Cyclone: A Static Timing and Power Engine for Asynchronous Circuits

Wenmian Hua
Yale University
wenmian.hua@yale.edu

Yi-Shan Lu, Keshav Pingali
University of Texas, Austin
yishanlu@utexas.edu, pingali@cs.utexas.edu

Rajit Manohar
Yale University
rajit.manohar@yale.edu

***Abstract*—Asynchronous circuits have potential advantages of higher speed and lower power consumption compared to their synchronous counterparts, but their poor CAD support is a major issue limiting adoption. In this paper, we present an integrated timing and power analysis engine capable of handling large asynchronous circuits. For timing, we introduce the notion of performance and correctness slack for asynchronous circuits; for power, we compute both the static and dynamic components. We provide a hierarchical approach to constructing the event-dependency graph, and use the Galois framework for parallelization to achieve fast runtime. The net result is Cyclone, a fast and accurate engine for both static timing and power analysis of asynchronous circuits.**

I. Introduction

Asynchronous circuits do not use global clocks, and have potential advantages over synchronous circuits including higher speed, lower power consumption, and robustness to process variations. Most recently, researchers have shown the benefits of asynchronous circuits in the context of field-programmable gate arrays [1] and neuromorphic computing [2], [3]. Unfortunately, researchers interested in evaluating or adopting asynchronous circuits are immediately confronted with the issue of the lack of design support by mainstream commercial flows.

A key component of any CAD flow for chip design is timing analysis, and the timing analysis problem for asynchronous logic is quite different from its synchronous counterpart. As is shown in Figure 1, timing analysis of synchronous circuits can be effectively done by analyzing acyclic regions of combinational logic separated by clocked state-holding elements. Max-delay constraints (setup time) determine the performance of the circuit, and min-delay

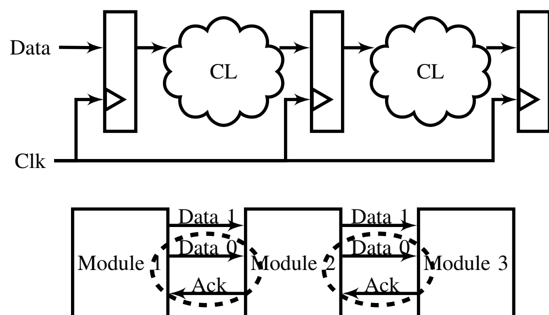

Figure 1: Synchronous vs asynchronous circuit

constraints (hold time) determine if the circuit will operate correctly. In asynchronous circuits, however, the time of each signal transition depends on that of many other signal transitions, and the timing paths that need to be examined are cyclic. Thus, traditional timing analysis methods and tools for synchronous circuits cannot be easily adapted for this purpose.

The problem of timing analysis for asynchronous circuits has been explored in many previous papers. One approach is to adapt commercial synchronous static timing analysis tools for use with asynchronous designs as in [4]. This approach involves manually (or scripted) removal of timing arcs so that the resulting timing graph is acyclic. The actual performance is then calculated by combining timing paths [5]. To be accurate, this approach requires that the designer knows the cyclic critical path in the asynchronous circuit—which will be very challenging and time consuming when the circuit is complex or generated by a synthesis procedure.

Another approach is to directly build and analyze timing graph models for asynchronous circuits. Petri nets [6], marked graphs [7], and repetitive event-rule (*RER*) systems [8] (and its extended version: extended repetitive event-rule (*XRER*) system [9]) are several representative models for asynchronous timing analysis. Properties of these models and timing of asynchronous circuits have been well studied in both mathematics (e.g. [10], [11], [12]) and engineering literature [8], [13], [9], [14], [15], [16], [17]. Maximum cycle ratio algorithms, surveyed by Dasdan [18], can be used to (i) characterize the performance of circuits assuming AND-causality and fixed delays, and (ii) derive performance bound for data-dependent timing analysis [19].

Most of the previous performance analysis of AND-causal asynchronous circuits was approximate, so it is precluded from analyzing timing constraints required for correctness. Recently, Hua [15] established theoretical results that can determine the *exact* time at which every event in an asynchronous circuit occurs. These results hold even when the asynchronous timing graph doesn't collapse into a single strongly connected component[1] (a requirement for previous results). This enables the creation of an engine that can deter-

[1]A simple data FIFO implemented with bundled-data logic is an example of an asynchronous circuit with more than one strongly connected component.

mine both the performance and timing constraints required for correct operation of the asynchronous circuits. However, none of these previous efforts consider slew rates. Therefore they cannot compute accurate gate delays and power using sophisticated delay models such as non-linear delay model (NLDM).

In this paper, we address several major differences and difficulties involved in asynchronous timing and power analysis. These include (i) cell characterization, (ii) timing graph construction, (iii) steady-state slew rate computation using NLDM in the presence of cycles of gates, (iv) the maximum cycle ratio algorithm. We also introduce the asynchronous analog of the notions of arrival time, required time, performance slack and correctness slack. The net result is Cyclone, an implementation of a comprehensive asynchronous timing and power analysis engine. The engine takes as input an asynchronous circuit netlist, multiple cell libraries using NLDM in the Synopsys liberty (`.lib`) format, and timing constraints from desired asynchronous circuit family for correctness. It performs a multi-corner analysis and reports maximum cycle ratio, power consumption, and two types of timing slacks: (i) the *performance slack* of each gate, which determines how much a gate can be slowed down without impacting the performance of the asynchronous circuit; and (ii) the *correctness slack* of each timing constraint, showing any violations of timing requirements or the margin that remains. We parallelize Cyclone using the Galois framework, and achieve up to 18.76× speedup for full static timing analysis compared to the sequential runs. Finally, we plan an open-source release of Cyclone to the community.

