cexp *= (1 + k * (WPD - r*T) / (r*T))   // k = 10
8   endif
9   foreach timing arc i → j
10      λ_{i→j} *= (1 + (a_j - q_j) / T)^cexp
11  endfor
12  KKT projection
```

Figure 2: Pseudo code for LM update

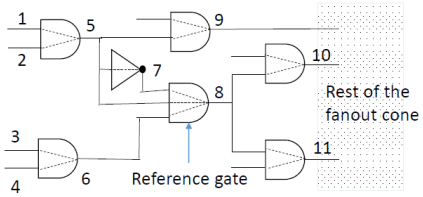

Figure 3: Local arcs include fanin-arcs: 1-5, 2-5, 5-7, 3-6, 4-6; gate-arcs: 7-8, 5-8, 6-8; fanout-arcs: 8-10, 8-11; side-arcs:5-9. Drain-nets: 10, 11; side-nets: 9.

gate, the *lambda-delay-cost* is computed for all the valid cells[2] of that gate and the cell with the lowest cost (*lambda-delay-cost* + *leakage*) is chosen. The *lambda-delay-cost* approximates *lambda-delay*. To exactly compute *lambda-delay*, an incremental timing analysis is needed which becomes prohibitively expensive when done for several cells of every gate. To limit the runtime overhead when analyzing alternate cells, timing of only the local arcs is recomputed [13]. To improve the accuracy, Flach et al. [15] proposed global delay/slew sensitivity functions to approximate the change in *lambda-delay* for the rest of the fanout cone (the second term in eq.(5)). Therefore, the *lambda-delay-cost* for a cell c can be computed as follows:

$$\textit{lambda-delay-cost}(c) = \sum_{i \to j \in local-arcs(c)} \lambda_{i \to j} d_{i \to j} + \sum_{n \in drain-nets(c) \cup side-nets(c)} \triangle D_n^{\lambda} \quad (5)$$

where $\triangle D_n^{\lambda}$ is the lambda-delay change in the net n due to the change in output slew of the gate driving the net [15]. Local-arcs constitute fanin-arcs, gate-arcs, side-arcs and fanout-arcs, as shown in Figure 3 along with the drain-nets and the side-nets. The LRS solver is the most expensive sub-block. It also offers great parallelism since multiple gates can be processed simultaneously.

C. Greedy post-pass

The last stage combines two greedy heuristics: greedy timing recovery (GTR), and greedy power recovery (GPR). The least power solution obtained from LDP is the starting point for the greedy post-pass. If the starting solution satisfies the timing, GTR is skipped then GPR tries to squeeze out as much power as possible by greedily downsizing the most sensitive gates without violating any constraint [3]. GTR very effectively complements LR, as GTR allows the final solution at the end of LDP to have small timing violations. Owing to its global view, LR is not very efficient in recovering timing exactly. To resolve the remaining timing violations, LR ends up expending more power than a more localized greedy approach like GTR. The GTR algorithm will be discussed in Section V-C. Both GPR and GTR lack parallelism, hence they are left sequential in this work.

[2]A cell is invalid if it causes capacitance or slew constraint violations.

```
1   Algorithm: MT-LRS via Clustering and Leveling
2   foreach level l
3       readyQ = cluster(l)
4       // Each thread executes in parallel
5       while (1)
6           # critical section
7               if (readyQ.empty()) break
8               z = readyQ.next()
9           # endcritical
10          process_cluster(z)
11      endwhile
12      barrier()
13  endfor
```

Figure 4: Pseudo code for MT-LRS by the simple approach.

IV. Multi-threaded Approach

We propose techniques to parallelize the following sub-blocks: LRS solver, STA, and LM update. Parallelization of the LRS is our key focus since its the most runtime expensive of all, so we discuss it first. We shall refer to multi-threaded LRS as MT-LRS.

A. Requirements for Multi-threading the LRS

As mentioned above, LRS involves OLR of all the gates and several gates can be processed simultaneously. We shall use OLR and "processing" interchangeably. Simultaneous OLR of two or more gates requires all of them to satisfy two properties that we have identified as follows:

Property 1: None of them should have a fanin in common. Otherwise, when two or more gates sharing a fanin are being resized, some of them might witness an unexpected change in the fanin's load while they are in the middle of OLR. Even worse, the fanin *maxcap* constraint might be violated if gates commit their new sizes simultaneously. This would require one or more gates to redo OLR, which can become very expensive and therefore should be avoided.

Property 2: None of them should lie in the fanout cone of a gate undergoing OLR. Otherwise, the gate in the fanout cone might be using the stale values for input slew. This can be easily avoided by following a forward topological order as used in the sequential approach (see Section III-B2). However, unlike the sequential execution, precomputing an order with multiple threads is undesirable because processing time of the gates is not known *a priori*. Therefore, instead of a precomputed order, a dynamic structure like a queue of *ready* gates whose fanins have already been resized needs to be maintained.

