Task-based Parallel Programming for Gate Sizing

Dimitrios Mangiras, David Chinnery and Giorgos Dimitrakopoulos, *Member, IEEE*

***Abstract*—Physical synthesis engines need to embrace all available parallelism to cope with the increasing complexity of modern designs and still offer high quality of results. To achieve this goal, the involved algorithms need to be expressed in a way that facilitates fast execution time across a range of computing platforms. In this work, we introduce a task-based parallel programming template that can be used for speeding up timing and power optimization. This approach utilizes all available parallelism and enables significant speedup relative to custom multithreaded approaches. Task-based parallelism is applied to all parts of the optimization engine covering also parts that are traditionally executed serially for preserving maximum timing accuracy. Using Taskflow as the parallel programming and execution engine, we achieved a speedup of 1.7× to 2.8× for gate sizing optimizations on the ISPD13 benchmarks with marginal extra leakage power relative to state-of-the-art multithreaded gate sizers. This result was supported by two dynamic heuristics that restrict the number of examined gate sizes and simplify local timing updates. Both heuristics trade off additional runtime reduction with marginal leakage power increases.**

***Index Terms*—task-based parallel programming, timing and power optimization, physical design, electronic design automation.**

I. Introduction

The implementation of physical synthesis algorithms should satisfy multiple contradictory goals [1]. First comes Quality-of-Results (QoR): to place and route a design that satisfies the required timing, area and power constraints [2], [3]. Then comes efficiency that should not compromise QoR: execute physical synthesis algorithms in the least amount of time even for very large designs. Last but not least is performance portability [4]: the implementation should be agnostic to the particularities of the hardware platform thus enabling physical synthesis engines to take advantage of new diverse computing hardware with the least effort for software adaptation.

Currently, the majority of the physical synthesis engines are manually parallelized with custom thread pools and work allocators developed using well-known multi-threaded programming interfaces [5], [6], [7]. Even if such efforts have shown good scalability, the performance starts to level off after a few active threads. There are multiple reasons for this limitation. In many cases, the algorithms are graph oriented and although they exhibit high degrees of parallelism, the patterns of parallelism involve irregular computations and possibly poor data locality that are harder to exploit than the structured parallelism patterns found in computational science [8], [9]. Also, the combined engineering effort required to describe algorithmic novelty together with multithreaded efficiency *is not trivial* and may lead to sub-optimal results. Additionally, custom thread management and scheduling may be inefficient or hard to adapt to new computing platforms.

To overcome such limitations, our goal is to *separate* algorithm development, that should focus solely at the relevant problem, e.g., placement, routing, or timing optimization, from its parallel execution. To achieve this goal, we argue that *physical synthesis algorithms* should be described as *task-based parallel programs*.

In task-based models, the programmers define elementary blocks of source code as individual tasks and express dependence relationships between those tasks [9]. In this way, the programmers do not manage processes or threads anymore but they focus only on how to decompose their program into tasks. Contrary to other parallel programming approaches, task-based programming is easier, safer and more efficient to human programmers [4]. In this way, physical synthesis software architecture is stable and can enjoy long-term availability, ease of maintenance and high performance across current and future computing platforms.

In this work, we derive a generic template for timing optimization using task-based parallel programming and apply it to gate sizing, i.e., select for each gate an appropriate size and threshold voltage from a discrete set of library cells [1]. The same template can be used for various forms of timing optimization such as timing-driven placement [10] or logic restructuring [11]. The presented approach can be used both for global timing optimization at the first steps of the physical synthesis flow or close to the end where repairing timing violations requires incremental operations that are nondisruptive and execute as fast as possible [12].

Timing optimization algorithms are often iterative and exhibit irregular computational patterns and complex control flow [10], [13]. For this reason, we selected Taskflow for transforming the multi-step timing optimization to a task-based parallel program. Other candidates, such as Intel oneTBB FlowGraph [6], OpenMP tasks [7] were not chosen. These models require programmers to implement control flow decisions outside the task dependency graph thus creating complicated implementations that compromise parallelism [9]. Taskflow [9] offers a simpler programming interface and allows building hierarchical task graphs. Also, it supports conditional dependencies and cyclic execution patterns that have already been used in accelerating static timing analysis [14] and detailed placement [15].

In overall, this work's contributions are summarized as follows:

- We introduce a generic task-based parallel programming template for timing optimization and test it on gate sizing algorithms. The presented approach covers all phases of a powerful Langrange-relaxation based gate sizer covering

Dimitrios Mangiras and Giorgos Dimitrakopoulos are with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece. Dimitrios Mangiras is supported by the Onassis Foundation - Scholarship ID: G ZO 014-1/2018-2019.
(e-mail: dmangira@ee.duth.gr, dimitrak@ee.duth.gr).

David Chinnery is with Siemens Digital Industries Software, Fremont, USA (e-mail: david.chinnery@siemens.com).

initial sizing, main iterative sizing and final recovery steps. All steps are parallelized for the first time –to the best of our knowledge– without requiring any steps executed serially and without compromising quality of results.
- For better exploring the runtime vs quality tradeoff, we propose two heuristics that (a) re-evaluate at each iteration the search space of examined sizes per gate and (b) dynamically assess the criticality of local timing arcs. The gates with not-critical timing arcs are pruned from the local timing graph thus speeding up local timing updates.
- Using the task-based formulation and reducing dynamically the examined sizes per gate gives a speedup of 1.7× to 2.8× when compared to state-of-the-art multithreaded gate sizers with only a marginal increase in leakage power. When enabling fast local timing updates, runtime is reduced further.

The rest of this paper is organized as follows: Section II discusses related work. Section III presents the overall template for a task-parallel gater sizer while Sections IV–VI describe its constituent parts. Experimental results are presented in Section VII and conclusions are drawn in the last Section.

II. Related Work

Gate sizing has been traditionally considered as a powerful tool for timing closure and power reduction that could execute in a reasonable runtime even for very large designs [16], [17]. Approaches that used linear programming have been also proposed. In these cases, positive slack was distributed to gates using linear programming with the goal to maximize power savings [18], [19]. Then, gate sizes were selected based on the available slack. Similarly, Held *et al.* [20] assigned slew targets, instead of delay targets.

Langrangian Relaxation (LR) has been widely used for design optimization in recent years. Ozdal *et al.* in [13] proposed a graph model to effectively decide the sizes of the LR-based gate sizing problem. LR-based gate sizing has been refined in [21] and [22]. To resize both data and clock gates Shklover *et al.* [23] extended the traditional LR method with clock-related formulations, while the work of [10] introduced a way to optimize all types of gates (e.g. flip-flops, combinational gates and clock buffers) using the same LR formulation.

Even the most efficient algorithms required parallelism to scale to increasing design sizes. A sensitivity-guided metaheuristic method that optimizes power and timing using parallel execution was proposed in [24] and enhanced in [25]. Sharma *et al.* in [22] implemented a multithreaded gate sizer using OpenMP obtaining good QoR with fast runtimes. This approach has been tested in an industrial setup in [26]. Intel's threading building blocks [6] have been used in the most computational intensive parts of the optimization flow of [27] and [28]. The Galois parallelization framework [8] has been used in [29], [30] to speedup global and maze routing. More closely related to this work, OpenTimer [14] exploited Taskflow to accelerate efficiently static timing analysis.

Other approaches have focused on speeding up execution of EDA algorithms in GPUs. Liu *et al.* [31] use a GPU to accelerate a dynamic-programming-based gate-sizer [32]. Also, a GPU has been employed in [33] to accelerate static timing analysis. GPUs were used also for accelerating path-based timing analysis [34], [35] and placement. In the latter case, global [36] and detailed placers [15] executed on GPUs show tremendous speedups in designs with million of gates when compared to multi-threaded implementations for CPUs.

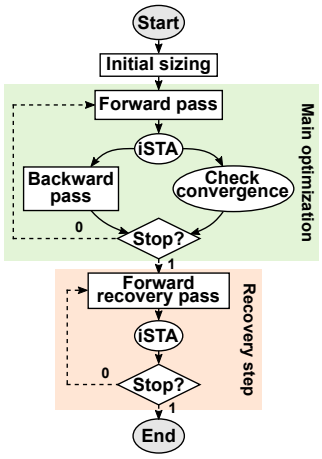


Fig. 1. The overall task dependency graph that consists of initial sizing, the main optimization loop and the final recovery loop.

III. Generic Gate Sizing Template

Timing and power optimization approaches consist mainly of three core steps: initial preparatory optimization, the main iterative optimization, and final timing and power recovery. The organization of these three main steps are depicted in the top-level task graph shown in Fig. 1. Rectangular tasks represent hierarchical blocks that can be further unwrapped to simpler tasks, round tasks are tasks executed at this level of abstraction, while diamond-shaped tasks represent conditional tasks. Conditional tasks check for a certain condition and determine accordingly the flow of execution. Solid-line edges between tasks represent dependence relations. On the contrary, dashed-line edges are conditional dependencies used in Taskflow [37] for describing cyclic processes.

Initial sizing that is executed first guarantees that gates are properly initialized so that they do not violate maximum load capacitance and maximum input slew design rules [27]. The forward pass (FP) of the main optimization loop selects appropriate sizes for each gate with the goal to minimize the selected cost function that reduces power and satisfies timing constraints. The result of this pass is quantified by the incremental static timing analysis (iSTA) step that follows. If power has not changed by more than 0.1% for three consecutive iterations the optimization moves to the recovery step. Otherwise, the program flow moves back to FP. This condition is checked by the convergence check task. In parallel, the backward pass (BP) updates the timing criticality of each gate using the updated timing information of iSTA.

The iterative recovery process begins only after the main optimization has converged. The goal of recovery is twofold: to correct the small remaining timing violations and re-

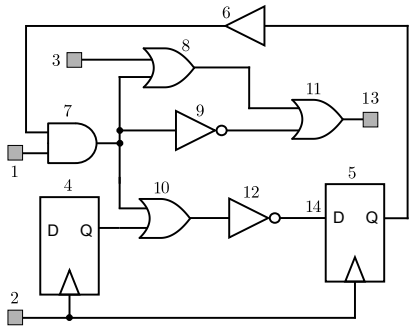


Fig. 2. The example logic-level netlist used as a running example.