The rest of the paper is organized as following: Section II summarizes the theoretical results that we leverage for timing analysis; Section III introduces our approach to timing constraints for correctness in asynchronous logic; Section IV describes the Cyclone analysis flow, focusing on its major differences and difficulties compared to synchronous timing and power analysis; Section V describes the parallelization strategies we use to parallelize all the core algorithms in Cyclone; Section VI includes our experimental results; and finally Section VII concludes and gives future work.

II. Preliminaries

We adopt the *repetitive event rule (RER) system* model introduced first by Burns [8]. Some of the basic definitions and properties of gates and *RER* systems are discussed in [8], [13], [15]. Here, we provide a more intuitive summary of the main concepts and key results from the literature.

In asynchronous timing analysis, the timing properties of gates are expressed as constraints on *events*. A gate with output x contributes to an infinite set of events of the form $\langle x\uparrow, 0\rangle$, $\langle x\downarrow, 0\rangle$, $\langle x\uparrow, 1\rangle$, ... that corresponds to the different occurrences of rising and falling transitions of x. Each such event is assigned a time, corresponding to when the event occurs.

Given a complete asynchronous circuit as a collection of gates, the relationship between events can be captured by edges labelled with a delay between events, where the edges correspond to timing arcs (AND constraints). Note that this approach would result in an infinite timing graph, since we have an infinite event set. However, as the circuit is a finite structure, its timing behavior is repetitive (as shown in [8], [13], [15]), so the conceptually infinite timing graph can be collapsed into a finite graph that closely resembles the topology of the gates in the circuit.

This finite graph-based representation is called the *repetitive event rule system*, or *RER* system. Nodes in *RER* system correspond to signal transitions like $x\uparrow$ and $x\downarrow$ (i.e. we delete the integer index). There are two types of edges in *RER* system: those connecting events from the same iteration (we say that $\epsilon = 0$ for these edges), and those that connect events from one iteration to the next (we say that $\epsilon = 1$ for these edges). Edges are also annotated with delay values (α) as their weights. An example of *RER* system without delay values is provided in Section IV-A.

We summarize a collection of previous theoretical results that compute exact timing of all events.

Definition 1: Given a cycle c in the *RER* system consisting of edges $e_1, \ldots, e_n$, cycle ratio is defined to be $\delta(c)/\epsilon(c)$ where $\delta(c) = \sum_i \alpha_i$ is the delay along the cycle, and $\epsilon(c) = \sum_i \epsilon_i$ is the ϵ-sum of the cycle. A critical cycle c is a simple cycle with the maximum cycle ratio $p^\star = \max_{c \text{ is a cycle}} \delta(c)/\epsilon(c)$.

Let $\hat{t}(\langle e, i\rangle)$ be the actual time at which the $(i+1)$th occurrence of signal transition e given the constraints specified by the *RER* system ($i \in \mathbb{N}$). For example $\hat{t}(\langle x\uparrow, 3\rangle)$ would be the time at which the fourth rising transition of x occurred. A weaker version of the main theorem from [15] states that:

Theorem 1: In a strongly connected *RER* system with AND-causality and fixed delay values, there are integer constants M and $k^\star$ such that for all transitions e and all integers $n \geq k^\star$,

$$\hat{t}(\langle e, n+M\rangle) - \hat{t}(\langle e, n\rangle) = Mp^\star.$$

Under the fixed delay model and conservative AND-causality assumption, after a finite time interval, each signal transition in the circuit will occur periodically with a period equal to $Mp^\star$ for some integer M. Our asynchronous timing analysis will calculate the values M and $p^\star$ of the *RER* system modeled from the input circuit. The theorem also guarantees that, after a finite initial interval, the function $\hat{t}$ can be completely characterized by finite quantities—namely, the quantities $\hat{t}(\langle e, n+i\rangle)$ for events e and i satisfying $0 \leq i < M$. Finally, although we have included strong connectivity in the statement of Theorem 1, the result holds even without the strong connectivity assumption; all that is required is that there is a path from the critical cycle to every transition in the *RER* system. For more details as to the precise requirements, the reader is referred to [15].

III. Timing Constraints

Given prior work, the key performance indicator for the asynchronous circuit is the value of $p^\star$, the maximum cycle ratio in the *RER* system. In addition, the periodicity result (Theorem 1) states that there is an integer M such that the circuit's timing is only periodic every M iterations. Hence, we track M different times for each transition to completely characterize the timing function $\hat{t}$.

In what follows, we assume that there is a path from an event on the critical cycle to every other event in the system, the condition needed for Theorem 1 to hold [15]. Note that this is equivalent to assuming there is a path from every event in the critical cycle to every other event in the system.

A. Arrival time, required time, and performance slack

We now introduce the notion of arrival time, required time, and performance slack for all the events in the asynchronous system. Increasing the delays of edges on a critical cycle further slows down the asynchronous circuit. Hence, all events on the critical cycle have zero performance slack [15].

Since we know that there is a path from every event on the critical cycle to all other events in the *RER* system, we can compute the *arrival time* of all the other events in the system relative to the events on the critical cycle. We arbitrary select one event on the critical cycle to have time $t = 0$ such that all the events have non-negative arrival times. All other event arrival times are determined by timing propagation similar to max-delay propagation in synchronous timing analysis. Note that we are *guaranteed* not to propagate through cycles, since we have already computed the arrival times for all the events on the critical cycle.