To ensure both of the above properties, we first propose a simple approach, followed by the MEE assignment and the DNT.

B. A Simple Approach - Clustering and Leveling

To ensure the first property, we propose to cluster all the gates that share fanins and process them by a single thread. If gate A shares a fanin with gate B, and B shares a fanin with gate C, then all three gates should be clustered, even if A and C do not share any fanin. Note that the clusters are always disjoint. The advantage of clustering is that it is simple to implement and the clusters can be precomputed, but the disadvantage is that it can cause heavy load imbalancing if the cluster sizes are highly non-uniform.

To ensure the second property, we propose to group all the clusters by their topological level (PIs being at level 0) and process one level at a time. We refer to this as *leveling*. If a cluster spans multiple levels, it can be safely broken down into that many sub-clusters. When all

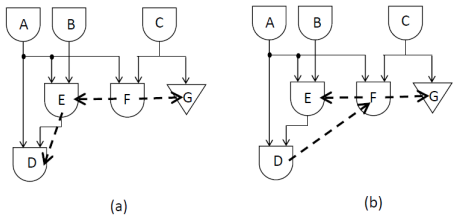

Figure 5: (a) One possible MEE assignment: Fanouts of A - {D,E,F} are chained by MEEs (dashed lines). Similarly, fanouts of C - {F,G} are chained. With MEEs, the fanouts of A will be processed in the following order: F then E then D. (b) An incorrect MEE assignment as it forms the cycle D-F-E. Edges D-F and F-E are MEEs, and the edge E-D is a netlist edge - due to D being a fanout of E.

the clusters at a level have been processed, all the clusters in the next level at once become ready. The advantage of leveling is that it does not require any book-keeping to determine when a cluster becomes ready, whereas the disadvantage is that threads need to wait for every other thread to finish before moving onto the next level.

Figure 4 shows the pseudo code for MT-LRS with this approach. At each level, a queue (*readyQ*) of ready clusters is initialized with the clusters at that level. Then, each thread enters a *critical section* to retrieve a cluster from the *readyQ*. In a critical section a mutually exclusive (mutex) lock is used to ensure that no other thread writes or reads the data that is being updated. Other threads wanting to read/write that data stall until the first thread exits the critical section. Threads process their respective clusters and re-enter the critical section. When the *readyQ* is empty, threads exit and wait at the *barrier* for the other threads to finish. In multi-threaded programming, barrier is used to synchronize all the threads at that point.

C. Mutual Exclusion Edge Assignment - An Alternative to Clustering

As noted above, clustering can cause heavy load imbalancing due to non-uniform cluster sizes. If we want to avoid clustering, we need an alternative mechanism to ensure the first property. We propose to *chain* the fanouts of every gate by additional edges, referred to as mutual exclusion edges (MEE), thereby ensuring that the fanouts of the same gate are not processed simultaneously. An example is shown in the Figure 5(a).

If the fanouts are arbitrarily chained then either cycles (Figure 5(b)) or very long chains might get created. While cycles would lead to deadlock, long chains would adversely affect the performance.

We propose a randomized algorithm for MEE assignment to avoid cycles and probabilistically reduce the maximum chain length. Its pseudo code is shown in Figure 6. It consists of two stages: (1) assignment of random IDs to each gate, and (2) assignment of MEEs. Random IDs are assigned such that the gates at the higher topological level have larger IDs. Then in the second stage, for every gate, its fanouts are sorted in ascending order of their IDs and an MEE is assigned between every consecutive pair of sorted fanouts - from the lower ID fanout to the higher ID fanout. Thus, the fanouts are chained. MEEs are assigned once and it has linear time complexity. In reference to the pseudo code, $s[i]$ is referred to as a *pseudofanin* of $s[i+1]$ and $s[i+1]$ is a *pseudofanout* of $s[i]$. In our scheme of MEE assignment, this strategy guarantees cycle-free assignment.

If during the ID assignment stage we do not ensure larger IDs for gates at higher levels (PIs are at level 0), then an MEE may get assigned from a higher to a lower level fanout. This may create cycles involving the netlist edges that are always from the lower to the higher levels.

While MEE can achieve much better load balancing than clustering, it creates additional edges which means more precedence

```
1   Algorithm: MEE Assignment
2   // Assign random ID
3   maxID = 0
4   for level l = 0 to maxlevel
5       x = maxID
6       foreach gate g in level l
7           g.ID = x
8           while (g.ID is not unique)
9               g.ID = x + rand()
10          endwhile
11          if (g.ID > maxID)
12              maxID = g.ID
13          endif
14      endfor
15  endfor
16  // Assign MEE
17  foreach gate g
18      s = sort fanouts of g by ascending IDs
19      for i = 0 to (s.size() − 2)
20          MEE(s[i] → s[i + 1])
21      endfor
22  endfor
```

Figure 6: Pseudo code for MEE assignment.

