ParallelClosure: A Parallel Design Optimizer for Timing Closure

Yi-Shan Lu*, Wenmian Hua†, Rajit Manohar†, Keshav Pingali*

*University of Texas at Austin
{yishanlu, pingali}@cs.utexas.edu

†Yale University
{wenmian.hua, rajit.manohar}@yale.edu

***Abstract*—Timing-driven optimization is essential for the success of closure flows. In this contest, we explore existing efficient buffer insertion and gate sizing algorithms to fix timing violations; and minimize area, leakage power, and clock period for a design without altering its functionality. Using the Galois infrastructure for parallelization, we prototype a fast timing-driven design optimizer, ParallelClosure, to effectively and efficiently optimize designs with a small number of timing violations.**

I. Introduction

Timing-driven optimization is imperative for the success of closure flows, and it is essential to find efficient ways to make changes to the design to satisfy timing constraints and optimize power, performance and area. In this contest, given a Verilog netlist, a SPEF file for net delay models, a set of cell libraries (corners) and an SDC file, we are going to generate an optimized design in Verilog and SPEF with (1) no violations of hold time and maximum slew transition for all given corners, and (2) minimized area, leakage power and clock period. Note that the optimization should consider multi-corner multi-mode (MCMM) scenarios.

Table I summarizes the optimizations allowed in this contest. For better control of the design optimization, we do not consider buffer deletion, as its effect to the design is unknown. As pointed out by Jiang et al. in [1], to minimize the increase of area, buffer insertion is preferred for fixing highly critical paths, and gate sizing is more suitable for addressing mildly critical paths. Therefore, we first fix highly critical paths by buffer insertion, and then recover area/power while optimizing for setup time using gate sizing. We only optimize for datapaths in between registers; clock network remains untouched throughout our optimization process to simplify the removal of setup/hold time violations.

Timing optimization can be approached in several ways. Our submission examines algorithms that can be parallelized efficiently and scalable in a graph-based parallelization framework. Specifically, we use operator formulation of algorithms [2], a *data-centric* abstraction of algorithms, to analyze the parallelism available in the algorithms used in our optimizer, ParallelClosure; and then parallelize the algorithms with the Galois framework [3], [4], a C++ library for parallel programming that implements the operator formulation.

The rest of the paper is organized as follows. Section II illustrates the buffer insertion in ParallelClosure. Section III elaborates the gate sizing in ParallelClousre. Section IV describes the parallelization strategy used in ParallelClosure. Section V reveals the current limitations of ParallelCloure and the plan to fix them in the future. Section VI concludes the paper.

II. Buffer Insertion

We use buffer insertion to first fix maximum capacitance violations, and then fix hold time violations.

To fix maximum capacitance violations, we identify pins that are driving loads larger than their maximum capacitance limits. Then for each such pin, we insert a buffer in between the pin and the loads. The inserted buffer should be of the minimum size that can drive the load legally. If no such buffer exists, the buffer of the largest size is inserted to reduce the magnitude of violating the maximum capacitance limits.

To fix hold time violations, we adopt the technique presented in [5]. We first perform STA and sort all the path endpoints by their worst hold time slacks across multiple corners. We then find the endpoint with worst negative hold time slack s_h, its corresponding path, and the associated corner. For all the edges on the path, we pick the edge (i, j) that is a wire segment having the largest setup time slack (s_s) for the corresponding corner, and then insert buffers in between pins i and j. To reduce the number of buffers inserted, we always insert the smallest buffer in the cell library. Since we are given cell libraries with non-linear delay model (NLDM), we approximate the delay of a buffer, d_{buf}, with the slew at i and the pin capacitance at j. To minimize the impact to setup time, the number of buffers we insert is $\max\{1, \lceil \frac{\min\{s_s, |s_h - thrd_h|\}}{d_{buf}} \rceil\}$, where $thrd_h$ is a predefined threshold for hold time slacks. We repeat this process until all the hold time slacks are above $thrd_h$. In this contest, $thrd_h$ falls in the interval of $[30ps, 80ps]$ depending on the original worst hold time slack of the given design.