The required time can be computed in a manner analogous to the required time in synchronous logic, by working backward from the critical cycle. The initial condition used is that each event on the critical cycle has a required time that is equal to its arrival time.

Finally, we have M different performance slack values for each signal transition, obtained by subtracting the arrival time from the required time. As the circuit is static, the final slack value for a transition is given by the minimum across all M values. We report this value as the *performance slack* of a signal transition. This is the amount by which that signal transition could be delayed without impacting the performance of the asynchronous circuit.

B. Correctness slack

While asynchronous circuits have been widely described as not having any "timing closure" problems, this is not entirely accurate. While asynchronous circuits can be designed to be robust to timing, they also have some timing constraints necessary for correct operation. It has been shown that asynchronous circuits that must operate correctly without any timing constraints are extremely limited in what they can compute [20], [21]. One of the mildest forms of timing assumptions is the isochronic fork [20], which requires that a wire branch is faster than an adversarial sequence of gates [22]. More generally, we express our timing constraints using a *timing fork*, borrowing a notion from timed distributed systems [23].

A *timing fork* $\langle r, i\rangle : \langle x, j\rangle < \langle y, k\rangle$ consists of vertex $\langle r, i\rangle$ (the "root"), and two paths in the *RER* system: $p_{\langle r,i\rangle,\langle x,j\rangle}$ from $\langle r, i\rangle$ to $\langle x, j\rangle$ and $p_{\langle r,i\rangle,\langle y,k\rangle}$ from $\langle r, i\rangle$ to $\langle y, k\rangle$, where $j, k \in \{i, i + 1\}$. The timing fork requires that the correctness slack $sl = \min\{delay(p_{\langle r,i\rangle,\langle y,k\rangle})\} - \max\{delay(p_{\langle r,i\rangle,\langle x,j\rangle})\} \geq 0$. Note that a path might include the output transition of a gate, or might end at the input of a gate.

Timing forks are generated during logic synthesis, as different asynchronous logic families have different timing forks. Hence, we assume they are inputs to our timing engine. In our evaluation section, we have isochronic fork timing constraints as well as the "bundling" timing constraint required for bundled-data asynchronous circuits to illustrate two different asynchronous circuit families that our engine can analyze.

We remark that the notion of a timing fork is similar to the concept - relative timing constraints [24] and generalizes the hold time requirement in synchronous logic. The hold time requirement corresponds to a timing fork that begins at the root of the common clock subtree for two flip-flops, with one path going through the launching flip-flop and combinational logic to the input of a capturing flip-flop, and the second path going directly to the clock input of the capturing flip-flop.

IV. Cyclone Analysis Flow

With all of the preliminaries presented in sections II and III, we now describe our asynchronous timing and power analysis flow. Our asynchronous circuit design is specified using a hierarchical format that contains the logical specification of the gates for each component, transistor sizing information, and connectivity. From the design description, we automatically generate the SPICE netlist and circuit netlist for the design. Unique gates or small groups of gates that are repeatedly used in the design are factored out into *cells*, and the netlist is rewritten so that all gates correspond to cell instances with individual unique cells being specified by their SPICE netlists. This information is used for cell characterization, followed by timing graph generation and then timing and power analysis. The overall flow is outlined in Figure 2.

A. RER Construction and RER Skeletons

Unlike synchronous logic where the timing arcs are specified in the `.lib` file, in asynchronous logic the same gate used in different contexts may have different timing arcs. The index priority simulation (IPS) algorithm for quasi delay-insensitive (QDI) circuits developed in [9] computes

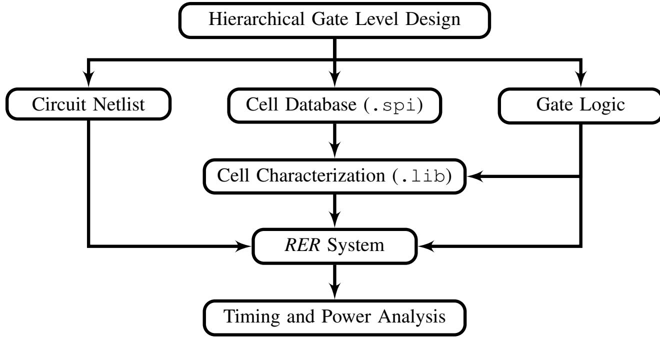

Figure 2: Overview of Cyclone analysis flow

the *RER* system from the gates and the state of the circuit when reset is asserted [9].

There are two major challenges with IPS algorithm: (i) *limited scalability*: the algorithm requires simulating the *entire* asynchronous circuit digitally until a repeating state is found—which can be *slower* than timing analysis in practice when the circuit is large; (ii) *limited capability*: the algorithm is designed to only handle systems with AND-causality and some limited forms of OR-causality, and cannot handle common scenarios that arise such as dual-rail encoded data. Our *RER* system generation method adopts the idea of digital simulation of asynchronous circuits, but has a number of modifications and improvements to address both limitations of IPS algorithm, and will work for general asynchronous circuit families.