constraints that limit the parallelism. We empirically show that the thread idling time is actually very small for larger designs.

D. DAG Based Netlist Traversal - An Alternative to Leveling

Though leveling is a simple idea, it has two disadvantages: 1) a barrier at the end of each level causes thread idling, and 2) within a level, parallelism is limited by MEEs. The repercussions of barrier are more visible with the clustering where loads can be highly imbalanced. With an increasing number of threads, this barrier and the limited parallelism worsen the thread utilization. An alternative approach to leveling is DAG based netlist traversal (DNT). By some book-keeping, we can track the gates as they become ready, and keep pushing them into the *readyQ*. As long as the *readyQ* is non-empty, threads need not wait.

Pseudo code for two book-keeping functions needed to implement the DNT are shown in Figure 7. The first function, *init_precedence_count(gate g)*, initializes the precedence count (*preCount*) of the gate g. *preCount* is the number of predecessors which can be either fanins or *pseudofanins*. The second function, *identify_ready_gates(gate g)*, is invoked when the gate g has been processed. It decrements the *preCount* of each one of its fanouts as well as *pseudofanouts*. Those fanouts or pseudofanouts whose *preCount* reaches zero are returned as ready gates.

The disadvantage of DNT over leveling is that it requires book-keeping to track the ready gates. This needs to be done inside a critical section because multiple threads might want to simultaneously read/write the *preCount* of the same gate. However, a critical section is in any case required to update the *readyQ*. Therefore, by merging the two critical sections, we can optimize away the additional thread idling.

Next, we present a better approach for MT-LRS with MEE assignment and DNT.

E. Modified Approach - An Alternative to the Simple Approach

This modified approach replaces clustering and leveling by MEE assignment and DNT, respectively. Like clustering, the MEE assignment is also required only once in the beginning. On the other hand, the DNT must perform book-keeping every time MT-LRS is invoked. Figure 8 shows the pseudo code of MT-LRS via the modified approach. Firstly, the *preCount* of each gate is initialized. The gates

```
1  Algorithm: init_precedence_count (gate g)
2  g.preCount = |g.fanins| + |g.pseudofanins|

1  Algorithm: identify_ready_gates (gate g)
2  readyG = ∅        // Ready gates
3  foreach x ∈ (g.fanout ∪ g.pseudofanout)
4      x.preCount = x.preCount − 1
5      if (x.preCount == 0)
6          readyG.push_back(x)
7      endif
8  endfor
9  return readyG
```

Figure 7: Pseudo codes for two book-keeping functions of DNT.

```
1  Algorithm: MT-LRS via MEE Assignment and DNT
2  foreach gate g
3      init_precendence_count(g)
4  endfor
5  init_readyQ()
6  // Each thread executes in parallel
7  # critical section
8      if (all_gates_done()) return
9      g = readyQ.next()
10 # endcritical
11 while (1)
12     process_gate(g)
13     # critical section
14         x = identify_ready_gates(g)
15         update_readyQ(x)
16         if (all_gates_done()) return
17         g = readyQ.next()
18     # endcritical
19 endwhile
```

Figure 8: Pseudo code for the MT-LRS via the modified approach.

with zero *preCount* are the ready gates and they form the *readyQ*. Then, each thread retrieves a ready gate (more gates can also be retrieved) from the *readyQ* and processes it. Processed gates identify new ready gates, if any, and update the *readyQ*. If all the gates have been processed, the thread exits, otherwise it retrieves the next ready gate and processes it. Note that unlike in the leveling, an empty *readyQ* does not imply all the gates are processed, rather it means that some threads are still working and they might soon generate new ready gates.

F. Parallelizing STA and LM update

Parallelizing the STA and the LM update is much more straightforward than parallelizing the LRS. In STA, gate timings are updated in the forward topological order; whereas in LM update, gate LMs are updated in the reverse topological order. We apply the leveling idea for topological traversal. Note that the clustering is not needed in either STA or LM update, because gates are not being resized. Therefore, at each topological level, there is total freedom to update any number of gates simultaneously and in any order. We form groups of ten gates at each level and feed them to the threads whenever threads become available. When all the gates at a level are updated, we go to the next level. Note that the STA and the LM update do not involve much computation and the time spent updating a group of gates is more or less the same across all the groups. Therefore, an explicit barrier at the end of each level does not degrade the thread utilization much.