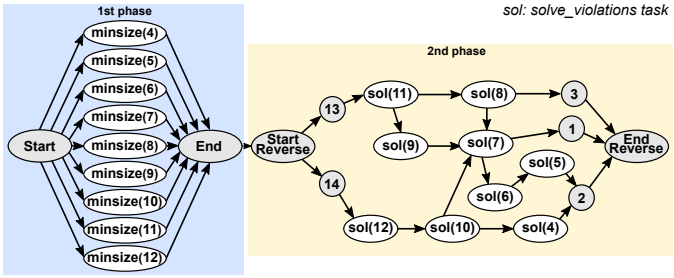


Fig. 3. The task graph of initial sizing is organized in two phases: In the first phase all cells are downsized in parallel. In the second phase the remaining load and slew violations are removed by visiting gates in reverse topological order.

cover power from gates with positive timing slack. To avoid disturbing the already optimized netlist, recovery executes carefully selected resizings with maximum timing accuracy at each step. State-of-the-art optimizers [21], [22] guarantee maximum timing accuracy by touching serially one gate at a time and updating timing globally after every update. In this work, we avoid serial execution and execute recovery in a *conservatively-parallel* way that allows for equivalent power savings but with significantly lower runtime. If timing or power improves, the recovery is repeated until all timing violations are solved and power stops changing. To assess timing improvement at the end of this recovery step a full incremental timing update takes place.

IV. Initial sizing

The purpose of the initial sizing is to solve any load and slew violation from the beginning. In this way, it is easier for the main optimization loop that follows to preserve this property without introducing new violations.

To implement initial sizing we map its operation to a specific task graph that is organized in two parts. In the first part all gates are downsized in parallel to their lowest size (and lowest leakage). In the second part, the gates are visited in reverse topological order to check and solve if the load and the slew constraints are violated. The first part does not involve any dependencies across tasks, while in the second part the execution of tasks follows the connectivity of the logic-level netlist in reverse order.

The task graph for initial sizing that corresponds to the design of Fig. 2 is depicted in Fig. 3 and built using Algorithm 1. The first phase contains one task for each gate or flip-flop of the design without dependencies between them. In the second phase, there are tasks which represent primary inputs (PIs) and outputs (POs), D-pins of the flip-flops, gates as well as flip-flops (FFs). From these tasks, only the tasks that represent gates and flip-flops are assigned with work. Pseudo-tasks are created to help in dependency propagation. Such tasks are represented with grey-colored circles. Dependencies guarantee that each task is executed after the tasks of its fanouts in logic level. Introducing separate tasks for the D and clock/Q pins of the flip-flops breaks any possible sequential loop of the design thus resulting in acyclic task graphs.

Algorithm 1: Create Task Graph for Initial Sizing

```
1  create_task("Start");create_task("End"); // 1st phase
2  foreach g in {Gates ∪ FFs} do
3  |  create_task(g); task(g).assign_work(minsize(g));
4  |  task("Start").precede(task(g));
5  |  task(g).precede(task("End"));
6  end
7  create_task("Start-Reverse"); // 2nd phase
8  foreach po in {POs ∪ D-pins} do
9  |  create_task(po); task("Start-Reverse").precede(task(po));
10 end
11 create_task("End-Reverse");
12 foreach pi in {PIs} do
13 |  create_task(pi); task(pi).precede(task("End-Reverse"));
14 end
15 foreach g in {Gates ∪ FFs} do
16 |  create_task(g); task(g).assign_work(sol(g));
17 end
18 foreach i in {Gates ∪ PIs ∪ FFs} do
19 |  foreach f in { fanouts of i } do
20 |  |  task(f).precede(task(i));
21 |  end
22 end
23 task("End").precede(task("Start-Reverse"));
```

Algorithm 2: solve_violations(gate g)

```
1  sizes ← {equivalent sizes of g from cell library};
2  sizes_sort ← {sort sizes in ascending power order};
3  i ← 0;
4  while violates_load(g) or violates_slew(g) do
5  |  i++; resize g to sizes_sort[i];
6  end
```

The work executed at each task `solve_violations` is described in Algorithm 2. For each gate that is already at its minimum leakage size, we check whether the gate can drive its output load without introducing slew violations. The output slew is computed directly from the output slew lookup tables of the library using the already known output load and assuming the maximum-allowed slew for the inputs. If a load or a slew violation still exists the size of the gate is gradually increased until violations are removed. If none of the available sizes can solve the slew or load violations, only

logic restructuring can fix them (e.g., by adding buffers); but this problem does not occur on the ISPD 2013 benchmarks.

V. Main gate sizing optimization

Design optimization targets the minimization of the total leakage power without violating any timing constraints:

$$\begin{aligned} \textbf{minimize} \quad & \sum_{\forall \text{gate } i} leakage_i & (1)\\ \textbf{subject to} \quad & a_i + d_{ij} \le a_j, \quad \text{for each timing arc } i \to j \\ & a_k \le r_k, \quad \text{for each endpoint } k \end{aligned}$$

Variable a_i is the arrival time at pin i while r_k is the required arrival time at a primary output or a D-pin of a flip-flop k. d_{ij} is the delay of the timing arc $i \to j$ that consists of the wire delay from the output pin of gate i to the input pin of gate j plus the cell delay of gate j.

Lagrangian Relaxation associates a non-negative weight λ_{ij}, called Lagrange Multiplier (LM), to each constraint [38]. These weights act as penalty factors whenever the corresponding timing constraints are not met. Incorporating the constraints in the objective function transforms the problem to the following unconstrained one:

$$\textbf{min} \sum_{\forall \text{gate } i} leakage_i + \sum_{\forall \text{arc } i \to j} (a_i + d_{ij} - a_j)\lambda_{ij} + \sum_{\forall \text{endpoint } k} (a_k - r_k)\lambda_k \quad (2)$$

Differentiating (2) with respect to arrival times, according to the Karush-Kuhn-Tucker (KKT) optimality conditions [13], [10], we end up with the following LM conservation rule.

$$\sum_{\forall \text{fanin } i \text{ of } j} \lambda_{ij} = \sum_{\forall \text{fanout } k \text{ of } j} \lambda_{jk} \quad (3)$$

Equation (3) implies that the sum of the LMs of the arcs ending to a gate is equal to the sum of the LMs of the arcs starting from this gate. For example, the LM flow for gate 7 of Fig. 2 implies that $\lambda_{17} + \lambda_{67} = \lambda_{78} + \lambda_{79} + \lambda_{710}$. Replacing the equality condition of (3) to (2) simplifies the problem to:

$$\textbf{min} \quad \sum_{\forall \text{gate } i} leakage_i + \sum_{\forall \text{arc } i \to j} \lambda_{ij} d_{ij} \quad (4)$$

A. Forward Pass

State-of-the-art LR-based optimizers try to minimize cost function (4) using many iterations of gate resizing and V_T reassignment steps. At each iteration implemented by FP all gates are visited in topological order and for each gate the best size is selected assuming constant LMs.

1) FP Task graph: FP uses a task graph different from the one used for initial sizing. The task graph of FP that corresponds to the example of Fig. 2 is shown in Fig. 4 and built using Algorithm 3. Primary inputs, primary outputs, and the D-pins of the flip flops are assigned to pseudo-tasks (grey circles). A task implementing the `resize` function of Algorithm 4 is assigned to each gate and flip-flop of the design (line 9 in Alg. 3). The dependencies across tasks follow the forward topological order of the logic-level netlist. After "Start" the pseudo-tasks of the primary input pins are visited first, while the rest dependencies follow the netlist connectivity

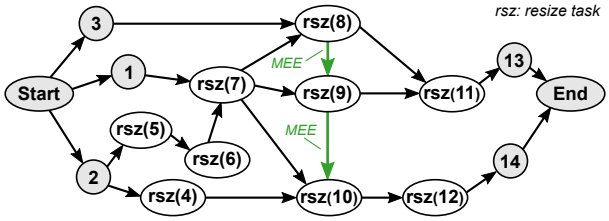


Fig. 4. In the forward task graph, the gates are visited in topological order to find new size. Apart from the logic-level dependencies (black color), the MEEs (colored green) of [22] are added to prevent the simultaneous sizing of gates driven by the same driver, e.g. gates 8, 9 and 10.

Algorithm 3: Create Task Graph for Forward Pass

```
1  create_task("Start"); create_task("End");
2  foreach pi in {PIs} do
3  |  create_task(pi); task("Start").precede(task(pi));
4  end
5  foreach po in {POs ∪ D-pins} do
6  |  create_task(po); task(po).precede(task("End"));
7  end
8  foreach g in {Gates ∪ FFs} do
9  |  create_task(g); task(g).assign_work(rsz(g));
10 end
11 foreach i of {Gates ∪ PIs ∪ FFs} do
12 |  f_i ← {fanouts of i in topological order} ;
13 |  foreach f in f_i do
14 |  |  task(i).precede(task(f));
15 |  |  next_f ← {the fanout next of f in f_i};
16 |  |  if dependency f → next_f doesn't exist then
17 |  |  |  task(f).precede(task(next_f)); // MEE
18 |  |  end
19 |  end
20 end
```

which implies that the task of a gate precedes the tasks of its fanouts (line 14 in Alg. 3).

According to [22] the tasks that correspond to gates that have a common fanin gate cannot execute in parallel using different threads. If their sizing is done in parallel each gate would estimate the delay change of their common fanin differently thus possibly leading to wrong sizing decisions. Also, when one of them is sized without knowing the size of the other, since it is changing in parallel, it may violate the maximum allowed capacitance of their common fanin gate. To solve this problem, the tasks that correspond to gates with a common driver should be executed serially. To impose this serial execution additional dependencies are added (lines 15–18 in Alg. 3), called mutual exclusion edges (MEE). For instance in Fig. 2, gate 7 drives gates 8, 9 and 10. Therefore, besides the normal dependencies $7 \to 8$, $7 \to 9$ and $7 \to 10$ that arise from the forward topological order of the netlist in Fig. 4, two extra MEE dependencies are added between the tasks 8, 9, 10 to serialize their execution.