To address limited scalability, notice that the goal of IPS is used to encode event dependencies as *RER* systems. These dependencies are determined by the local circuit within an individual process. We know the process boundaries within our circuits, because the input to the timing analysis flow is a hierarchical design. Hence, we partition the design into individual components at the level of modules with input and output channels, and generate for each module its local *RER* system. Also, we record the events that are shared between processes in the environment. This entire procedure only has to be executed once per unique process in the system; if modules are re-used, this step need not be repeated.

We refer to this as generating an *RER skeleton*—a building block of the overall *RER* system for the complete circuit. A circuit also needs to be closed by the environment in order for IPS algorithm to generate the correct *RER* skeleton. In our implementation, we connect data sources, which always send input data, and data sinks, which always consume output data, when necessary.[2]

To address limited capability, consider a simple scenario where data is encoded as a dual rail value on true and false rails. If we use IPS, then the *RER* skeleton generated will

[2]This is correct for slack elastic systems [25] like pipelined asynchronous circuits. More general systems will require more sophisticated techniques to generate valid environments.

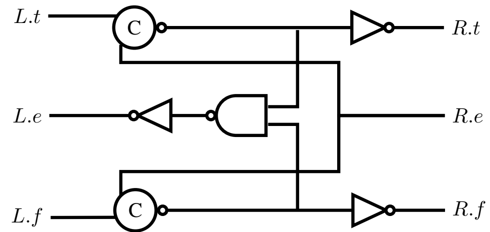

Figure 3: 1-bit WCHB FIFO Circuit

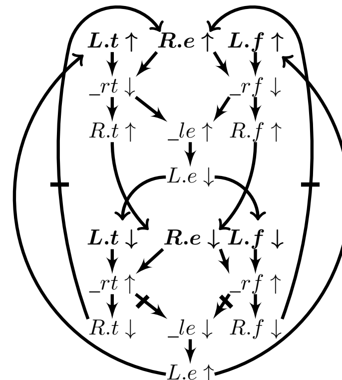

Figure 4: Complete *RER* skeleton for WCHB. Events generated by the environment are shown in bold.

only cover transitions resulting from either the true or false rail, but not both, because it always simulates and includes the data branch with highest index priority (predetermined) in its state graph. We build a complete *RER* skeleton by adopting the similar idea presented in [26], and tracking for both true and false data branches of the circuit to include all possible digital states for the *RER* skeleton. Our implementation does not assume the perfect symmetry as in [26], and can exclude non-reachable state transitions due to any timing constraints. Thus, our method can be applied to more general asynchronous circuit families. Based on this accurate state graph, we can construct a complete *RER* skeleton for local circuits. Assuming AND-causality, this *RER* skeleton provides a worst-case estimate of the timing analysis for asynchronous circuits as desired. This is similar to the situation in synchronous logic, where an upper bound on the maximum delay is computed during static timing analysis. Figure 3 gives the circuit of 1-bit weak-conditioned half-buffer (WCHB) and 4 shows its complete *RER* skeleton.

The final complete *RER* system for the circuit is constructed by combining *RER* skeletons for all components. The skeleton of two communicating processes will have shared events; those events can be merged to construct the combined *RER* system. This procedure avoids global digital simulation of the circuit, and is significantly faster than using traditional IPS.

The complete *RER* system obtained in this manner is the *timing graph* that we use for further analysis.

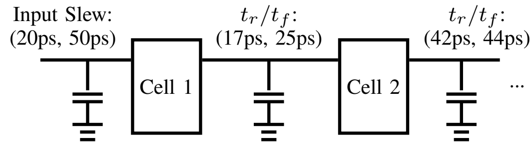

Figure 5: Slew range propagation between stages. The final value will depend on cell sizes and parasitic capacitance.

B. Cell Characterization

SPICE netlists for individual cells are simulated in HSPICE for cell characterization. Once again, previously characterized cells can omit this step by using the pre-computed cell information. We store characterization results in Synopsys Liberty (`.lib`) format. For timing analysis, we use the standard "non-linear delay model" (NLDM) for gate delay, and both delay and output transition time are stored in 2D tables, indexed by input slew rate and output load. For power analysis, we store leakage power and internal energy. Cell leakage power is a single value for each input vector; internal energy per timing arc is stored in 2D tables, indexed by input slew rate and output capacitance as well.

It should come as no surprise that the problem of characterizing combinational gates in asynchronous circuit design is the same as that in its synchronous counterpart. However, asynchronous circuits can include general state-holding gates with internal feedback, and our characterization framework can handle these gates as well.

C. Steady-State Slew Rate Computation

We need to annotate timing graph edges with delay values and calculate internal energy. Using NLDM, delay and internal energy values can be extracted from 2D look-up tables indexed by input slew rate and output capacitance. In synchronous timing and power analysis, circuits are partitioned into acyclic regions separated by clocked elements and the input slew rates are determined by parameters related to the clock input or user input (e.g. driving cells). For asynchronous circuits, however, there is no such "break point" that can be used to obtain the input slew rate of each stage. Instead, the input slew rate of a cell is given by the rise/falling transition time of its input, which in turn depends on the slew rate of the cell before it. Such a cyclic dependence needs a different technique to extract the slew rates used for delay and power calculation.

Fortunately, the rising/falling transition time of a cell output is a weak function of its input slew rate. Even starting with a wide range of input slew rate at a cell, the falling/rising transition time converges to a narrow range after a few stages of propagation, as the example shown in Figure 5.