V. Enhancements in Sequential Approach

To complement the multi-threading performance improvements, the sequential approach should be free from sub-optimality as far as possible. Although the recent LR-based gate sizers have been demonstrated to be the fastest on the ISPD 2012 benchmarks, at several places in the general strategy we observe sub-optimal performance. For example: during the OLR of a gate, [13]–[15] suggest evaluating all the cells, but this may not be necessary in all the iterations. While [14] executes a fixed number of LDP iterations, [15] does not define convergence criteria for when LDP can be terminated. To address these issues and more, we propose the following three enhancements.

```
1  Algorithm: Fast-OLR
2  candidates = ∅
3  currw = current width
4  for current Vth, one Vth higher and one Vth lower
5      c = cell(currw, Vth)
6      bestcost = cost(c)
7      bestcell = c
8      for w = currw: maxw
9          c = cell(w, Vth)
10         Ensure c is valid
11         if cost(c) < bestcost
12             bestcost = cost(c)
13             bestcell = c
14         else break
15     endfor
16     candidates.insert(bestcell)
17     // Repeat lines 5-16 and
18     // replace maxw by minw
19 endfor
20 Apply min_{c∈candidates} cost(c) such that
21 local slack does not worsen
```

Figure 9: Pseudo-code for Fast-OLR.

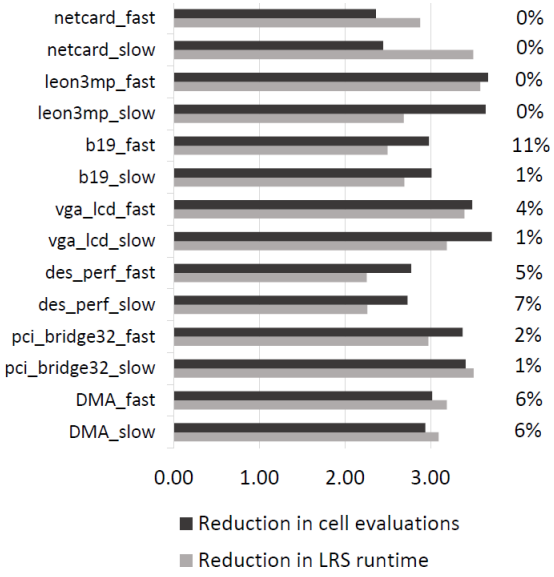

Figure 10: The bar chart shows the reductions in the cell evaluations and the LRS runtime due to Fast-OLR. The numbers on the right of each bar denote the power savings after the LDP stage, relative to OLR.

A. Fast-OLR

During the OLR, a gate is resized to the cell that has the lowest cost (*lambda-delay-cost + leakage*). To determine such a cell, [13]–[15] suggest evaluating all the valid alternatives. However in practice, we observed that the cost function almost always has a single local minimum considering a fixed Vth and varying sizes. Therefore, by searching for the optimal cell locally, we can reach the globally optimal cell most of the time. Although this may be an artifact

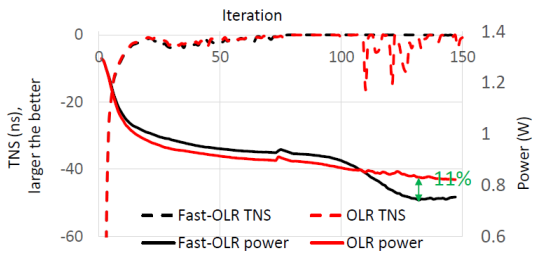

Figure 11: Comparing TNS and power profiles due to Fast-OLR and OLR, on b19_fast. After 100 iterations, TNS destabilizes due to OLR. Consequently, OLR cannot focus on power recovery, and ends up with 11% higher power.

of the ISPD 2012 contest library, there is another advantage of local searching which is library independent. Local searching induces incremental changes to the current solution, whereas jumping to the globally optimal cell may significantly perturb the current solution. Large perturbations during the last few iterations tend to destabilize the solution, preventing convergence and degrading the results.

Figure 9 shows the pseudo code of our proposed algorithm, Fast-OLR. A cell is characterized by its size (or width) and Vth. Since we do local searching, we restrict our search to the current Vth, the next higher Vth and the next lower Vth cells. For each of the three Vth, we iterate over the cells with increasing sizes. We continue as long as the cost is reducing, and store the least cost cell as a suitable candidate. The same procedure is repeated for the decreasing sizes. At the end, the least cost candidate that does not worsen the local slack much is applied.

Before choosing the least cost cell, Flach et al. suggested [15] computing the change in the slack of the driver and the sink nets because the locally optimal cell, whether found by local searching or otherwise, might significantly worsen the TNS. We apply this check as we recover power, after the design timing is within 1% of the target delay.