III. Gate Sizing

We use gate sizing to optimize for setup time, area and leakage power, while not introducing hold time violations and satisfying maximum slew requirement.

TABLE I: Impact of Incremental Design Changes

Technique	Setup Timing	Hold Timing	Max Transition	Leakage Power	Area
Add Buffer	Degrade	Improve	Improve	Increase	Increase
Upsize Gate	Improve	Degrade	Improve	Increase	Increase
Downsize Gate	Degrade	Improve	Degrade	Decrease	Decrease
Delete Buffer	Degrade or Improve	Degrade or Improve	Degrade or Improve	Decrease	Decrease

In this contest, we implement the slew targeting method introduced by Held in [6], because (1) it directly addresses slew violations, (2) it avoids a large number of incremental timing updates, and (3) it is highly parallelizable. Since the paper focuses on fixing setup time violations, we generalize the method a little bit to handle hold time violations.

A. Rationale behind Slew Targeting

Slew targeting works by associating each gate output pin with a slew target to guide the gate sizing process [6]. To upsize/downsize a gate, just decrease/increase the slew target of its output pins. To facilitate MCMM optimization, each pin should have slew targets for all combinations of corners and delay modes.

The original slew targeting method assumes that delay and output slew are monotonic with respect to input slew. However, this assumption does not hold in general. For example, in the cell libraries given by the contest committee, this monotonicity does not hold when input slew is large and the output load is small for several cells. We think slew targeting still works in this case for two reasons: (1) The case when monotonicity does not hold is the situation we want to optimize away for the design; and (2) empirically, output slew and delay are weak functions of input slew, and the small fluctuations will not affect the accuracy of the final solution of our implementation by much.

In order for slew targeting to work, the following questions need to be addressed:

1) How to set the initial slew target for each gate output?
2) How to update the slew targets?
3) How to assign cells to gates?
4) When does the gate sizing converge?

B. Slew Target Initialization

We perform STA after buffer insertion and use the slew values as the initial slew targets.

C. Slew Target Update

Let $slewt(p)$ be the slew target of an output pin p of gate g. In setup time analysis, if p has a negative slack and it is on a critical path, we upsize g by decreasing $slewt(p)$; otherwise, we downsize g by increasing $slewt(p)$. In hold time analysis, the operation is opposite: downsize the gate when one of its output pins has negative slack and the pin is on a critical path, and upsize otherwise.

To know whether p is on a critical path, we compute the local criticality of p, $lc(p)$, as follows. Let $slack^-(g) = \min\{slack(i)|(i,j) \in W \wedge j \in input(g)\}$, where W is the set of wires and $input(g)$ is the set of input pins for gate g. Then $lc(p) = \max\{slack(p) - slack^-(g), 0\}$. Note that if p belongs to a combinational gate g, then $lc(p) \geq 0$, and p is on a critical path if and only if $lc(p) = 0$. Since we only optimize for datapaths in between registers, this definition of $lc(p)$ suffices to guide the update of slew targets.

Now we elaborate how to update the value for slew target of p. Instead of computing the change in slew target based on slacks as in [6], we leverage the information in the given cell libraries to directly come up with new slew targets.

Tables II and III show the rising slews for pin Z of cell BUF_X1 and that of BUF_X2, respectively. Note that (1) both tables contain similar values, and (2) for a given input slew and output capacitance, the corresponding entry in Table III is left to the one in Table II, and the two entries are always on the same row. For example, for input slew value of 15.6743 and output capacitance value of 30.3269, the output slew is given by the entry at row 3 and column 6 in Table II, and by the entry at row 3 and column 5 in Table III. This relation also holds for other types of functionally equivalent cells with different sizes across all corners. Therefore, the slew target of p can be updated according to a sequence of numbers derived through table look-up into output slew tables using current input slew and different output capacitance. We call this sequence the slew possibilities of p.