To decide to which pair of tasks we should add an MEE edge we consider the following rule. An edge $u \to v$ is added between tasks u and v if: (a) Tasks u and v correspond to

Algorithm 4: resize(gate g)

```
1  min_cost ← inf ;
2  best_size ← size(g) ;
3  init_slack ← local_TNS(g) ;
4  trial_sizes ← get_available_sizes(g) ;
5  foreach size s of trial_sizes do
6      resize g to s ;
7      if violates_load(g) or violates_slew(g) then
8          reject s ;
9      end
10     local_timing_update(g);
11     if local_TNS(g) < γ · init_slack then
12         reject s ;
13     end
14     cost ← leakage_g + Σ_{i→j around g} λ_ij d_ij;
15     if (cost < min_cost) then
16         min_cost ← cost;
17         best_size ← s;
18     end
19 end
20 resize g to best_size ;
21 local_timing_update(g);
```

gates that have the same driver in the logic level netlist; (b) u is visited before v in the forward topological ordering of the netlist. In this way, the corresponding tasks are executed serially and cyclic dependencies are avoided since an MEE is always a "forward" edge with respect to the topological order.

2) The task executed per gate: According to Algorithm 4, task `resize` stores first the current size of the gate and computes its local total negative slack (TNS). Local TNS corresponds to the negative slack at the output pin of the examined gate and the negative slacks at the output pins of its driving gates. Then, the cell resizing loop examines all available trial sizes and selects the one that minimizes the local cost function (4), without introducing load violations and without degrading the local TNS over a threshold γ [21], [39]. To compute the value of γ we use the same approach proposed in [21]: $\gamma = -\min(0, WNS)/T + 1$, where WNS is the worst negative slack of the design and T is the clock period. In this way, γ changes as the optimization evolves. The idea is to allow the local TNS to degrade a little bit for better solution space exploration, but at the same time to keep the local TNS under control. In the first few iterations, γ has a large value to allow the local TNS to have large degradation; and as timing improves, γ allows only fine-grained changes that do not disturb the already optimized design.

The local cost is calculated as the summation of the leakage power of the new size and the neighbor arc delays multiplied by their corresponding LMs. The neighbors of a gate are the ones connected at its fanin and fanout, including also the side gates, i.e., those that share a driver with the resized gate. For instance, for gate 9 in Fig. 2 the following arcs are involved in the local cost: its input arc ($7 \rightarrow 9$), the arcs of fanin ($1 \rightarrow 7$, $6 \rightarrow 7$), the arcs of fanout ($9 \rightarrow 11$, $8 \rightarrow 11$) and the input arc of gates 8 and 10 ($3 \rightarrow 8$, $7 \rightarrow 8$, $7 \rightarrow 10$, $4 \rightarrow 10$).

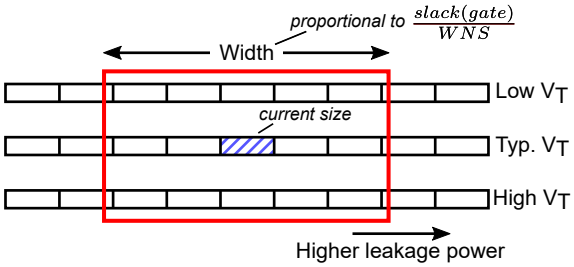


Fig. 5. The search window is centered around the current gate size and its width changes dynamically in proportion to the gate's slack divided by WNS. The width of the search window cannot reduce below three sizes per threshold.

In the baseline case, the available set of trial sizes examined per gate includes all possible alternatives. Also, during local timing update all neighbor arc delays are examined. In an effort to tradeoff runtime with QoR, we present two new approaches that can dynamically narrow down the set of trial sizes and examined timing arcs. Those heuristics are *only selectively enabled* and are not part of the baseline execution flow of gate sizing.

3) Reduce Trial Sizes (RTS): Based on the timing slack of each examined gate, we are not obliged to examine all available sizes for each gate. For instance, Sharma *et al.* in [22] observed that by trying all available sizes during the first five iterations of the main optimization loop and using only a subset of them for the rest iterations, is enough to accelerate gate sizing with good final QoR.

In this work, the set of sizes that need to be examined are dynamically decided at each iteration and separately per gate. The smallest set of available sizes corresponds to three sizes for each V_T, i.e., the currently selected size and the immediately smaller and larger gate size, times the number of available thresholds (nine in total for the benchmarks used in the experimental results). On the contrary, the largest set corresponds to all available sizes per V_T and all available thresholds (thirty sizes for the examined benchmarks). In all cases, the examined set of available sizes lies between those two extremes. How large is the search space depends on the negative slack of the output of the each gate: the more timing critical a gate is, the more options are tried to solve its violation fast.

If a gate has positive slack only the minimum of three sizes per V_T is tried. If the slack s is negative, the width of the search window centered around the currently selected size grows according to the ratio of $\frac{s}{WNS}$, as highlighted in Fig. 5. Put formally the width W of the window shown in Fig. 5 equals $W = 1 + \max(2, \#\text{sizes} \times \frac{\min(0,s)}{WNS})$

4) Fast Local Timing Update (FLTU): The local timing update calculates the new arc delays and the slews of the gates which are immediately affected after modifying a gate's size. The computed delays are used for computing the local cost function and the local TNS in Algorithm 4. To speedup this process, we dynamically alter which timing arcs are actually updated. We aim at skipping the update of timing arcs that are associated with relatively small LMs and do not affect the overall local cost. The proposed approach is detailed in Algorithm 5.

Algorithm 5: local_timing_update(gate g)

```
1  sum ← 0, neigh_arcs ← 0, upd_gates ← {} ;
2  foreach gate j in get_neighbors(g) do
3  |  foreach input arc i → j do
4  |  |  sum ← sum + λ_ij d_ij; neigh_arcs++ ;
5  |  end
6  end
7  crit_thres ← α · (1/neigh_arcs) ;
8  foreach gate j in get_neighbors(g) do
9  |  foreach input arc i → j do
10 |  |  crit ← λ_ij d_ij / sum ;
11 |  |  if crit > crit_thres then
12 |  |  |  upd_gates ← upd_gates ∪ j ;
13 |  |  end
14 |  end
15 end
16 if fast_local_STA_disabled or local_TNS(g) < 0 then
17 |  upd_gates ← get_neighbors(g);
18 end
19 foreach gate j of upd_gates in topological order do
20 |  update_slews_and_arrival_times(j) ;
21 end
22 foreach
     gate j of upd_gates in reverse topological order do
23 |  update_required_times_and_slacks(j) ;
24 end
```

Initially, the sum of the delays of the neighboring timing arcs multiplied with their corresponding LMs is computed, which is similar to the local version of Eq. (4) except of the leakage power term. For each neighbor, the real contribution of each timing arc is computed as the ratio of the arc's delay multiplied with the corresponding LM to the $\sum \lambda_{ij} d_{ij}$ of all neighboring arcs. When at least one of the neighbor's arcs contributes more than the threshold, the corresponding neighbor gate is not skipped. The threshold is set to $\alpha \cdot (1/\text{\#neigh\_arcs})$ where α is a non-negative weight that takes any value in between 0 and 1 and alters the number and which of the neighbors are skipped. For $\alpha = 0$, all the neighboring arcs contribute more than the threshold and thus, all neighbors are updated. On the contrary, for $\alpha = 1$ the threshold allows each arc to contribute equally to the threshold. In our experiments $\alpha = 0.5$ was used, as it minimally impacts the leakage power and provides good runtime savings.

Although a neighbor is ignored, its stale arc delays are still taking part in the local cost. If the local TNS of the examined gate is negative, all neighbors are updated to avoid timing oscillations.

The set of neighboring gates that need local timing update are not statically determined but they are dynamically redefined for each gate in every iteration. The example in Fig. 6 illustrates which gates in the neighborhood of gate 9 are skipped during local timing update as the optimization evolves. In every iteration, the criticality of each arc of the neighboring gates (gates 7, 8, 9, 10, 11) is compared to the threshold in which $a = 1$ for simplicity. During the third iteration,

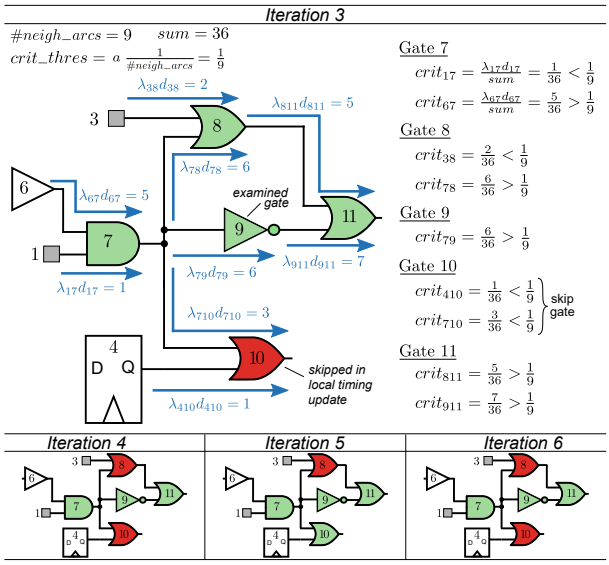


Fig. 6. Since the criticality of timing arcs 7 → 10 and 4 → 10 of gate 10 are less than the threshold, gate 10 can be removed from the neighbors of gate 9 that participate in local timing update. This decision is dynamic and the neighbors skipped in the next iterations is re-evaluated.

the criticality of both arcs of neighbor 10 are less than the threshold and thus gate 10 is skipped. In the next iterations, the same checks are performed for each arc thus possibly skipping gates 8 and 10 in the fourth and the fifth iteration, respectively.