In Figure 10, we compare the Fast-OLR against OLR for the following three metrics: the number of cell evaluations, the LRS runtime, and the power after LDP. Results are from single-threaded execution. On average, the cell evaluations reduce by 3.3x and the LRS runtime reduces by 3.0x. We also observe an average 3% reduction in the power due to the better solution stability offered by local searching, as discussed above. In Figure 11, we demonstrate destabilization of TNS due to OLR after 100 iterations for the b19_fast benchmark. If the local slack worsens significantly that further adds to the instability.

B. Early Exit Policy

When the timing constraint has almost been met, the early-exit policy determines if it is likely that power will reduce in future iterations or not. It can be used as a rule of thumb to terminate the LDP iterations before the maximum number of iterations are reached. *The LDP solver can be terminated if neither the average power, nor the minimum power solution found thus far, improve during two consecutive sets of iterations.*

The early exit policy is derived based on the following empirical observations: power averaged over a few iterations (5, in this work) reduces before stabilizing; in general, power is not a monotonic function of the number of iterations; and, in some cases, power may oscillate around a value. If the power is oscillating, we may never observe power degradation for two consecutive sets of iterations. So there must be a reduction in the minimum power found thus far to justify continuing. Our experiments showed that some designs (large, as well as small) terminate LDP after 35 iterations, whereas some required up to 160 iterations. The benchmarks required on average 95 iterations to converge, with a standard deviation of 49.

C. Fast-GTR

The GTR applied in the last phase of the optimization serves to fix small timing violations without expending much power. At this stage timing degradation is not allowed. Although different researchers refer to it by different names like Slack Legalization [3] and Timing Recovery [15], the basic algorithm is the same. It's an iterative algorithm: in each iteration, the critical gates are sorted by some criterion like slack, or change in the delay after upsizing; then the most critical gate is upsized; followed by an incremental STA. If the TNS degrades, the change is undone. GTR terminates when all timing violations have been fixed.

The sub-optimality here is that the incremental STA can be a significant waste of CPU cycles if the TNS degrades. One of the solutions is to have a metric that can predict if the timing is going to degrade. Hu *et. al.* [3] developed one such metric for their approach. However, it may generate false negatives, thereby causing a wasteful incremental STA nonetheless. We propose a very simple heuristic, referred to as Fast-GTR. It is based on the empirical observation that the gates that fail to improve the timing in the current iteration, are unlikely to improve the timing in the future iterations as well, unless one of its side gates (a gate that shares a fanin) is successfully upsized. Therefore, we simply skip such gates until then.

In general, Fast-GTR improves the parallelized fraction of the total runtime by speeding up the greedy post-pass stage which directly benefits the multi-threaded speedup (discussed in Section VI-C). In particular, we observed that Timing Recovery [15] can consume up to 50% of the total runtime on the benchmarks like des_perf, whereas Fast-GTR consumes less than 5% of runtime without degrading the results or causing any timing constraint violations.

VI. Experimental Results

Our gate-sizer is implemented in C++. Experiments are performed on a server with two 2.67GHz Intel(R) Xeon(R) X5650 CPUs. Each CPU has six cores and each core has two hyperthreads. Aggregate memory is 48GB. For multi-threading, OpenMP [25] is used. We use ISPD 2012 gate-sizing contest benchmarks to test our sizer. All of our reported results are averaged over 5 runs and all the final optimized designs satisfy the timing constraints.

We compare our results against works of the other researchers, namely [15] and [13]. [15] used a single thread on a faster machine, a 3.40GHz Intel(R) Core(TM) i7-3770 CPU. [13] employed 8 threads on a server like ours with two 2.67GHz CPUs with 6 cores and 72GB memory. Since we do not have access to their source code, results are cited from their respective works.

In this section we shall discuss results pertaining to different types of executions. Their nomenclature is defined as follows: the modified approach + Fast-GTR is referred to as Fast-Fast (FF); the modified approach + Timing Recovery [15] is Fast-Slow (FS); and, the simple approach + Fast-GTR is Slow-Fast (SF). When x threads are used, they are respectively referred to as FFx, FSx and SFx. All of them are equipped with the Fast-OLR (we switch from OLR to Fast-OLR at the fifth iteration) as well as the Early Exit policy.

A. Comparing Power and Performance Against Previous Works

Referring to Table I, we first compare FF1 against [15]. FF1 is at par with [15], averaging 3% faster and 2.5% higher leakage power. On benchmarks with more than 500K gates, FF1 is 23% faster, primarily due to the early exit policy. Consequently, FF1 provides an excellent baseline to showcase the multi-threaded speedup that can be achieved. Compared to [15], FF8 is on average 5.40x faster (multiply the last

TABLE I: Comparing quality and performance of our single-threaded (FF1) and 8-threaded (FF8) against results in [13] and [15].