To upsize/downsize a gate g to which pin p belongs, we take smaller/larger terms in p's slew possibilities, which can be approximated as follows. Through table look-up, the smallest term, lb, is computed using the current input slew and zero driving capacitance; and the largest term, ub, using the current input slew and the maximum driving capacitance of p. Intermediate terms can be approximated based on a geometric sequence whose common ratio $r = (\frac{ub}{lb})^{\frac{1}{k}}$, where k is a tunable parameter. In this contest, we have 8 terms in slew possibilities, and we set $k = 20$. The intermediate 6 terms are $lb \cdot r$, $lb \cdot r^3$, $lb \cdot r^5$, $lb \cdot r^8$, $lb \cdot r^{11}$, and $lb \cdot r^{15}$.

Recall that each output pin may have different slew targets for different timing corners and delay modes. These slew targets can be set independently from each other.

By updating the slew targets properly, we can eliminate maximum slew violation by construction.

D. Cell Assignment

1) Order of Sizing: Having the slew target updated, we now size the gates. Since output capacitance usually has larger impact on the cell timing than the input slew, we want to fix all the downstream gates of g before sizing g. This requires a reverse topological order of all gates. However, if all the combinational logic and registers are included, then the circuit topology has cycles. We cut the cycles at the data inputs to registers in order to work with a directed acyclic graph (DAG), from which we define a reverse topological order for gates.

TABLE II: The Rising Slew Table for BUF_X1 from Nangate 45nm Typical Corner

Input Transition \ Output Capacitance	0.365616	1.895430	3.790860	7.581710	15.163400	30.326900	60.653700
1.23599	3.33809	5.59725	8.60523	14.8575	27.5164	52.8765	103.604
4.43724	3.33727	5.59699	8.60578	14.8576	27.5188	52.8775	103.599
15.6743	3.40246	5.62543	8.61689	14.8582	27.517	52.8787	103.599
37.1331	4.36023	6.10464	8.84317	14.9465	27.5247	52.8726	103.605
70.5649	5.85455	7.27833	9.43026	15.0988	27.6409	52.9322	103.603
117.474	7.61897	9.14083	10.8314	15.5462	27.6912	53.0238	103.669
179.199	9.58764	11.3565	13.0249	16.7347	27.8716	53.0513	103.775

TABLE III: The Rising Slew Table for BUF_X2 from Nangate 45nm Typical Corner

Input Transition \ Output Capacitance	0.365616	3.786090	7.572190	15.144400	30.288800	60.577500	121.155000
1.23599	3.10917	5.67693	8.71288	14.9785	27.635	52.969	103.657
4.43724	3.10875	5.67786	8.71402	14.9788	27.6339	52.9719	103.66
15.6743	3.20354	5.70984	8.72471	14.9811	27.631	52.9744	103.651
37.1331	4.20264	6.15463	8.94062	15.0761	27.6468	52.967	103.666
70.5649	5.70174	7.27713	9.47332	15.2076	27.7634	53.0379	103.659
117.474	7.47026	9.1372	10.8172	15.6132	27.8134	53.1232	103.735
179.199	9.44195	11.3787	12.9969	16.7387	27.9813	53.162	103.831

2) Input Slew Estimation: As the given cell libraries are of NLDM, and upstream gates of g are not fixed yet, we need to estimate slews for each of g's input pins. Figure 1 shows an example. Let $est\_slew(p) = \theta slewt(p) + (1-\theta)slew(p)$ be the estimated slew at output pin p, where $slew(p)$ is the slew of p from STA, and $\theta \in [0,1]$ is a parameter that is close to 1 when gate sizing starts but close to 0 when gate sizing is about to converge. Then, $est\_slew(q) = est\_slew(p') + slew\_degrad(p', q)$, where $slew\_degrad(p', q)$ denotes the slew degradation due to the wire segment from p' to q, which can be computed by the RC-tree model [7].

3) Cell Selection: Let $out\_est\_slew(p)$ be the estimated output slew at pin p through table look-up using the load seen by p and $est\_slew(q)$, where (q, p) is a timing arc. A pin may have different slew targets for each combination of timing corners and setup/hold time analysis, so use the corresponding one when sizing a gate at a specific corner.