B. Backward pass

Backward pass is responsible for updating the timing criticality of each timing arc of the design by properly updating the values of the LMs. Fig. 7 depicts the task graph of the backward pass for the same running example. The task graph in this case includes one task for each gate, primary output and D-pin of each flip-flop, connected in reverse topological order. The task graph of the backward pass follows the same structure as the second part of the task graph of initial sizing. Therefore, to build the task graph of the backward pass, we can follow the lines 7–22 of Alg. 1 and assign the `backward` function to each task that corresponds to a gate, flip-flop, primary output, or D-pin of a flip-flop.

During the backward pass there are no netlist changes. Thus, we can omit the addition of the MEEs which chain the fanout gates. If they were included, they would degrade the parallel performance of the backward step because they would enforce more serialization on the execution of the tasks.

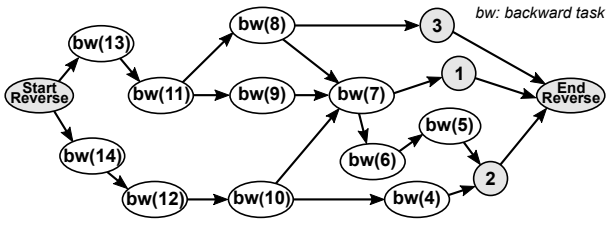


Fig. 7. In the backward graph the tasks are visited in reverse topological order.

The operation of LM update executed in the backward pass is described in Algorithm 6. The LMs of the input arcs are updated to reflect the timing changes due to the re-sizings of the forward pass. The LMs act as penalty factors and their values should reflect if the timing constraints are met or not. The LMs are updated using the approach proposed in [22]. D_{ij} is the worst path delay that passes through timing arc $i \rightarrow j$ and T indicates the clock period target. Therefore, for a timing arc with negative slack $D_{ij} > T$ and thus the LM increases to reflect the violation of the timing constraint. On the other hand, when $D_{ij} < T$ makes $\frac{D_{ij}}{T} < 1$ and decreases λ_{ij}. The exponent K is used to speedup LM increments and reductions and its value is also adopted from [22]. Initially, the design has multiple timing violations and therefore the exponent for the critical arcs is set to 1 and for the non-critical arcs to 0.25. Once, the TNS becomes less than 20% of the clock target and the majority of the timing violations have been resolved, K=4 is used for the positive arcs because the respective LMs need reduction with higher rate in order to save power.

Algorithm 6: backward(gate j)

```
foreach input arc i → j do // LM update
    λ_ij^upd = λ_ij (D_ij / T)^K
end
foreach input arc i → j do // LM scale
    λ_ij = (λ_ij^upd / Σ_{∀m→j} λ_mj^upd) (Σ_{∀j→k} λ_jk)
end
```

Once the LMs of the input arcs are updated, they have to be scaled to respect the KKT condition of (3). The sum of the LMs of the output arcs must be equal to the sum of its input arcs LMs. To achieve this, each input LM gets a percentage of the sum of the output LMs that is proportional to its updated value. For example, for the timing arc 6 $\rightarrow$ 7 of gate 7 (Fig. 2) it is $\lambda_{67} = \left(\lambda_{67}^{upd}/(\lambda_{17}^{upd} + \lambda_{67}^{upd})\right) \cdot (\lambda_{78} + \lambda_{79} + \lambda_{710})$.

VI. Timing and Power Recovery

The recovery step aims at identifying and optimizing the gates that were kept in an un-optimized state after main sizing optimization. For instance, it deals with gates that have negative slack or have remained to a high-leakage size but have positive slack to spend. This recovery step is common in many optimization algorithms and especially in those that rely on Langrangian relaxation [21], [22], [39]. In recovery, a small number of new sizes are tried per gate, each one followed by a complete incremental timing propagation to accurately reflect the timing of the affected paths. To keep the timing picture of the design as accurate as possible after each resizing, state-of-the-art sizers execute this step serially using a single thread that operates at one gate at a time.

In this work, to accelerate the execution of the recovery step, we propose its parallel execution *by allowing multiple gates to be sized in parallel but in a more conservative way*. The task graph used for the recovery step is derived from the task graph of FP after adding extra dependency edges called *timing safety edges* (TSEs) and replacing the function for each task to `recover` implemented in Alg. 8. The addition of TSEs increases timing safety by imposing the recovery task for a gate to execute not only before its fanout but also before the fanout of its side gates. A TSE is added between tasks u and v on top of the task graph of FP when (a) Task u corresponds to a gate that one of its side gates is a fanin of the gate that corresponds to task v and (b) u is visited before v in the forward topological ordering of the netlist thus avoiding any cyclic dependencies. The addition of TSEs on top of the task graph of the forward pass is implemented by Algorithm 7.

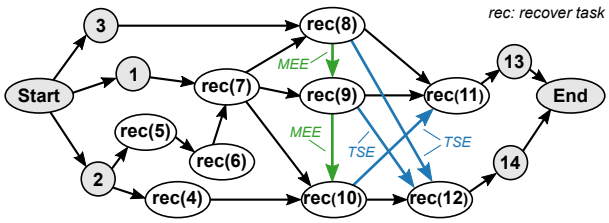


Fig. 8. The recovery task graph is built on top of the task graph of FP. TSE dependencies are added to eliminate timing inaccuracies. The TSE are added from a gate towards the fanouts of its side gates.

Algorithm 7: Add Timing Safety Edges

```
foreach g in {Gates} do
    sides ← {side gates of g};
    sides_f ← {fanouts of sides};
    foreach f of sides_f do
        if g seen before f in topological order and
          dependency g→f doesn't exist then
            task(g).precede(task(f));
        end
    end
end
```

The forward graph for the recovery step of the example in Fig. 2 is depicted in Fig. 8. Gates 8, 9 and 10 share the driver 7 and thus, they are side gates. The task of gate 8 needs to add two TSE dependencies towards the tasks of the fanouts of its side gates 9 and 10, i.e. 11 and 12. But gate 11 is also fanout of gate 8 and therefore a logic-level dependency already exists between them. Therefore, as shown in Fig. 8, only one TSE dependency is added from the task of gate 8 towards 12. Similarly, the TSE dependencies from task 9 towards 12 and from task 10 towards 11 are added. Even though TSEs enforce more serial execution, they are essential for the timing accuracy needed by the resizings of this step. For instance, assume the case of tasks 10 and 11 in Fig. 8. If they were not connected by a TSE it means that they could have executed recovery in parallel. If this was allowed, each task would have affected differently the timing of gate 9 since the latter is a side gate for 10 and a fanin gate for 11. Since the delay of gate 9 affects the local TNS of both gates 10 and 11, resizing them in parallel would have been highly inaccurate for the sensitive recovery step.

Please notice that this inaccuracy in timing does not affect

the convergence of the main optimization loop. The main optimization loop is an iterative process that relies on the minimization of the sum of the arc delays multiplied with their corresponding LMs (cost function (4)). The LMs are updated gradually during every BP and thus keep historic information with respect to the criticality of the corresponding timing arc. For instance, an arc that was critical for multiple iterations it keeps a high value even if its timing is improved. Therefore, the LMs work as safeguard and any wrong decision due to timing inaccuracy, is fixed in the next iteration.

In contrast, in the recovery step the optimization does not depend on a joined product with LMs, but the local decisions are strictly based on the actual slacks as returned from the timer. Therefore, even the small timing inaccuracies which can harm the convergence of the recovery, must be avoided.

Algorithm 8: recover(gate g)

```
if local_TNS(g) > 0 then // Power recovery
    resize g to {next increased V_T size of g's initial size};
    if reject_size(g) then
        resize g to {next smaller size of g's initial size};
        if reject_size(g) then
            resize g to it's initial size;
        end
    end
else // Timing recovery
    resize g to {next bigger size of g's initial size};
    if reject_size(g) then
        resize g to it's initial size;
    end
end
```

The assigned work to the tasks in the recovery step is described in Algorithm 8. For each visited gate, the local TNS is computed and its sign defines whether the gate is sized to save power or to solve timing violations.

For power reduction, only two sizes are tried one after the other; the next V_T and the smaller size. After each resize, timing is updated locally considering all the neighboring gates. The first option that does not violate the design rules and does not degrade the local TNS, is kept. For timing reduction, the gate is sized only to the next bigger size to improve timing. If this size violates the design rule constraints or after a local timing update it worsens the local TNS, the gate is resized back to its initial size.

VII. Experimental Results

The proposed approach was implemented in C++ using the RSyn framework [40] that provides the essential functions for netlist traversal and update as well as timing analysis. The creation and the execution of task graphs was performed with Taskflow [37]. All experiments ran on the same CentOS workstation equiped with 128 GB RAM and two Intel Xeon Silver 4214 @ 2.20GHz CPUs with twelve 2-way multithreaded processors each. In all cases, we used the benchmarks of the ISPD 2013 contest [41] and the final timing results are validated with OpenTimer [14]. The final results obtained from the proposed approach meet all the timing, as well as, the maximum load and slew constraints.

TABLE I
The size of the designs and the properties of the corresponding task graphs (in thousands).