More precisely, suppose $f(x) \in \mathbb{R}_{>0}$ is the output transition time for a cell output given the input transition time $x \in \mathbb{R}_{>0}$ and the output load. We note that given an interval $[a, b]$ with $b - a \geq \delta$, $f(b) \geq f(a)$ and $|(f(b) - f(a))/(b - a)| \leq c < 1$. Thus, for any arbitrary signal transition (node) on the timing graph, we can start with a slew rate range assumption and propagate this through the circuit in an iterative fashion. Capacitance information is obtained from parasitics as well as the characterization information from the `.lib` file, so that the appropriate slew rate is propagated through gates. We keep propagating the slew rate ranges on the timing graph for each signal transition, until the ranges converge. Experimental results show that slew rate ranges nearly always converge to very small intervals within several propagations, which provides an efficient way to calculate the accurate steady-state input slew rate for every signal transition.

D. Delay Calculation

Once slew rates are obtained for all signals, delay values for all the edges in the timing graph can be computed by using the slew rate and output load, combined with the cell characterization information in the `.lib` file. At this point in the flow, we will have a graph $G = (V, E)$ corresponding to the *RER* system constructed. In addition, each edge $e \in E$ has a weight $w(e)$ given by the delay values computed as above, and a tick value $\epsilon(e)$ that is either zero or one. Armed with this graph, we compute the cycle period $p^\star$ as described next.

E. Maximum Cycle Ratio Algorithm

The maximum (and minimum) cycle ratio problem has been explored in many previous papers. Burns' original *RER* formulation [8] uses linear programming as a direct method to compute $p^\star$, and recent work also uses linear programming as the baseline algorithm [16]. The performance of linear programming depends strongly on the solution technique and the quality of the solver. As we will show in Section VI, linear programming becomes much slower for large circuits.

Our implementation uses the Young-Tarjan-Orlin (YTO) algorithm [27], an asymptotically efficient implementation of Karp-Orlin's algorithm [28]. A previous survey has verified that these two algorithms are the fastest and most robust maximum cycle ratio algorithms both in theory and practice [18]. Although their worst-case time complexity is $\mathcal{O}(|V||E|log|V|)$, the runtime for practical timing graphs is significantly faster than the worst-case complexity.

F. Slack Computation

The work [13] shows that for a strongly connected system, a conservative M value will be the ϵ-sum of the critical cycle we found using YTO algorithm. It can be shown with slightly more work that this result on M also holds when the system has only one strongly connected component, which is the assumption enough for the discussion in this paper. [3]

[3]The system with more than one strongly connected component can have larger M value [15]

Once we have determined the maximum cycle ratio ($p^\star$) and M, we can use standard timing propagation as described in Section III to compute the performance slack of each event, and the correctness slack for each timing fork. These values have to be computed for M versions of each transition, and the most conservative value is reported as the slack.

G. Power Calculation

The power of a circuit includes leakage power and dynamic power, and in particular, dynamic power consists of internal (short circuit) power and switching power (charging or discharging the capacitors). Given the cell characterization results in the .lib files, we compute for each cell i $P_{leakage_m}(i)$, the maximum leakage power over different input vectors; and for each signal transition j in the *RER* system $E_{internal_m}(j)$, the maximum internal energy per cycle for timing arcs joining at j. Then we report the pessimistic total power of the circuit if it runs at frequency $\frac{1}{p^\star}$: $P_{total} = P_{leak}(circuit) + P_{dyna}(circuit) = \Sigma_i P_{leakage_m}(i) + \frac{\alpha(j)}{p^\star}(\Sigma_j E_{internal_m}(j) + \Sigma_j \frac{1}{2} C_{load}(j) V_{DD}^2)$ where $C_{load}(j)$ is the driving load of the gate output pin corresponding to j and $\alpha(j)$ is the activity factor of j. For QDI circuits, $\alpha(j)$ is 1 since each signal transition occurs at frequency $p^\star$. For combinational logic in bundled-data circuits, simulation-based activity factors can be incorporated into the dynamic power computed by Cyclone.

V. Parallelization Strategy

A. The Operator Formulation

We leverage parallelization to achieve fast timing and power analysis. To do so effectively, we analyze the parallelism available in Cyclone with the operator formulation [29], a *data-centric* abstraction of algorithms.

The operator formulation starts from identifying the data structures involved in the algorithm, e.g., a graph. *Active elements* capture where in the graph the computation needs to be done. An *operator* specifies the rules to update the graph, and it will be applied to active elements. Each application of an operator to an active element is called an *action*. An action may need to read from or write to a set of nodes and edges around the active element, which is termed the *neighborhood* of the action. Active elements become inactive once the actions are finished.

Algorithms can be categorized as *data-driven* or *topology-driven* based on the pattern of active elements. A data-driven algorithm begins with a set of initially active elements, generates new active elements on the fly, and terminates when there are no more active elements to be processed. In contrast, a topology-driven algorithm makes sweeps over all nodes/edges until certain convergence criteria is reached.

Scheduling needs to be considered when there are multiple active elements at the same time. For *unordered* algorithms, processing active elements in any order gives the same answer. However, some ordering may be more efficient than the others. For *ordered* algorithms, active elements should appear to be processed in certain ordering for correctness.

Parallelism in graph algorithms can be exploited among actions with disjoint neighborhoods.

B. Available Parallelism

With the operator formulation of algorithms, we are ready to analyze the parallelism available in Cyclone, composed of deriving edge weights, maximum cycle ratio, timing propagation, power computation and timing constraint checking.

1) Deriving Edge Weights: As mentioned in Section IV-C, edge weights in a timing graph are computed by computing the steady-state slew rates first and then computing delays by table lookup.