- In setup time analysis, we select for gate g the minimum size of equivalent cells such that $out\_est\_slew(p) \leq slewt(p)$ holds for all output pins p of gate g at the corner being considered. Then we choose the maximum sized cell from the selected cells across all corners for gate g in setup time analysis. We define $size_s(g)$ as the cell size selected for g in setup time analysis.
- In hold time analysis, the selection works in the opposite way: we select the maximum size of equivalent cells such that $out\_est\_slew(p) \geq slewt(p)$ holds for all output pins p of gate g at the corner being considered; and then choose the minimum sized cell from the selected cells across all corners for gate g in hold time analysis. We define $size_h(g)$ as the cell size selected for g in hold time analysis.
- If $size_s(g) \leq size_h(g)$, then assign the cell of size $size_s(g)$ to gate g. This is because any cell sizes in $[size_s(g), size_h(g)]$ satisfy both slew targets in setup time and hold time analysis. In order to minimize area and leakage power, we choose $size_s(g)$ for gate g.

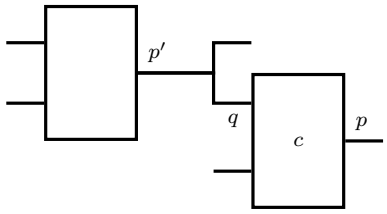

Fig. 1: A Cell c and a Predecessor

- If $size_s(g) > size_h(g)$, then assign the cell of size $size_h(g)$ to gate g. This is because there is no cell size that can satisfy both slew targets in setup time and hold time analysis at the same time. We honor hold time constraints in this case without sacrificing setup time seriously, since hold time violations cannot be fixed by adjusting the clock frequency.
- The cell size for gate g should not violate maximum capacitance limits for any of g's output pins. Otherwise, the cell of minimum size without such violation is chosen for gate g.

E. Convergence

After cell selection for all gates, we run STA again. If the worst negative slack improves for all corners, we proceed to next round of gate sizing using the current cell assignment. Otherwise, we score the current cell assignment by a linear combination of worst negative slack, average total negative slack over all path endpoints, and average cell area over all gates. The lower the score, the better. If the score decreases for all corners, we also proceed to next round of gate sizing using the current cell assignment. If not, we revert to the previous cell assignment and quit gate sizing.

IV. Parallelization

A. The Operator Formulation

We leverage parallelization to achieve fast design optimization. To do this, we analyze the parallelism available in

our optimizer with the operator formulation, a *data-centric* abstraction that presents a *local view* and a *global view* of algorithms [2].

The local view is described by an *operator*, a graph update rule applied to *active nodes*, which are nodes in the graph where there is computation to be done. Each operator application, called an *action*, may read from/write to a set of nodes and edges around the active node, termed the *neighborhood* of the action. Active nodes become inactive once the actions are finished.

The global view is captured by (1) the location of active nodes, and (2) the order by which the actions must appear to be carried out. For *unordered* algorithms, e.g., chaotic relaxation for SSSP, processing active nodes in any order gives the same answer. However, some orderings may be more efficient than the others.

Algorithms can be categorized as *data-driven* or *topology-driven* based on the pattern of active nodes. A data-driven algorithm begins with a set of initially active nodes, generates new active nodes on the fly, and terminates when there are no more active nodes to be processed. Dijkstra's single-source shortest path (SSSP) algorithm is an example. In contrast, a topology-driven algorithm makes sweeps over all nodes until certain convergence criteria is reached. Bellman-Ford algorithm is an example.

Although this discussion has focused on algorithms in which nodes are active, the same approach works for algorithms in which *edges* are active. So in general, we refer to active *elements* in the graph.

Parallelism in graph algorithms can be exploited by parallel execution of actions with disjoint neighborhoods.

B. Available Parallelism

With the operator formulation of algorithms, we are ready to analyze the parallelism available in our design optimizer, composed of static timing analysis, buffer insertion, and gate sizing using slew targeting.

1) Static Timing Analysis: Given a synchronous design, STA represents the design as a timing graph $G = (V, E)$, a DAG, where nodes in V are the pins and $(u, v) \in E$ are wires or timing arcs between pins. For an edge (u, v), we say u is v's predecessor and v is u's successor. STA then computes for all pins their arrival times and slew rates in topological order from primary inputs (forward propagation); and computes required times and slacks in reverse topological order from primary outputs and constrained pins, e.g., register inputs (backward propagation).