Designs	#Gates	Task Graph Properties	Initial Sizing	Forward Pass	Backward Pass	Recovery
usb_phy	0.6	Nodes	1.4	0.7	0.7	0.7
		Edges	2.5	1.3	1.3	1.3
		MEEs	-	0.4	-	0.4
		TSEs	-	-	-	2.7
pci_bridge32	30.8	Nodes	64.9	34.3	34.3	34.3
		Edges	122.0	60.8	60.8	60.8
		MEEs	-	21.8	-	21.8
		TSEs	-	-	-	245.8
fft	33.8	Nodes	70.5	37.8	37.8	37.8
		Edges	142.1	76.6	76.6	76.6
		MEEs	-	27.1	-	27.1
		TSEs	-	-	-	181.6
cordic	42.9	Nodes	87.1	44.2	44.2	44.2
		Edges	167.3	81.5	81.5	81.5
		MEEs	-	28.9	-	28.9
		TSEs	-	-	-	235.9
des_perf	113.3	Nodes	235.4	122.3	122.3	122.3
		Edges	442.1	215.9	215.9	215.9
		MEEs	-	80.2	-	80.2
		TSEs	-	-	-	604.9
edit_dist	129.2	Nodes	261.6	134.9	134.9	134.9
		Edges	506.9	252.6	252.6	252.6
		MEEs	-	95.0	-	95.0
		TSEs	-	-	-	732.3
matrix_mult	159.6	Nodes	320.6	164.1	164.1	164.1
		Edges	620.9	301.2	301.2	301.2
		MEEs	-	110.9	-	110.9
		TSEs	-	-	-	775.6
netcard	984.1	Nodes	2064.2	1081.9	1081.9	1081.9
		Edges	3952.5	1987.9	1987.9	1987.9
		MEEs	-	761.9	-	761.9
		TSEs	-	-	-	10785.5

A. The characteristics of the tasks graphs

Before comparing the proposed approach to the state-of-the-art, it would be useful to examine the characteristics of the ISPD13 benchmarks used in the evaluation and how they affect the size and structure of the corresponding task graphs. The characteristics of the task graph depend solely on the structural properties of each design and not on the timing constraints ('slow' or 'fast') associated with each benchmark.

Table I presents the number of gates of each design together with the properties of each task graph in each case. The number of nodes for all types of tasks graphs are linearly dependent on the number of gates and flip-flops of the design as well as on the number of primary inputs and outputs. The number of simple edges follows the connectivity of the netlist of each design being linearly dependent on the number of design's nets. On the contrary, the MEEs and TSEs added in the fordward pass and in recovery, respectively, depend on the fanout of certain nets of the design. The higher the fanout per net, the more the MEEs and TSEs added.

TSEs are used to maximize timing safety by imposing a more restrictive execution order to certain tasks of the recovery phase. In all cases, the number of TSEs exceed by far the number of simple edges and MEEs. We expect this

TABLE II
THE LEAKAGE POWER AND RUNTIME OF [21] COMPARED TO THE PROPOSED TASK-BASED GATE SIZING.

Design	Leakage Power (W)		Runtime (min)		Speedup
	[21]	Ours 1 thread	[21]	Ours 1 thread	
usb_phy_slow	0.001	0.001	0.49	0.04	12.25
usb_phy_fast	0.002	0.002	0.42	0.09	4.67
pci_bridge32_slow	0.057	0.058	10.53	3.39	3.11
pci_bridge32_fast	0.085	0.096	22.62	7.06	3.20
fft_slow	0.087	0.088	25.71	6.74	3.81
fft_fast	0.194	0.214	40.43	12.50	3.23
cordic_slow	0.271	0.292	69.04	19.70	3.50
cordic_fast	1.001	1.013	117.08	30.14	3.88
des_perf_slow	0.330	0.342	132.27	23.20	5.70
des_perf_fast	0.649	0.654	347.87	35.94	9.68
edit_dist_slow	0.425	0.451	123.90	21.60	5.74
edit_dist_fast	0.540	0.571	352.96	36.34	9.71
matrix_mult_slow	0.444	0.469	226.13	38.04	5.94
matrix_mult_fast	1.611	1.673	395.96	60.54	6.54
netcard_slow	5.155	5.156	483.55	106.22	4.55
netcard_fast	5.200	5.202	400.89	148.56	2.70
Total	16.050	16.280	2749.85	550.10	-
Geomean	0.220	0.230	57.53	11.57	4.97

characteristic to translate to less structural parallelism in the recovery task graph relative to the task graph of the forward pass.

B. Comparison with state-of-the-art

Initially, we would like to compare the proposed task-based approach with state-of-the-art gate sizers [21] and [22]. The leakage power of state-of-the-art methods are the final results reported in [21], [22] after finishing both main sizing optimization and their corresponding timing/power recovery steps. The runtimes reported [21], [22] are taken verbatim from the respective papers.

The runtime results of [21] correspond to single-thread implementations run on an Intel i7-3770 @ 3.40GH, while the runtime results of [22] refer to a mixed multi-threaded and single-thread implementation executed on a system with two quad-core Intel Xeon E3-1240 v5 @ 3.50GHz CPUs and 16GBs of memory. In [22] the main optimization step was executed using 8-threads described with OpenMP and the initial sizing as well as the final recovery step ran serially using only one thread. In both [21], [22], the final recovery step is executed on purpose on a single thread to guarantee the maximum timing accuracy.

Table II highlights the performance of [21] relative to the baseline task-based formulation of the gate sizing problem, without enabling RTS or FLTU that dynamically reduce sizing alternatives and speedup local timing updates. All steps of the proposed gate sizer were also executed on a single thread. The proposed flow achieves similar leakage power results and significant runtime savings when compared to [21]. For instance, the single-threaded execution of the proposed approach achieves 4.97× speedup at the cost of 5% higher leakage power as reported by the geometric mean average of leakage power and speedup per benchmark, respectively. Geometric mean average is used to facilitate data averaging with a wide range in values. The reduced execution time of the proposed approach is a result of the smaller number of iterations of the main optimization step and the replacement of the full-incremental timing update in the recovery step of [21] with a local timing update.

Similarly, Table III compares the proposed approach relative to the results reported in [22] for an eight-thread execution. For the proposed sizer we consider the baseline approach that allows each task to examine all possible gate sizes and the one that enables RTS. RTS alters dynamically the number of sizes that are tried for each gate based on the gate's timing slack. Restricting the search space may slow down the convergence of the main optimization since more iterations are needed to find the most suitable size. However, each iteration is faster.

The leakage power achieved in each case is depicted in columns 2–4 of Table III. The two variants of the proposed approach have less than 4% higher leakage power on average than [22]. As expected, the solutions with RTS enabled have higher leakage relative to the baseline approach. This holds for all benchmarks except 'netcard'. In this case, RTS performs marginally better. This result is an artifact caused by multithreading. Every time we run the same experiment with the same input, the final results can be slightly different. Nevertheless, the difference observed in all cases is always at the granularity of the third decimal digit.

The runtime of [22] is shown in columns 5–6 of Table III. Column 'Orig.' refers to the total runtime reported in [22] for each benchmark. This runtime involves also the time needed for timing-model calibration that is done using an external timer. To have a fair comparison the overhead of communicating with the external timer should be removed from the comparisons. According to [22] the useful runtime for timing calibrations that does not involve TCL and file processing is roughly 20% of the total runtime spent for timing calibrations. To compute the time that should be removed for each benchmark, we used the contribution in runtime of timing calibrations reported in Table V of [22]. The trimmed runtime derived for each benchmark is shown in column 'Trimmed'. In benchmarks that timing calibration was a large part of the overall runtime, the runtime reductions are significant. For instance, for netcard_fast the total runtime reduced from 30.6 to 18.3 minutes.

The runtime of the two variants of the proposed method for 8 and 24 threads and the speedup achieved relative to the trimmed runtime of [22] are depicted in the last six columns of Table III. The baseline version of the proposed approach shows a marginal reduction in the total runtime needed to execute all benchmarks using 8 threads but improves significantly for 24 threads.

The work of [22], after the first five iterations of the LR sizing loop, examines a subset of the available sizes for each gate. This has a significant impact on the overall runtime. Therefore, to have a *fair* comparison with the proposed work we need to compare the trimmed runtimes of [22] with the runtimes achieved by the proposed method with the RTS heuristic enabled. Under this *apple-to-apple* comparison, i.e., both approaches employ multithreading under an equal number of threads and both use a heuristic that examines only a subset of the available gate sizes, the proposed method

TABLE III
THE LEAKAGE POWER AND RUNTIME OF [22] COMPARED TO THE PROPOSED TASK-BASED GATE SIZING.

Designs	Leakage Power (W)			Runtime (min)							
	[22]	Ours		[22]		Ours - Base		Ours with RTS enabled			
		Base	w. RTS	Orig.	Trimmed	8 thr.	24 thr.	8 thr.	Speedup vs Trimmed	24 thr.	Speedup vs Trimmed
usb_phy_slow	0.001	0.001	0.001	0.22	0.05	0.01	0.01	0.01	5.00	0.01	5.00
usb_phy_fast	0.002	0.002	0.002	0.23	0.06	0.02	0.05	0.01	6.00	0.01	6.00
pci_bridge32_slow	0.058	0.058	0.058	0.97	0.40	0.55	0.27	0.37	1.08	0.22	1.82
pci_bridge32_fast	0.090	0.095	0.100	1.54	0.97	1.21	0.53	0.79	1.23	0.40	2.43
fft_slow	0.088	0.087	0.089	1.37	0.73	1.20	0.59	0.73	1.00	0.42	1.74
fft_fast	0.213	0.219	0.230	1.64	1.01	1.51	0.92	1.06	0.95	0.70	1.44
cordic_slow	0.293	0.299	0.338	2.29	1.56	2.62	1.48	1.62	0.96	1.02	1.53
cordic_fast	1.080	1.025	1.100	5.60	4.88	4.77	2.21	2.92	1.67	1.46	3.34
des_perf_slow	0.332	0.339	0.348	7.27	5.82	3.59	1.73	2.56	2.27	1.37	4.25
des_perf_fast	0.639	0.653	0.657	26.16	24.70	5.83	2.33	4.31	5.73	1.71	14.44
edit_dist_slow	0.440	0.451	0.453	4.92	3.15	3.49	1.74	2.24	1.41	1.30	2.42
edit_dist_fast	0.549	0.573	0.587	6.66	4.90	5.99	2.73	4.07	1.20	2.23	2.20
matrix_mult_slow	0.448	0.468	0.475	8.80	6.76	7.76	3.26	4.51	1.50	2.64	2.56
matrix_mult_fast	1.633	1.672	1.680	13.94	11.82	10.97	5.10	6.12	1.93	2.83	4.18
netcard_slow	5.170	5.156	5.155	24.67	12.43	18.91	9.78	11.29	1.10	7.74	1.61
netcard_fast	5.205	5.202	5.200	30.60	18.36	24.81	12.10	13.79	1.33	8.93	2.06
Total	16.241	16.300	16.473	136.88	97.60	93.24	44.83	56.40	-	32.99	-
Geomean	0.230	0.233	0.239	3.70	2.18	1.97	1.10	1.27	1.72	0.76	2.86

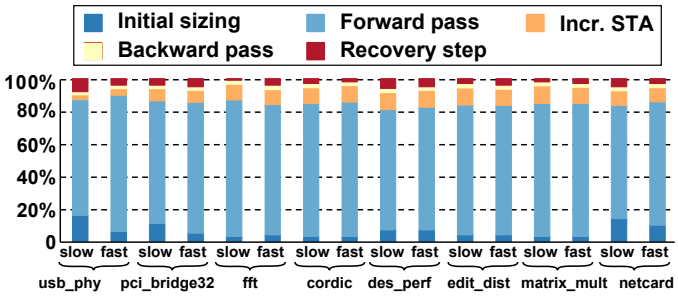


Fig. 9. The breakdown of the total runtime for all benchmarks. The parallel initial sizing uses the 6% on average of the total runtime. The majority of the time is consumed in the forward pass of the main optimization step and only 9% in the single-threaded incremental timing analysis. The backward pass needs 2% on average while the recovery step utilizes the 4% on average.