Slew propagation is a topology-driven, unordered algorithm. It sweeps all nodes until the slew intervals are small enough. In one sweep, all nodes can be processed in parallel, since for each node n, it computes n's new slew interval based on n's predecessors' old slew intervals and n's successors' loads.

Delay computation is a topology-driven, unordered algorithm. It only makes one sweep over all nodes. All nodes can be processed in parallel, since the table lookup for node n only reads from n's predecessors for slew rate, reads from n's successors for loads, and writes to n for delay.

2) Maximum Cycle Ratio: Recall from Section IV-E that we use Young-Tarjan-Orlin algorithm for computing maximum cycle ratio. It contains two main pieces: longest-path tree construction, and edge swapping [27].

Longest-path tree construction is a data-driven, unordered algorithm. It starts from a synthetic source node, which is connected to all nodes in the timing graph. By using a pull-style operator, e.g., reading from the predecessors of node n and writing to the n itself, a node will activate its successors if its distance from the source increases. All active nodes can be processed in parallel, even if their neighborhoods overlap because (i) maximum function exhibits monotonicity; and (ii) no updates will be missed using the aforementioned activation scheme.

Edge swapping is a data-driven, ordered algorithm. At any point, only the edge with the largest key value is active. This means that there is no parallelism available here.

3) Timing Propagation: Timing propagation consists of computing arrival time and computing required time, both are data-driven, unordered algorithms. For arrival time, nodes on the critical cycle with incoming ticked edges are active initially, and other nodes become active when their predecessors' arrival times are updated. For required time, nodes on the critical cycle are initially active, and other nodes will be activated once their successors' required times are updated. All active nodes can be processed in parallel, similar to longest-tree construction.

4) Power Computation: Computing leakage power for cells is a topology-driven, unordered algorithm, as it sweeps once over all cells, each of which reads from the cell libraries and writes to itself. Computing dynamic power is also a topology-driven, unordered algorithm, since nodes representing gate outputs are all active, only read from their neighboring nodes, and no two gate outputs can be neighbors.

5) Correctness Slack Computation: Correctness slacks are computed as follows. For a timing fork $\langle r, 0\rangle : \langle x, j\rangle < \langle y, k\rangle$ where $j, k \in \{0, 1\}$, $\langle r, 0\rangle$ is initially active with $delay(p_{\langle r,0\rangle,\langle r,0\rangle}) = 0$. We then compute $FO_{\langle r,0\rangle}$, which contains all the $\langle n, i\rangle$ reachable from $\langle r, 0\rangle$ using at most one ticked edge. Computing maximum/minimum delay from $\langle r, 0\rangle$ for nodes in $FO_{\langle r,0\rangle}$ is similar to propagating arrival times; as $FO_{\langle r,0\rangle}$ is always a directed acyclic graph (DAG), we can process its nodes in topological order. Finally, correctness slack is computed as defined in Section III.

Computing $FO_{\langle r,0\rangle}$ is a data-driven, unordered algorithm, as nodes can be marked in arbitrary order. Computing maximum/minimum delay from $\langle r, 0\rangle$ for all nodes in $FO_{\langle r,0\rangle}$ is a data-driven, ordered algorithm. $\langle r, 0\rangle$ is initially active in both algorithms. All timing forks can be processed in parallel provided that they are tracked separately.

C. Implementation in Galois

We implement Cyclone using the Galois framework [30], [31], a C++ library for parallel programming based on the operator formulation. The Galois framework (i) provides parallel data structures, and language constructs for highlighting parallelization opportunities; and (ii) supports dynamic work generation, load balance, resource management, and transactional execution of operators.

All sub-algorithms in asynchronous timing analysis are implemented as described in Section V-B. Below we illustrate some special handling using the Galois framework.

- Sub-algorithms use pull-style operators.
- To have better locality for memory accesses, all variables only have one copy. To ensure proper synchronization, atomic instructions are used for all sub-algorithms except for delay computation and edge swapping.
- Steady-state slew calculation starts from $[0, \rho]$ and ends when the resultant interval is smaller than $\rho/1000$, where ρ is the default maximum transition time. The difference given by the final interval is $< 1ps$ in our experiments.
- In longest-path tree construction, instead of introducing a synthetic source node, we initialize the distances of all nodes to $-\tau = -(1 + \sum_{e \in G} |delay(e)|)$ similar to that in [28] and let all nodes be active initially. τ can be computed with `galois::GAccumulator`: each thread adds $|delay(e)|$ for its portion of edges to a thread-local sum, and then the master thread reduces the thread-local sums to the final answer.
- During edge swapping cases, any updates to distance and ticks for a subtree are parallelized.
- For computing correctness slacks, each node keeps a map from timing forks' roots to its maximum/minimum path delays in order to track multiple timing forks simultaneously. The maps are backed by Galois per-thread memory allocators to avoid serialization due to system calls for allocating memory.

VI. Experimental Results

We implement Cyclone in C++ using the g++ 8.1 compiler, boost 1.67 libraries, and the Galois 5.0 framework for parallelization. Cyclone can support multi-corner analysis, as well as extracted parasitics in the `.spef` file format. For the reported runtime, we use two corners (SSA and FFA), and ideal wire models. All experiments are conducted on a Linux machine with CentOS 7, 56 cores Intel Xeon Gold 5120 2.2GHz CPU and 187GB memory.