As the timing graph for a synchronous design is a DAG, the processing order in STA can be enforced by explicitly tracking the number of unresolved dependencies for each pin. A pin can be processed if all its dependencies are resolved, and all the pins whose dependencies are cleared at the same time can be processed in parallel.

Specifically, for each pin v, let $dep(v)$ be its number of unresolved dependencies, $pred(v)$ its predecessors, and $succ(v)$ its successors. For forward propagation, initially $dep(v) = |pred(v)|$; after pin u computes its arrival time and slew rate, u atomically decrements $dep(v)$, $v \in succ(u)$. For backward propagation, initially $dep(v) = |succ(v)|$; after a pin w computes its required time and slack, w atomically decrements $dep(v)$, $v \in pred(w)$. For both forward and backward propagations, pin v becomes active when $dep(v) = 0$, and pins whose deps are zero at the same time can be processed in parallel.

As STA needs to track active pins explicitly, STA is a *data-driven* algorithm. All active pins can be processed in any order, so STA is an *unordered* algorithm. Note that the ordering in STA only defines when pins should become active but not the processing order of active pins.

2) Buffer Insertion:

a) For fixing maximum load constraints: As explained in Section II, a buffer is inserted in between a gate output pin, v, and its load, $load(v)$, if $load(v) > maxC(v)$, where $maxC(v)$ is the maximum load that can be driven by v. Every gate output can insert such a buffer independently, so this is a *topology-driven*, *unordered* algorithm.

b) For fixing hold time violations: Recall from Section II that we insert buffers in a design to fix hold time violations round by round. In each round, we run STA to identify the most-critical hold-time path, and insert buffers to the wire segment having the largest setup-time slack on the path. Since there is only one active edge in a round, our buffer insertion scheme contains no parallelism. It is a *data-driven* algorithm, since the active edge in a round depends on the circuit timing in the round.

3) Gate Sizing with Slew Targeting: Recall from Section III that gate sizing with slew targeting [6] is a round-based algorithm. In each round, a full STA is performed first, then all pins set their slew targets, and then all gates select their cells. Finally, the new cell assignment is scored and then kept or reverted.

a) Setting slew targets: Slew targets can be set for each pin independently in any order, since each pin p can get from STA the required information for computing new slew target in the next round: current slew, p's own slack, and the neighboring gates' pins' slacks. Therefore, setting slew targets is a *topology-driven*, *unordered* algorithm.

b) Cell assignment: The parallelism available in assigning cells to gates is similar to that in computing required times and slacks in STA. As mentioned in III-D1, gates should be sized in reverse topological order on the graph of gate connectivity, with edges feeding into register data inputs ignored. Instead of constructing a gate connectivity graph and then ignore some edges in it, we can enforce the order of sizing gates with the idea that a gate should be processed after all its pins are processed. This reduces the memory space requirement of ParallelClosure by utilizing the timing graph for STA for gate sizing as well.

Specifically, we associate for each gate g a counter, $untouched(g)$, to track the number of untouched pins for g. Initially $untouched(g) = |pin(g)|$, where $pin(g)$ is the set of pins belonging to gate g. The dependency among

pins are tracked as in Section IV-B1 for computing required times and slacks in STA. When a pin v is processed, v also atomically decrements $untouch(gate(v))$, where $gate(v)$ is the gate to which pin v belongs. Gate g becomes active when $untouched(g) = 0$. Initially, no gates are active, and pins with no successors are active. All gates with $untouched = 0$ simultaneously can be assigned to cells in parallel.