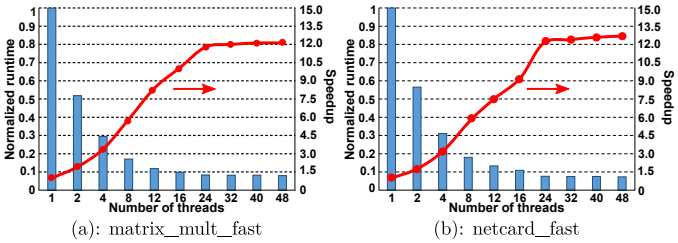


Fig. 10. The normalized runtime and speedup of the proposed approach for increasing number of threads for the two largest benchmarks. The runtime decreases sufficiently until thread count reaches 24 that matches the amount of physical CPUs used in this experiment.

achieves a mean speedup improvement of 1.72× for eight threads that improves to 2.86× for 24 threads.

The runtime reduction reported is the combined result of two factors. First, the proposed approach executes in parallel all steps of the optimization including initial sizing, main optimization and final recovery while the method of [22] performs single-threaded initial sizing and recovery. Also, our thread scheduling is not performed manually, as in [22], but done automatically by Taskflow that allows scaling the gate sizer smoothly to a higher number of threads.

Fig. 9 highlights the contribution in runtime of each step of the proposed task-based parallel gate sizer assuming eight available threads. The FP of the main optimization loop utilizes on overage the 79% of the total runtime while BP takes only 2% of the total runtime. Initial sizing consumes 6% of the total runtime on average, while the parallel implementation of the iterative recovery step reduced its runtime contribution to only 4% on average. Incremental timing analysis accounts for 9% of the total execution time, on average. In our implementation the incremental timing analysis is performed using only one thread. Therefore, a multi-threaded implementation of the timing analysis can further reduce the amount of time consumed by this step.

The scalability of the proposed approach with increasing the number of threads for the two largest designs matrix_mult_fast and netcard_fast is shown in Fig. 10. The execution time is normalized to the runtime of the single-thread run. Moving from one to two threads speeds the execution of matrix_mult_fast by 1.9× and 1.7× for netcard_fast. Enabling four threads gives an additional speedup of 70% on average for both designs. Beyond 24 threads, there is little reduction in runtime. This is caused by the dependencies of the task graphs that limit the parallelism that can be achieved and, at the same time, with 24 threads we reach the maximum number of physical CPUs.

C. Highlighting the contribution of RTS and FLTU

The results of Table III have shown the effectiveness of RTS in reducing the overall runtime. In this section we want to clarify the behavior of RTS and also highlight the contribution of FLTU that reduces dynamically the number of timing arcs included at each local timing update.

TABLE IV
THE LEAKAGE POWER AND THE RUNTIME OF THE PROPOSED FLOW WITH ONLY RTS (RTS) AND WITH RTS AND FLTU (RTS & FLTU) ENABLED.

Design	Leakage Power (W)		Total Runtime (min)		Speedup
	RTS	RTS & FLTU	RTS	RTS & FLTU	
usb_phy_slow	0.001	0.001	0.01	0.01	1.00
usb_phy_fast	0.002	0.002	0.01	0.01	1.00
pci_bridge32_slow	0.058	0.058	0.22	0.19	1.16
pci_bridge32_fast	0.100	0.107	0.40	0.36	1.11
fft_slow	0.089	0.089	0.42	0.37	1.14
fft_fast	0.230	0.235	0.70	0.62	1.13
cordic_slow	0.338	0.339	1.02	0.83	1.23
cordic_fast	1.100	1.121	1.46	1.29	1.13
des_perf_slow	0.348	0.349	1.37	1.24	1.10
des_perf_fast	0.657	0.662	1.71	1.39	1.23
edit_dist_slow	0.453	0.455	1.30	1.23	1.06
edit_dist_fast	0.587	0.593	2.23	1.94	1.15
matrix_mult_slow	0.475	0.480	2.64	2.17	1.22
matrix_mult_fast	1.680	1.684	2.83	2.41	1.17
netcard_slow	5.155	5.156	7.74	7.25	1.07
netcard_fast	5.200	5.202	8.93	8.27	1.08
Total	16.473	16.533	32.99	29.58	-
Geomean	0.239	0.241	0.76	0.68	1.12

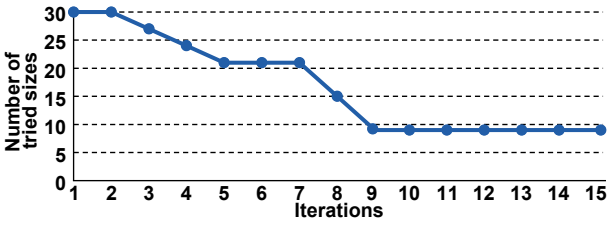


Fig. 11. The number of sizes that are examined in gate sizing as the optimization evolves for a specific gate of matrix_mult_fast design. At first, the gate has large negative slack and thus all the available sizes are tried. As slack improves the number of examined sizes is reduced to minimum.

To clarify the dynamic behavior of RTS, Fig. 11 depicts the total number of sizes that are tried in each iteration for a specific gate in matrix_mult_fast benchmark. In the first iteration, the gate is part of the most critical path and therefore all the available sizes are examined. In the next iteration, even though the gates slack and WNS are improved, the gate remains in the critical path and thus again all sizes are examined. As the optimization evolves the number of examined sizes decreases because the ratio of the gates slack to WNS is getting smaller. Once the gate obtains positive slack (iteration 9) the examined sizes are fixed to nine options: three sizes per V_T for three available V_T.

The combined effect of RTS and FLTU relative to gate sizing with only RTS enabled is shown in Table IV. In all cases, the task graphs are executed using 24 threads. Enabling FLTU in addition to RTS offers an additional 1.12× speedup on average due to the simplification of the local timing updates at the cost of less than 1% higher leakage power.

To understand better the application of FLTU, we show in Fig. 12 the average percentage of the neighbors that are updated during the local timing update at each iteration. The design used in this example is edit_dist_slow. Initially, almost all neighbors are updated because the design starts with many timing violations distributed across all paths. During the next

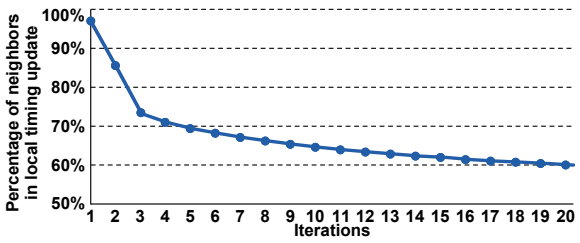


Fig. 12. The average percentage of the neighbors which are updated when fast local timing update is enabled in edit_dist_slow. Initially, all neighbors are considered for update but during the next two iterations, the number of neighbors considered is significantly reduced.

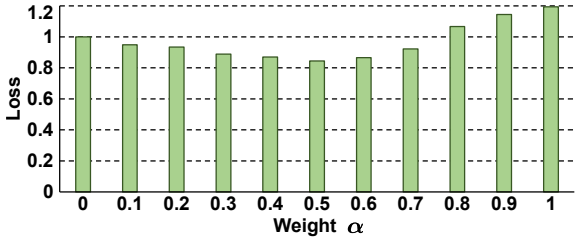


Fig. 13. The loss for different values of weight α in the criticality threshold in edit_dist_slow. The minimum loss observed in 0.5. Beyond $\alpha = 0.8$ the neighbors which are skipped affect the timing accuracy significantly and therefore more iterations are needed to converge.

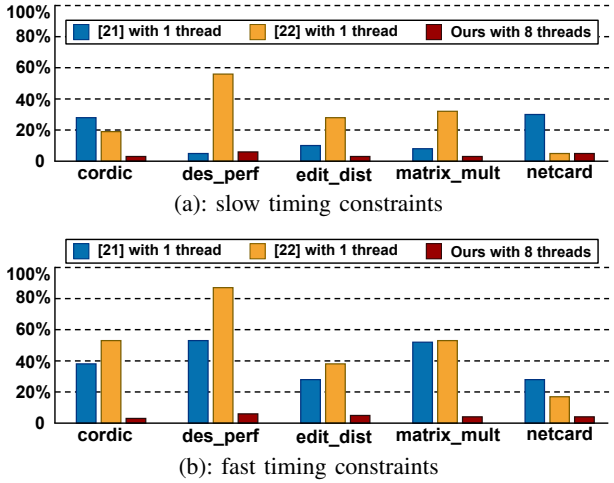
two iterations, the timing improvement is sufficient and thus there are many neighbors for which the timing update locally is not essential. More specifically, in the third iteration only the 72% of the neighbors are considered for local timing update. From this point forward more and more neighbor timing arcs stop contributing to the local cost and therefore can be ignored. Similar results are obtained for all benchmarks.