Benchmark	Total Cells	Leakage Power (W)						Total Runtime (min)					
		[13]	[15]	FF1	FF8	FF1/ [15]	FF8/FF1	[13]	[15]	FF1	FF8	[15]/FF1	FF1/FF8
DMA_slow	25301	0.153	0.132	0.136	0.135	1.027	0.995	0.60	0.79	0.64	0.15	1.23	4.28
DMA_fast	25301	0.281	0.238	0.250	0.250	1.050	0.999	0.60	0.92	1.48	0.34	0.62	4.37
pci_bridge32_slow	33203	0.111	0.096	0.099	0.100	1.036	1.003	1.20	0.87	1.78	0.34	0.49	5.29
pci_bridge32_fast	33203	0.167	0.136	0.143	0.143	1.053	1.000	1.20	0.92	1.95	0.37	0.47	5.21
des_perf_slow	111229	0.671	0.570	0.597	0.595	1.048	0.996	6.00	25.31	1.79	0.51	14.11	3.53
des_perf_fast	111229	1.930	1.395	1.457	1.457	1.045	1.000	6.60	16.37	5.82	1.38	2.81	4.23
vga_lcd_slow	164891	0.375	0.328	0.330	0.330	1.007	1.000	7.80	5.67	9.08	1.61	0.62	5.64
vga_lcd_fast	164891	0.460	0.413	0.432	0.432	1.046	1.000	10.20	8.37	11.70	1.94	0.72	6.04
b19_slow	219268	0.604	0.564	0.568	0.569	1.007	1.001	10.20	9.15	20.55	3.45	0.45	5.95
b19_fast	219268	0.784	0.717	0.734	0.734	1.024	0.999	12.00	11.75	21.88	3.60	0.54	6.08
leon3mp_slow	649191	1.400	1.334	1.333	1.334	0.999	1.000	43.80	38.98	26.75	4.53	1.46	5.91
leon3mp_fast	649191	1.640	1.443	1.445	1.442	1.002	0.998	54.60	46.62	33.55	5.94	1.39	5.65
netcard_slow	958780	1.780	1.763	1.763	1.763	1.000	1.000	48.00	34.39	37.07	5.86	0.93	6.32
netcard_fast	958780	2.180	1.841	1.849	1.850	1.004	1.001	88.80	47.41	41.82	7.38	1.13	5.67
Average		**0.895**	**0.784**	**0.796**	**0.795**	**1.025**	**0.999**	**20.83**	**17.68**	**15.42**	**2.67**	**1.03**†	**5.23**†

† Geometric mean. The geometric mean has been used to compare speedups due to the wider range in values.

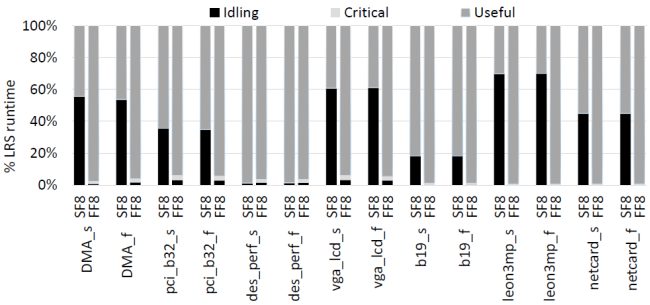

Figure 12: Comparing the MT-LRS runtime breakdown for the modified approach (FF8) and the simple approach (SF8).

two columns) with only 2.2% higher power. Runtime improvement is likely to be even more than 5.40x as Flach et al. [15] use a faster machine.

Compared to [13], which also used 8 threads, FF8 spends 7.80x lesser runtime ($= 20.83/2.67$) to execute all the benchmarks and saves 12% more power. With FF8 we demonstrate an average speedup of 5.23x over FF1 without degrading results. In comparison, Li et al. [13] reported an overall speedup of only 2.2x with 8 threads, mainly because 40% of their algorithm's runtime is unparallelized.

B. Comparison Against the Simple Approach

To compare the modified (FF8) and the simple approach (SF8), we analyze the runtime of MT-LRS for both of them in Figure 12. A thread executing LRS would either be idling, executing a critical section, or doing useful work, i.e., resizing. For SF8, we observed that on 10 out of 14 benchmarks, threads could be idling for $>35\%$ of the LRS runtime. This is mainly the waiting time at the barrier caused by heavy load imbalancing, which is caused by the clustering approach. The worst cluster sizes at a given level can have up to 46% of the gates at that level. On des_perf benchmark where SF8's idling time is only 1%, the worst cluster sizes were no bigger than 1%. In the worst case for FF8, threads idle for only $< 3\%$ of the LRS runtime.