Similar to STA, cell assignment is a *data-driven*, *unordered* algorithm. Nevertheless, its operator is different from the one for computing required times and slacks in STA.

c) Scoring a cell assignment: As mentioned in Section III-E, a new cell assignment is scored as a linear combination of worst negative slack, average total negative slack, and average cell area. The score can be computed by dividing the gates to threads to compute thread-local results, and then reduce all the thread-local results to the final answer. All gates and constrained pins can be processed in parallel. Therefore, scoring a cell assignment is a *topology-driven*, *unordered* algorithm.

d) Keeping/reverting a cell assignment: Each gate can be processed independently, so this is a *topology-driven*, *unordered* algorithm.

C. Implementation in Galois

We implement our design optimizer using the Galois framework [3], [4], a C++ library for parallel programming based on the operator formulation. The Galois framework (1) provides parallel data structures, and language constructs for highlighting parallelization opportunities; and (2) supports dynamic work generation, load balance, resource management, and transactional execution of operators.

All sub-algorithms in our design optimizer are implemented as described in Section IV-B except for buffer insertion, where everything is done sequentially to maintain the mapping consistency from names to gates/wires. The timing graph is constructed as follows: gates track their pins; pins track their wires and belonging gates, if any; and wires track their member pins. Cell assignment is associated with gates and pins consistently. All timing corners and delay modes are analyzed simultaneously for all algorithms. This is done by storing MCMM data for all pins, and processing all MCMM data when processing a pin/gate.

V. Limitation

ParallelClosure, our design optimizer for this contest, currently has the following limitations.

- The buffer insertion is purely sequential. This will be addressed from two aspects: (1) algorithmically, we need to find a buffer insertion algorithm that has more parallelism; (2) in terms of data structure, we need to change the implementation for representing Verilog netlists and SPEF wire models so that concurrent updates to the circuit connectivity can scale and do not corrupt the mapping consistency in between names and gate/wires.
- It inserts lots of buffers when there is a large number of paths with hold time violations. This reduces the quality of the design, and also worsens the execution time of the algorithm. This will be addressed in the future with techniques related to clock network, such as clock skew scheduling [8].
- The buffer insertion in ParallelClosure currently neither considers the wire topology for a net nor the optimal buffer insertion for a net. This will be addressed by incorporating tree topology generation for a net, e.g., C-tree as in [9]; and van Ginneken's algorithm for optimal buffer insertion given a tree topology [10].
- The parameters needs to be further tuned. For example, the convergence criteria of gate sizing can be tuned so that the sizing can proceed for more than two rounds.

VI. Conclusions

In this contest, we explore existing efficient buffer insertion and gate sizing algorithms in the literature to fix timing violations; and optimize area, leakage power, and clock period for a design without changing its functionality. We prototype a design optimizer, ParallelClosure, by incorporating the idea presented in [5] to insert buffers for fixing hold time violations; and by generalizing the gate sizing by slew targeting [6] to optimize for area and leakage power while not introducing setup or hold time violations. We analyze the parallelism available in STA and gate sizing by slew targeting, and use the Galois framework to parallelize them in ParallelClosure. We believe that ParallelClosure is an efficient and effective design optimizer for designs with a small number of timing violations.

References

[1] Y. Jiang, S. S. Sapatnekar, C. Bamji, and J. Kim, "Interleaving buffer insertion and transistor sizing into a single optimization," *IEEE Transactions on Very Large Scale Integration Systems*, 1998.

[2] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The TAO of parallelism in algorithms," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pp. 12–25, 2011.

[3] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP, pp. 456–471, 2013.

[4] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, pp. 78–87, Apr. 2016.

[5] N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Minimum padding to satisfy short path constraints," *International Conference on Computer-Aided Design*, 1993.

[6] S. Held, "Gate sizing for large cell-based designs," *Design, Automation and Test in Europe*, pp. 827–832, 2009.

[7] T. C. Committee, "Tau 2019 contest education." https://sites.google.com/view/tau-contest-2019.

[8] E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 665–692, 2001.

[9] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar, A. J. Sullivan, and P. Villarrubia, "Buffered steiner trees for difficult instances," *IEEE/ACM Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 1, pp. 3–14, 2002.

[10] L. P. P. P. van Ginneken, "Buffer placement in distributed rc-tree networks for minimal elmore delay," *Internatinal Symposium on Circuits and Systems*, pp. 865–868, 1990.