We should not forget that excluding timing arcs from the local timing update of each gate inevitably keeps their delay unchanged. When those timing arcs participate in the local cost function with their not-updated (stale) delays may lead to sub-optimal local gate sizing choices. The effect of such choices, translates to designs with increased leakage power (2% more leakage is observed relative to the baseline approach).

The criticality threshold is fundamental in the FLTU because it defines how many and which of the neighbors are skipped from the timing update and therefore can affect the overall QoR. For example, a low value for the threshold implies that the majority of the neighbors are updated, while a high value reduces the neighboring gates to update. As the number of skipped neighbors increases, the runtime reduces but the leakage is degraded. For this reason, to better understand how the criticality threshold affects the overall QoR, we modify its value by increasing the weight α that is part of the threshold. To better evaluate the obtained final results, a loss function is defined which takes into account both the runtime and the power changes. More specifically, the loss for a specific α, $Loss_\alpha$, should be considered minimal if the leakage is not appreciably increased and the runtime is significantly reduced when compared to the corresponding results obtained with

TABLE V
LEAKAGE POWER BEFORE RECOVERY AND THE INCREMENTAL CHANGE OF POWER (ΔPOWER) AFTER RECOVERY.

Design	Leakage Power (W) before recovery			ΔPower (%) after recovery		
	[21]	[22]	Ours	[21]	[22]	Ours
usb_phy_slow	**0.001**	**0.001**	**0.001**	0.0	0.0	0.0
usb_phy_fast	**0.002**	**0.002**	**0.002**	0.0	0.0	0.0
pci_bridge32_slow	**0.057**	**0.057**	0.058	0.0	1.8	0.0
pci_bridge32_fast	**0.088**	**0.088**	0.093	-3.4	2.3	2.2
fft_slow	**0.087**	0.088	0.089	0.0	0.0	-2.2
fft_fast	**0.204**	0.209	0.222	-4.9	1.9	-1.4
cordic_slow	0.309	**0.296**	0.303	-12.3	-1.0	-1.3
cordic_fast	1.665	**1.273**	1.289	-39.9	-15.2	-20.5
des_perf_slow	0.339	**0.328**	0.336	-2.7	1.2	0.9
des_perf_fast	0.750	**0.648**	0.668	-13.5	-1.4	-2.2
edit_dist_slow	**0.429**	0.439	0.454	-0.9	0.2	-0.7
edit_dist_fast	0.573	**0.551**	0.572	-5.8	-0.4	0.2
matrix_mult_slow	0.463	**0.454**	0.485	-4.1	-1.3	-3.5
matrix_mult_fast	2.032	**1.859**	1.894	-20.7	-12.2	-11.7
netcard_slow	**5.117**	5.169	5.156	0.7	0.0	0.0
netcard_fast	**5.148**	5.195	5.205	1.0	0.2	-0.1



(a): slow timing constraints

(b): fast timing constraints

Fig. 14. The percentage of the total runtime utilized by recovery in the five largest designs under (a) slow and (b) fast timing constraints. Both single-threaded recovery steps utilize significant part of the total runtime when compared to the proposed method.

$a = 0$ i.e. none of the neighbors is skipped.

$$Loss_\alpha = \frac{leakage_\alpha}{leakage_0} \cdot \frac{runtime_\alpha}{runtime_0}$$

The overall loss for increasing the value of weight α in edit_dist_slow is illustrated in Fig. 13. As the weight α increases, the threshold also increases and therefore less neighbors are considered for the timing update. Until $\alpha = 0.7$, the overall loss is lower than with $\alpha = 0$, because the runtime is reduced and simultaneously the leakage is marginally increased. For $\alpha \geq 0.8$ more neighbors are skipped from the timing update and thus even less loss is expected. However, beyond this point, the increased number of skipped neighbors affects negatively the timing accuracy. This leads to sub-optimal solutions with higher leakage power and that causes the gate sizing to run for more iterations. The same behavior is observed in all designs. Therefore, in our experiments $\alpha = 0.5$ was used, as it minimally impacts the leakage power and provides good runtime savings.

D. The contribution of final timing and power recovery

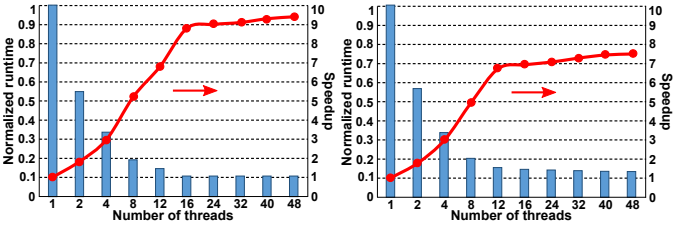
All three methods under comparison are using a Lagrangian Relaxation based formulation for the main optimization step that is accompanied by a timing and power recovery step at the end. This step is crucial in correcting the remaining small timing violations and recovering part of the excessive leakage power. To enable surgical-accuracy resizing decisions, the methods of [21], [22] require examinining serially one gate at a time (from a limited set of gates) and performing a full incremental timing update after each change. This requirement increases inevitably the runtime of the recovery step and limits the applicability of LR-based gate sizers. This limitation is effectively removed by the proposed approach.

Table V reports in columns 2–4 the leakage power at the end of the main optimization step for all methods under comparison irrespective of the runtime needed to finish the main optimization. The results show that more or less all three methods converge to similar leakage power. If we observe the results more carefully, we see that the method of [21] or the method of [22] achieve the lowest leakage power after main optimization. Therefore, the proposed approach has to recover slightly more leakage power than the rest.

Also, columns 5–7 of Table V depict the reduction in leakage power achieved after recovery (timing is closed in all cases). Even if each method behaves differently during recovery, the overall trend per design remains the same for all methods. For instance, in netcard and edit_dist the recovery step fails to reduce the leakage power. On the contrary, the reductions observed in cordic and matrix_mult are significant.

Even in cases that didn't benefit a lot from timing and power recovery, the runtime spent is not negligible. Fig. 14 illustrates the percentage of the total runtime consumed by the recovery step in the five largest designs. For each design, we include the percentage of the single-threaded recovery step of [21], [22] and the percentage of the proposed *conservatively-parallel* recovery step executed using 8 threads. The recovery step in [21] and [22] accounts for the 18% and 27% of the total runtime on average, respectively. There are cases, such as des_perf_fast or matrix_mult_fast for [22], where "the last mile" optimization represents more than the 50% of the total runtime. On the contrary, the iterative recovery of the proposed approach performs on average 5 iterations and takes 4% of the total runtime. This result stems from the task-based execution of the recovery step and the extra TSE edges added to the task graph that allow for resizing multiple gates in parallel and preserving the timing accuracy needed in this step.

The scalability of the task-based recovery step with respect to number of active threads is depicted in Fig. 15 for matrix_mult_fast and netcard_fast designs. Initially, the speed up of matrix_mult_fast scales sufficiently until 16 threads. Then, the improvement stops. In the main optimization task graph this behavior is observed at 24 threads. The reason is that the recovery is executed on a more constrained task graph with 3× more dependencies (due to the TSE edges) that limit inevitably the available parallelism but ensure better timing



(a): matrix_mult_fast (b): netcard_fast

Fig. 15. The normalized runtime and speedup of the proposed recovery step for the largest benchmarks using different number of threads. TSEs limit the improvement of speedup at 16 and 12 threads for (a) matrix_mult_fast and (b) netcard_fast, respectively.

accuracy after each local timing update. Similarly, at first the runtime of netcard_fast (Fig. 15(b)) starts improving until 12 threads where speedup levels off for the same reason. The recovery graph of netcard_fast contains 5× more dependencies and therefore speedup saturates earlier than matrix_mult_fast.

E. Recovery with Composite Tasks

One extra choice that would possibly improve the runtime behavior of the recovery step is to allow the resizing of multiple gates in the same trial. To enable this feature, we should group multiple gates in the same composite task and adjust appropriately the dependency edges of the restructured recovery task graph. Then, inside each composite task all choices are examined allowing two or more gates to change size in the same trial.

To test this feature, we replaced pairs of tasks of the original task graph with composite tasks of two gates. Two tasks are merged when they both represent timing-critical gates (with negative slack at their outputs) or gates with enough positive slack (above 50ps) that can be grouped for power recovery. The pairs of tasks that are grouped should refer to directly connected gates in the netlist and their replacement by a composite task in the recovery task graph should not create any cyclic dependency. This is achieved by performing a depth-first search from the composite task after each merge. If there is a cycle, the merge it is undone, otherwise it is kept.

Inside each composite task, the possible solutions of both gates are enumerated and visited in ascending order of their cumulative power. The first solution that improves local TNS (for timing recovery) or does not degrade local TNS (for power reduction) is selected. The available sizes per gate are the same as the baseline recovery Algorithm 8. However, by allowing the resizing of multiple gates at once, the number of choices increases exponentially.

The obtained results using 24 threads are depicted in Fig. 16. The runtime of the recovery improves in all examined cases by 10% on average without affecting negatively the final leakage power of the design. The number of composite tasks depends on the timing slack of each gate and ranges between 10k and 24k in the examined designs. In other words, composite tasks are between 2.5% and 15% of the total number of tasks.

This feature, even if it seems a promising solution, in its current form offers negligible improvements in the total runtime of gate sizing. Future research work will highlight

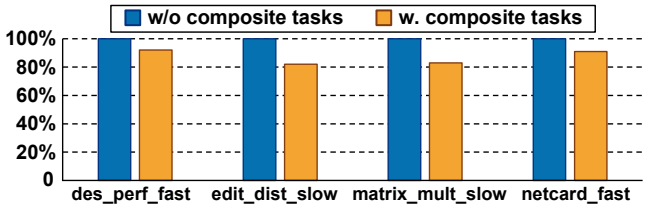


Fig. 16. The normalized runtime of the proposed recovery without and with composite tasks for the largest benchmarks at 24 threads.

what is the best approach to group gates in composite tasks, how to examine fast the increased options per composite task, and how to quantify how the formation of composite tasks affects the structural parallelism of the baseline recovery task graph.