We study different configurations of asynchronous dataflow pipelines including cyclic structures and branching pipelines. The benchmark circuits are listed in Table I. We use a small bundled-data benchmark (bd203) to show that Cyclone can analyze such circuits as well. Other circuits are converted automatically from synchronous benchmarks to their QDI analog (following [32]).

For each benchmark, we report (i) circuit properties: circuit name, number of gate pins, number of concurrent processes, number of timing constraints, $p^{\star}$, M, minimum correctness slack, and total power consumption; (ii) maximum cycle ratio runtime: best parallel runtime by YTO algorithm and best runtime by CPLEX, one of the fastest linear programming solver developed by IBM; and (iv) full static timing analysis (maximum cycle ratio + timing propagation + timing constraints checking) runtime: sequential and best runtime, and speedup. For best runtime, we also report the number of threads i used in parenthesis.

In full static timing analysis, the geometric mean of the best speedup across all benchmarks is 3.93, and that across large benchmarks ($> 200K$ pins) is 6.90.

By comparing columns 9 and 12 in Table I, we notice that YTO algorithm always takes up a significant fraction of time in full static timing analysis after parallelization; for large benchmarks, it takes 35% to 70%. This is because timing propagation can be parallelized as discussed in Section V, but edge-swapping in YTO algorithm [27] is inherently sequential. Hence, it is reasonable that Cyclone achieves 6.90× speedup on average in full static timing analysis for large benchmarks.

When calculating the maximum cycle ratio, CPLEX runs as fast as YTO algorithm for very small benchmarks, but much slower for large benchmarks. For example, when running on "*vga_lcd*" (with around 5.69M pins and 18.08M

Table I: Analysis results and runtime of Cyclone. Large benchmarks are below the thick line.

Circuit Properties								Maximum Cycle Ratio		Full Static Timing Analysis		
$Name$	$\#Pins$	$\#Proc$	$\#TC$	$p^\star(ns)$	M	$sl(ps)$	$P(mW)$	$YTO(i)(s)$	$LP(s)$	$seq(s)$	$best(i)(s)$	sp
$bd203$	495	20	10	0.44	1	33.17	0.44	0.01(1)	< 0.01	0.03	0.03(1)	1.00
$s27$	817	31	164	2.02	1	21.27	0.22	< 0.01(1)	0.01	0.02	0.02(1)	1.00
$c2670$	22171	796	4540	1.65	5	21.48	7.54	0.44(1)	0.69	1.69	0.86(7)	1.97
$s1488$	37766	1502	9008	5.64	2	21.27	3.93	0.23(28)	1.80	3.09	0.60(28)	5.15
$c3540$	42772	1682	10016	7.50	1	21.48	3.34	0.32(21)	1.53	1.96	0.59(14)	3.32
$c7552$	60355	2278	13444	4.11	1	21.48	8.47	0.95(14)	2.01	2.87	1.24(14)	2.31
$c6288$	72621	2877	17260	7.90	2	21.48	5.49	1.95(14)	2.80	7.93	2.63(14)	3.02
$s5378$	88292	3595	20880	4.39	3	21.27	11.71	0.97(14)	2.83	9.10	2.01(14)	4.53
$s9234$	137723	5594	32724	7.82	1	21.27	10.31	0.81(28)	5.23	6.71	1.63(28)	4.12
wb_dma	212247	8593	49464	4.17	2	21.27	29.40	2.08(21)	12.82	14.49	3.64(21)	3.98
$tv80$	315219	12801	75352	9.10	2	21.27	20.36	7.17(21)	18.41	65.20	11.06(21)	5.90
$ac97_ctrl$	650709	27215	154472	3.79	3	20.26	99.70	8.87(21)	79.93	102.53	15.54(35)	6.60
usb_funct	798895	32838	190020	8.17	1	21.27	57.06	3.87(42)	96.49	58.02	9.09(35)	6.38
$s38584$	807903	32975	192676	9.68	1	21.27	48.81	3.92(56)	79.39	51.07	9.58(35)	5.33
aes_core	1017817	40833	242616	8.61	1	21.27	69.49	4.76(49)	74.44	95.30	12.62(42)	7.55
vga_lcd	5689435	237059	1354068	7.05	1	21.27	473.20	105.48(56)	2959.01	2994.24	159.58(56)	18.76

linear constraints), CPLEX can be more than 28× slower than YTO algorithm, since CPLEX is a general purpose linear programming solution package, but YTO algorithm is designed specifically for maximum cycle ratio problem.

Note that the runtime and speedup are not only related to the scale of the circuit, but also related to the circuit topology and the structure of the critical path. For large values of M, the effective number of pins is multiplied by M since there are M different times associated with each signal transition. This increases the time for the delay propagation phase accordingly.

VII. Conclusions and Potential Future Work

In this paper, we present and implement Cyclone, which is, to our best knowledge, the first comprehensive description and implementation of timing and power analysis engine capable of handling large asynchronous circuits. By leveraging existing theory and efficient algorithms, and using the Galois framework for parallelization, Cyclone is fast and accurate, as demonstrated by the experimental results.

Specifically, we address several major issues in asynchronous timing and power analysis: (i) timing graph construction; (ii) slew propagation and delay calculations for annotating timing graph; (iii) fast maximum cycle ratio algorithm implementation; (iv) asynchronous notions of arrival time, required time, and performance slack, and correctness slack of timing forks; (v) the power calculation; and (vi) parallelization using Galois to achieve 6.90× speedup for large benchmarks in full static timing analysis.