The time spent in the critical section by SF8 threads is negligible. On the other hand, FF8 threads can expend up to 3% of the LRS runtime in the critical section. This is due to book-keeping to track the ready gates and updating of the *readyQ*. Across all the benchmarks, the average thread utilization for FF8 threads is 97%, and for SF8 threads it is only 59%. As a result, the FF8 MT-LRS is on average 1.73x faster than SF8 MT-LRS.

TABLE II: Overall runtime speedup with Fast-GTR (FF8) versus Timing Recovery (FS8). Speedups are roughly correlated with the parallelized runtime fractions. Major improvements are highlighted in bold.

Benchmark	Speedup		Parallel Fraction	
	FS1/FS8	FF1/FF8	FS1	FF1
DMA_slow	4.62	4.28	0.93	0.91
DMA_fast	4.86	4.37	0.96	0.96
pci_bridge32_slow	5.72	5.29	0.98	0.97
pci_bridge32_fast	5.50	5.21	0.97	0.97
des_perf_slow	**1.83**	**3.53**	**0.27**	**0.90**
des_perf_fast	**2.38**	**4.23**	**0.65**	**0.94**
vga_lcd_slow	5.83	5.64	0.98	0.97
vga_lcd_fast	5.86	6.04	0.97	0.97
b19_slow	5.82	5.95	0.98	0.98
b19_fast	6.17	6.08	0.98	0.98
leon3mp_slow	5.82	5.91	0.92	0.94
leon3mp_fast	5.09	5.65	**0.88**	**0.95**
netcard_slow	6.43	6.32	0.97	0.97
netcard_fast	4.40	5.67	0.90	0.88
Geometric Mean	**4.77**	**5.23**	**0.84**	**0.95**

C. Impact of Fast-GTR on the Overall Speedup

In Table II we compare the speedup in the total runtime achieved by FF8 (with Fast-GTR) and FS8 (without Fast-GTR) with respect to their corresponding single-threaded versions. On average, FF8 is 5.23x faster than FF1, whereas FS8 is 4.77x faster than FS1. FF is faster due to the improvement in the parallel fraction of the sequential runtime, from 0.84 for FS8 to 0.95 for FF8. On des_perf and leon3mp, Fast-GTR significantly reduces the time spent in the greedy post-pass as shown in the same table. We observed larger speedup with FF in netcard_fast despite slightly lower parallel fraction. This is supposedly due to the runtime noise caused by server loading.

D. Scalability Analysis

In this subsection, we analyze how the speedup scales as the number of threads grow and what factors contribute to the loss of the speedup. In Table III, we show the speedups obtained from FF2, FF4, FF8, FF12, FF14 and FF16 with respect to FF1.

We see a performance saturation from 12 to 14 threads, and slight performance degradation from 14 to 16 threads. The average speed-up with 16 threads is 6.04x which is significantly smaller than the theoretical upper bound of 9x predicted by Amdahl's law. There are two primary factors for this gap: 1) limitations of the hardware architecture, and 2) increase in overhead. The server has 2 CPUs each with 6 cores with 2 hyper threads. The two hyperthreads per core do not provide much additional speedup due to hardware resource contention between threads [26]. As a result speedup begins saturating as threads exceed the physical cores. Our experiments on a

TABLE III: Speedup for different threads with respect to FF1.

Benchmark	FF1	FF2	FF4	FF8	FF12	FF14	FF16
DMA_s	1.00	1.71	2.77	4.28	5.36	4.62	4.40
DMA_f	1.00	1.64	2.75	4.37	5.89	5.52	4.86
pci_bridge32_s	1.00	1.95	3.13	5.29	6.18	6.00	5.56
pci_bridge32_f	1.00	1.96	3.19	5.21	6.58	6.16	5.63
des_perf_s	1.00	1.57	2.37	3.53	4.15	4.39	3.93
des_perf_f	1.00	1.66	2.73	4.23	3.06	4.90	4.72
vga_lcd_s	1.00	1.88	3.10	5.64	7.85	7.48	6.75
vga_lcd_f	1.00	1.76	2.89	6.04	6.25	6.12	6.63
b19_s	1.00	1.83	3.11	5.95	7.57	7.63	7.65
b19_f	1.00	1.66	3.23	6.08	7.96	7.75	7.69
leon3mp_s	1.00	1.70	3.14	5.91	7.75	7.46	7.34
leon3mp_f	1.00	1.85	2.86	5.65	7.03	7.18	7.08
netcard_s	1.00	1.95	2.97	6.32	7.97	8.13	8.05
netcard_f	1.00	1.97	2.86	5.67	5.21	6.08	6.37
Geom Mean	**1.00**	**1.79**	**2.93**	**5.23**	**6.14**	**6.27**	**6.04**

synthetic completely parallel code showed that we could only achieve a 12x speed-up using 16-threads. In other words, we effectively have only 12 threads. Considering this, we would expect the upper-bound of the achievable speed-up on our multi-threaded application to be 7.74x, which is closer to our achieved result. With an increase in the number of threads, 1) thread idling increases, and 2) critical section overhead increases. Additionally, we suspect runtime overhead due to increased communication to keep memories synchronized.