VIII. Conclusions

Expressing all parts of timing and power optimization, even those that are traditionally considered as serial operations, as a task-based parallel program allows for scalable runtime improvements and maximum performance portability. The development of the task-parallel gate sizer included the selection of the appropriate optimization kernels for each part of the sizing process and the definition of the dependencies between them. No effort was spent on managing thread execution. Execution order was constrained by the introduced dependencies and handled automatically by Taskflow.

Choosing appropriate dependencies between gate resizing tasks enables their execution with varying levels of timing accuracy. Iterative optimization steps at the beginning of the design flow can operate with relaxed timing accuracy and enjoy many iterations with fast runtime. On the other hand, final recovery steps of gate sizing or when gate sizing is applied at the end of the physical design flow need higher timing accuracy to still enjoy parallel execution but also not compromise the already achieved QoR.

Additional runtime was saved by speeding up the execution of each task. This was achieved by enabling the two aditional heuristics proposed in this work that dynamically alter the number of examined sizes per gate, or reduce the neighbor gates that participate in local timing update.

References

[1] L. Lavagno, G. Martin, I. L. Markov, and L. K. Scheffer, *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*. Taylor and Francis group, 2016.

[2] L. C. Lu, "Physical design challenges and innovations to meet power, speed, and area scaling trend," Intern. Symp. Phys. Des. (ISPD) 2017.

[3] N. D. MacDonald, "Timing closure in deep submicron designs," in *Design Automation Conference (DAC)*, 2010.

[4] O. Aumage, P. Carpenter, and S. Benkner, "Task-based performance portability in HPC," Oct. ETP4HPC whitepaper, 2021.

[5] A. D. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning, 2012.

[6] J. Reinders, *Intel threading building blocks*. OReilly Media, 2007.

[7] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Comp. Science and Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[8] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Symp. on Operating Systems Principles*, 2013, pp. 456–471.

[9] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Trans. on Parallel and Distributed Systems*, vol. 33, pp. 1303–1320, 2022.

[10] D. Mangiras, A. Stefanidis, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos, "Timing-Driven Placement Optimization Facilitated by Timing-Compatibility Flip-Flop Clustering," *in IEEE Trans. on CAD*, vol. 39, no. 10, pp. 2835 – 2848, Oct. 2020.

[11] A. Stefanidis, D. Mangiras, C. Nicopoulos, D. Chinnery, and G. Dimitrakopoulos, "Design Optimization by Fine-Grained Interleaving of Local Netlist Transformations in Lagrangian Relaxation," in *Intern. Symp. on Physical Design (ISPD)*, 2020, pp. 87–94.

[12] D. Mangiras and G. Dimitrakopoulos, "Incremental lagrangian relaxation based discrete gate sizing and threshold voltage assignment," *Technologies*, vol. 9, no. 4, 2021.

[13] M. M. Ozdal, S. Burns, and J. Hu, "Algorithms for gate sizing and device parameter selection for high-performance designs," *IEEE Trans. on CAD*, vol. 31, no. 10, pp. 1558–1571, October 2012.

[14] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A new parallel incremental timing analysis engine," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 40, no. 4, pp. 776–789, 2021.

[15] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCDPlace: accelerated batch-based concurrent detailed placement on multithreaded CPUs and GPUs," *IEEE Trans. on CAD*, vol. 39, no. 12, pp. 5083–5096, 2020.

[16] O. Coudert, "Gate Sizing for Constrained Delay/Power/Area Optimization," *IEEE Trans. on VLSI Systems*, vol. 5, no. 4, pp. 465–472, 1997.

[17] S. Hu, M. Ketkar, and J. Hu, "Gate sizing for cell library-based designs," in *Design Automation Conference (DAC)*, June 2007, pp. 847–852.

[18] D. Nguyen and et al., "Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization," in *Int. Symp. Low Power Electronics and Design*, 2003, pp. 158–163.

[19] D. G. Chinnery and K. Keutzer, "Linear programming for sizing, Vth and Vdd assignment," in *Proc. of the Intern. Symp. on Low Power Electronics and Design (ISLPED)*, 2005, pp. 149–154.

[20] S. Held, "Gate sizing for large cell-based designs," in *Proceedings of the Conf. on Design, Automation and Test in Europe*, 2009, pp. 827–832.

[21] G. Flach and et al., "Effective method for simultaneous gate sizing and vth assignment using lagrangian relaxation," *IEEE Trans. on CAD*, vol. 33, no. 4, pp. 546–557, April 2014.

[22] A. Sharma, D. Chinnery, T. Reimann, S. Bhardwaj, and C. Chu, "Fast Lagrangian Relaxation Based Multi-Threaded Gate Sizing Using Simple Timing Calibrations," *IEEE Trans. on CAD*, vol. 39, no. 7, pp. 1456–1469, 2019.

[23] G. Shklover and B. Emanuel, "Simultaneous Clock and Data Gate Sizing Algorithm with Common Global Objective," in *Int. Symp. Physical Design*, 2012, pp. 145–152.

[24] J. Hu, A. B. Kahng, S. Kang, M.-C. Kim, and I. L. Markov, "Sensitivity-guided metaheuristics for accurate discrete gate sizing," in *Int. Conf. on CAD (ICCAD)*, 2012, pp. 233–239.

[25] A. B. Kahng, S. Kang, H. Lee, I. L. Markov, and P. Thapar, "High-performance gate sizing with a signoff timer," in *Int. Conf. on CAD (ICCAD)*, 2013, pp. 450–457.

[26] D. Chinnery and A. Sharma, "Integrating LR gate sizing in an industrial place-and-route flow," in *Int. Symp. on Physical Design (ISPD)*, 2022, p. 3948.

[27] L. Li, P. Kang, Y. Lu, and H. Zhou, "An efficient algorithm for library-based cell-type selection in high-performance," in *2012 IEEE/ACM Intern. Conf. on Computer-Aided Design (ICCAD)*, pp. 226–232.

[28] S. Roy, D. Liu, J. Singh, J. Um, and D. Z. Pan, "OSFA: A new paradigm of aging aware gate-sizing for power/performance optimizations under multiple operating conditions," *IEEE Trans. on CAD*, vol. 35, pp. 1618–1629, Oct 2016.

[29] Y. O. M. Moctar and P. Brisk, "Parallel FPGA routing based on the operator formulation," in *Design Automation Conf. (DAC)*, 2014.

[30] J. He, M. Burtscher, R. Manohar, and K. Pingali, "SPRoute: A scalable parallel negotiation-based global router," in *Intern. Conf. on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[31] Y. Liu and J. Hu, "GPU-based parallelization for fast circuit optimization," vol. 16, no. 3, June 2011.

[32] Y. Liu and J. Hu, "A new algorithm for simultaneous gate sizing and threshold voltage assignment," *IEEE Trans. on CAD*, vol. 29, no. 2, pp. 223–234, 2010.

[33] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *Int. Conf. on CAD (ICCAD)*, 2020, pp. 1–9.

[34] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated critical path generation with path constraints," in *Int. Conf. on CAD (ICCAD)*, 2021, pp. 1–9.

[35] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated path-based timing analysis," in *Design Automation Conference (DAC)*, 2021, pp. 721–726.

[36] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Design Automation Conf. (DAC)*, 2019.

[37] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-taskflow: A general-purpose parallel task programming system at scale," *IEEE Trans. on CAD*, vol. 40, no. 8, pp. 1687–1700, 2021.

[38] 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 CAD*, vol. 18, no. 7, pp. 1014–1025, July 1999.

[39] A. Stefanidis, D. Mangiras, C. Nicopoulos, D. Chinnery, and G. Dimitrakopoulos, "Autonomous application of netlist transformations inside Lagrangian Relaxation-based optimization," *IEEE Trans. on CAD*, vol. 40, no. 8, pp. 1672–1686, August 2021.

[40] G. Flach, M. Fogaça, J. Monteiro, M. Johann, and R. Reis, "Rsyn: An extensible physical synthesis framework," in *Int. Symp. on Physical Design*, 2017, pp. 33–40.

[41] M. Ozdal, C. Amin, A. Ayupov, S. M. Burns, G. R. Wilke, and C. Zhuo, "An improved benchmark suite for the ISPD-2013 discrete cell sizing contest," in *Int. Symp. on Physical Design*, 2013, p. 168170.

Dimitrios Mangiras Dimitrios Mangiras received the Diploma in electrical and computer engineering from the Democritus University of Thrace, Xanthi, Greece, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include electronic design automation for physical design, clock tree synthesis and machine-learning based optimization as well as, design of energy-efficient integrated circuits and automated verification methodologies.

Mr. Mangiras received the Best Student Paper Award at the International Conference on Modern Circuits and Systems Technologies (MOCAST) in 2021.

David Chinnery David Chinnery received a Ph.D. in Electrical Engineering from the University of California at Berkeley in 2006. He is the author of two books on Closing the Gap Between ASIC and Custom with tools and techniques for high-performance and low-power design. He is an author of two chapters in the EDA for Integrated Circuits Handbook, and various conference and journal papers.

David has worked for the past ten years at Mentor Graphics, which is now part of Siemens Digital Industries Software. From 2017, he was the R&D manager for the optimization R&D group of the Nitro place-and-route tool. Since 2021, he has been a member of the Aprisa place-and-route optimization R&D team. Previously, David worked for five years in the CAD group at Advanced Micro Devices supporting both custom and synthesized microprocessor designs.

Giorgos Dimitrakopoulos Giorgos Dimitrakopoulos received the B.S., M.Sc., and Ph.D. degrees in computer engineering from the University of Patras, Patras, Greece, in 2001, 2003, and 2007, respectively.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece. He is interested in the design of digital integrated circuits, energy-efficient data-parallel accelerators, functional safety architectures, and the use of high-level synthesis for agile ASIC and FPGA design flows.

He received two Best Paper Awards at the Design Automation and Test in Europe (DATE) Conference in 2015 and 2019, respectively. Also, he received the HIPEAC Technology Transfer Award in 2015.