This paper assumes AND-causality and gives conservative timing and power analysis for asynchronous circuit. In the future, we plan to extend our analysis to systems with both AND-causality and OR-causality. The current correctness slack uses delay values based on steady-state input slew rate, and we also plan to give more accurate correctness slack by considering transient slew rate during the initialization period of asynchronous circuits. Finally, we plan to incorporate advanced delay models that include statistical timing properties.

VIII. Acknowledgement

This work was supported in part by DARPA FA8650-18-2-7850, and a version without power analysis and correctness slack computation was presented at the DARPA POSH & IDEA Silicon Compiler Workshop, Jan 2019. The authors would also like to thank Steven M. Burns for his script that converts files of timing constraints to CPLEX LP format.

References

[1] C. LaFrieda, B. Hill, and R. Manohar, "An asynchronous FPGA with two-phase enable scaled routing," in *IEEE International Symposium on Asynchronous Circuits and Systems*, May 2010.

[2] A. Neckar, S. Fok, B. V. Benjamin, T. C. Stewart, N. Oza, A. R. Voelker, C. Eliasmith, R. Manohar, and K. Boahen, "Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model," *Proceedings of the IEEE*, vol. 107, pp. 144–164, Jan 2019.

[3] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, *et al.*, "Truenorth: Design and tool flow of a 65mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, October 2015.

[4] G. Gimenez, A. Cherkaoui, G. Cogniard, and L. Fesquet, "Static timing analysis of asynchronous bundled-data circuits," *24th IEEEInternational Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018.

[5] G. Miorandi, M. Balboni, S. M. Nowick, and D. Bertozzi, "Accurate assessment of bundled-data asynchronous nocs enabled by a predictable and efficient hierarchical synthesis flow," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 10–17, IEEE, 2017.

[6] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[7] F. Commoner, A. W. Holt, S. Even, and A. Pnueli, "Marked directed graphs," *Journal of Computer System and Sciences*, vol. 5, pp. 511–523, 1971.

[8] S. M. Burns, *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Instititute of Technology, 1991.

[9] T. K. Lee, *A general approach to performance analysis and optimization of asynchronous circuits*. PhD thesis, Caltech, 1995.

[10] J. Gunawardena, "Timing analysis of digital circuits and the theory of min-max functions," *International workshop on timing issues in the specifications and synthesis of digital systems (TAU)*, September 1993.

[11] S. Gaubert and C. Klimann, "Rational computation in dioid algebra and its application to performance evaluation of discrete event systems," *Algebraic computer in control*, 1991. Number 165 in Lecture Notes in Computer Science, Springer.

[12] M. Hartman and C. Arguelles, "Transience bounds for long walks," *Mathematics of Operational Research*, vol. 24, no. 2, pp. 414–439, 1999.

[13] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An algorithm for exact bounds on the time separation of events in concurrent systems," Tech. Rep. 94-02-02, University of Washington, Department of Compter Science and Engineering, 1994.

[14] P. B. McGee and S. M. Nowick, "An efficient algorithm for time separation of events in concurrent systems," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 180–187, 2007.

[15] W. Hua and R. Manohar, "Exact timing analysis for asynchronous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 203–216, 2018.

[16] Y.-H. Lai, C.-C. Chuang, and J.-H. R.Jiang, "A general framework for efficient performance analysis of acyclic asynchronous pipelines," *Proceedings of International Conference on Computer-Aided Design*, pp. 736–743, 2015.

[17] M. Najibi and P. A. Beerel, "Deriving performance bounds for conditional asynchronous circuits using linear programing," *18th IEEEInternational Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 9–16, 2012.

[18] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 385–418, 2004.

[19] M. Najibi and P. A. Beerel, "Performance bounds of asynchronous circuits with mode-based conditional behavior," in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pp. 9–16, IEEE, 2012.

[20] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the sixth MIT conference on Advanced research in VLSI*, AUSCRYPT '90, (Cambridge, MA, USA), pp. 263–278, MIT Press, 1990.

[21] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 102–109, IEEE, 2017.

[22] R. Manohar and Y. Moses, "Analyzing isochronic forks with potential causality," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pp. 69–76, IEEE, 2015.

[23] A. Dan, R. Manohar, and Y. Moses, "On using time without clocks via zigzag causality," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 241–250, ACM, 2017.

[24] J. V. Manoranjan and K. S. Stevens, "Qualifying relative timing constraints for asynchronous circuits," *22nd IEEEInternational Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 91–98, 2016.

[25] R. Manohar and A. Martin, "Slack elasticity in concurrent computing," *Lecture Notes in Computer Science*, Jan 1998.

[26] R. Karmazin, S. Longfield, C. T. O. Otero, and R. Manohar, "Timing driven placement for quasi delay-insensitive circuit," *Proceedings of 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 45–52, 2015.

[27] N. E. Young, R. E. Tarjan, and J. B. Orlin, "Faster parametric shortest path and minimum-balance algorithms," *Networks*, vol. 21, no. 2, pp. 205–221, 1991.

[28] R. M. Karp and J. B. Orlin, "Parametric shortest path algorithms with an application to cyclic staffing," *Discrete Applied Mathematics*, vol. 3, pp. 37–45, 1981.

[29] 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.

[30] 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.

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

[32] F. Akopyan, C. Tadeo, O. Otero, and R. Manohar, "Hybrid synchronous-asynchronous tool flow for emerging vlsi design," in *IEEE International Workshop on Logic Synthesis*, 2016.