VII. Conclusion

Today's designs with millions of gates require very fast gate-sizing and threshold voltage assignment, as it is a crucial circuit optimization that is performed at multiple steps in the design flow. We have shown the effectiveness of our proposed techniques to speed up multi-threading of Lagrangian relaxation-based gate sizing; mutual exclusion edge assignment and DAG-based netlist traversal help achieve 97% thread utilization with 8 threads. In contrast, the simpler multi-threading strategies - clustering and topological level-based traversal, allow only 59% thread utilization. To complement our multi-threading techniques, we also propose fast optimal local resizing, early-exit policy, and fast greedy timing recovery, all of which are simple yet highly performance effective enhancements to the sequential LR-based gate sizing approach. We combine all these to realize a very fast and high quality gate sizer. Compared to the state-of-the-art (both in runtime as well as power) algorithm [15], our gate sizer using 8 threads is 5.40x faster and has only 2.2% higher power, without any timing constraint or other violations, on the ISPD 2012 Disrete Gate Sizing Contest benchmarks.

VIII. Acknowledgment

This work is partially supported by NSF under grant CCF-1219100 and by Mentor Graphics. We would also like to thank Ivailo Nedelchev from Mentor Graphics for his valuable guidance.

References

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

[2] JP Fishburn. TILOS: A posynomial programming approach to transistor sizing. In *Proc. of IEEE International Conference of Computer-Aided Design, Nov. 1985*, 1985.

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

[4] D Nguyen *et al.* Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 158–163. ACM, 2003.

[5] DG Chinnery and K Keutzer. Linear programming for sizing, Vth and Vdd assignment. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 149–154. ACM, 2005.

[6] K Kasamsetty, M Ketkar, and SS Sapatnekar. A new class of convex functions for delay modeling and its application to the transistor sizing problem [CMOS gates]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(7):779–788, 2000.

[7] S Roy, CC-P Chen, and YH Hu. Numerically convex forms and their application in gate sizing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(9):1637–1647, 2007.

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

[9] H Tennakoon and C Sechen. Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 395–402. ACM, 2002.

[10] J Wang, D Das, and H Zhou. Gate sizing by Lagrangian relaxation revisited. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):1071–1084, 2009.

[11] MM Ozdal, S Burns, and J Hu. Gate sizing and device technology selection algorithms for high-performance industrial designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 724–731. IEEE Press, 2011.

[12] Y-L Huang, J Hu, and W Shi. Lagrangian relaxation for gate implementation selection. In *Proceedings of the 2011 international symposium on Physical design*, pages 167–174. ACM, 2011.

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

[14] VS Livramento *et al.* Fast and efficient lagrangian relaxation-based discrete gate sizing. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1855–1860. EDA Consortium, 2013.

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

[16] V Sundararajan, SS Sapatnekar, and KK Parhi. Fast and exact transistor sizing based on iterative relaxation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(5):568–581, 2002.

[17] H Ren and S Dutt. A Network-Flow Based Cell Sizing Algorithm. In *The International Workshop on Logic Synthesis*, 2008.

[18] S Hu, M Ketkar, and J Hu. Gate sizing for cell library-based designs. In *Proceedings of the 44th annual Design Automation Conference*, pages 847–852. ACM, 2007.

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

[20] Y Liu and J Hu. GPU-based parallelization for fast circuit optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(3):24, 2011.

[21] M Rahman, H Tennakoon, and C Sechen. Library-Based Cell-Size Selection Using Extended Logical Effort. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(7):1086–1099, 2013.

[22] T Reimann *et al.* Simultaneous gate sizing and vt assignment using fanin/fanout ratio and simulated annealing. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 2549–2552. IEEE, 2013.

[23] MM Ozdal *et al.* The ISPD-2012 discrete cell sizing contest and benchmark suite. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 161–164. ACM, 2012.

[24] MM Ozdal *et al.* An improved benchmark suite for the ISPD-2013 discrete cell sizing contest. In *Proceedings of the 2013 ACM international symposium on International symposium on physical design*, pages 168–170. ACM, 2013.

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

[26] A Valles. Performance Insights to Intel(r) Hyper-Threading Technology. Technical report, http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology, 2009.