Timing Path-Driven Cycle Cutting for Sequential Controllers

WILLIAM LEE, University of Utah
VIKAS S. VIJ, Intel Corp.
KENNETH S. STEVENS, University of Utah

Power and performance optimization of integrated circuits is performed by timing-driven algorithms that operate on directed acyclic graphs. Sequential circuits and circuits with topological feedback contain cycles. Cyclic circuits must be represented as directed acyclic graphs to be optimized and evaluated using static timing analysis. Algorithms in commercial electronic design automation tools generate the required acyclic graphs by cutting cycles without considering timing paths. This work reports on a method for generating directed acyclic circuit graphs that do not cut the specified timing paths. The algorithm is applied to over 125 benchmark designs and asynchronous handshake controllers. The runtime is less than 1 second, even for even the largest published controllers. Circuit timing graphs generated using this method retain the necessary timing paths, which enables circuit validation and optimization employing the commercial tools. Additional benefits show these designs are on an average a third in size, operate 33.3% faster, and consume one-fourth the energy.

CCS Concepts: • **Hardware → Application specific integrated circuits**

Additional Key Words and Phrases: Asynchronous, design automation

ACM Reference Format:
William Lee, Vikas S. Vij, and Kenneth S. Stevens. 2016. Timing path-driven cycle cutting for sequential controllers. ACM Trans. Des. Autom. Electron. Syst. 21, 4, Article 64 (June 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2893473

1. INTRODUCTION

Asynchronous architectures and design methodologies are an excellent means of generating fast, low-power circuit topologies. Handshake protocols and their associated controllers are used to implement the timing and sequencing of the design. The complexity of developing a complex asynchronous design can be simplified due to the modularity and composability of these controllers.

All asynchronous circuits have cycles. Cycles in these graphs come from three primary sources. First, the handshake controllers themselves are often sequential controllers. The state memories in these sequential controllers are commonly implemented using combinational gates with feedback. Second, the basic nature of asynchronous handshake protocols produces cyclical feedback loops; the acknowledge signal creates a circuit cycle that responds to the request. This cyclical ring of request acknowledge logic gates produces an oscillator that dictates the operational frequency of each asynchronous pipeline stage. Third, cycles are created in system-level architectures where data is fed back to previous pipeline stages.

Many asynchronous design approaches leverage commercial electronic design automation (EDA) tools to optimize and validate power, performance, and timing correctness, and to perform timing-driven optimizations for synthesis and place and route [Ligthart et al. 2000; Kondratyev and Lwin 2002; Taubin et al. 2007; Smirnov 2009;

This material is based upon work supported by the National Science Foundation under Grant Number 1218012 and the Semiconductor Research Corporation under Grant Number 2235.001.
Authors' addresses: W. Lee, V. S. Vij, and K. S. Stevens, 50S Central Campus Drive, MEB Room 2110, Salt Lake City, UT 84112; emails: william.lee@utah.edu, vikasvij@gmail.com, kstevens@ece.utah.edu.

DOI: http://dx.doi.org/10.1145/2893473

Stevens et al. 2009; Beerel et al. 2011]. The timing-driven algorithms in commercial EDA tools employ fast static timing analysis (STA) algorithms, which require circuit models to be represented as directed acyclic graphs (DAGs). Unfortunately, the sequential nature of asynchronous controllers and design approaches results in numerous topological feedback paths, presenting a fundamental challenge in employing commercial EDA tools.

Sequential asynchronous circuits must be modeled with acyclic timing graphs to employ commercial EDA tools. This can be achieved with two fundamental elements that are directly supported by the commercial EDA flows in the industry standard Synopsys design constraints (sdc) format. First, circuit timing can be defined with a set of path-based timing constraints and their delay targets. The paths are identified by timing endpoints. Second, a set of timing cuts can be defined that model the native cyclic timing graph of the design as an acyclic graph without cutting the path-based timing constraints. Timing cuts can also be employed to remove false paths.

Following are three key observations for supporting cyclical circuits in the DAG-based commercial tools with the above two constraint sets.

(1) If a cyclic circuit is given directly to a commercial EDA tool, a DAG representation will automatically be created without respect to timing paths. If a timing path is cut, it will not be employed in the timing-driven algorithms of the commercial EDA tools. This is true for both optimization and validation. The cut timing paths are considered to be vacuously true and are not reported as failures because they cannot be evaluated.
(2) A DAG representation of a sequential circuit cannot directly model all of the necessary timing of a sequential circuit. For instance, a handshake cycle in an asynchronous design can be implemented with a controllable ring oscillator that has a target frequency based on the function of the pipeline stage. A DAG-based model will cut the ring, and thus, one cannot give a frequency-based delay target to optimize the design or validate its performance. Thus, sequential timing constraints of an asynchronous design may require two or more timing runs to validate full cyclical timing paths.
(3) All design approaches that map sequential asynchronous designs to commercial EDA tools will require an additional methodology for cycle cutting, timing validation, and performance verification. The methodology and CAD tool reported here for creating DAGs are developed in such a way that it should be applicable to nearly all high-level methodologies that address system-level timing for asynchronous designs.

Providing a DAG that supports a timing model of a sequential circuit to commercial EDA tools results in several benefits. In some design approaches, such as bundled data design, a nonfunctional design results if timing constraints do not hold. This can occur if timing constraints are cut in creating the DAG. Likewise, commercial EDA cannot be employed to evaluate circuit timing unless the graph is represented as a DAG and the timing paths used for evaluation are not cut. Providing a DAG with associated timing constraints enables the commercial EDA tools to better optimize designs, resulting in significant improvement in the power, area, and performance of such designs. An example across a complete family of controllers demonstrates that, on average, the circuits are a third in size, consume one-fourth the energy, and operate 33% faster.

The primary contributions of this article are the following. The fundamental problem is addressed of creating a DAG timing model for sequential controllers and systems that does not cut the set of timing paths used for design synthesis, optimization, and validation. This is the first published work to do so. Path-based timing constraints are identified by timing endpoints, just as is done using sdc constraints in the commercial EDA tools. This work also takes as input a path defined by timing endpoints. This simplifies this work's translation of the timing directives into commercial EDA tool constraints. An algorithm is implemented that creates cycle cuts that preserve the specified path-based timing constraints. This work also introduces the concept of identifying paths that must be cut in a design in order to remove cycles formed by

netlist connectivity external to the controller, along with its associated algorithm. This algorithm is also used to remove false paths in the design. The tool uses vectorless graph traversal algorithms similar to commercial static timing analysis approaches. It assumes that the design has been technology mapped to specific gate implementations, and inputs the Verilog used in the design. The CAD algorithm is intended to be applied to single asynchronous sequential controllers, making them the focal point for timing paths and system-level cycle cuts. The combination of vectorless algorithms and controller-based focus makes this CAD tool applicable to most, if not all, asynchronous design methodologies and styles. The tool writes out sdc constraints that are directly supported by the commercial EDA tools. The tool reports coverage and the quality of the results of the cycle-cutting algorithm.

A timed burst-mode protocol and its circuit realization are employed as an example for this article. The CAD algorithms in this article are applied to several examples and compared against a commercial EDA tool. Results are reported on applying the algorithm to 131 separate asynchronous sequential control circuits and to eight benchmark designs. The comparison of these designs are performed with respect to forward latency, backward latency, cycle time, area, power, and energy per token. The results are also analyzed for quality by ensuring that the cycle cuts produce a DAG, false paths have all been cut, and the number of gates that have no timing path passing through them are reported. Having at least one timing path passing through each gate in a design results in the gates being power and performance optimized based on the timing path constraints.

2. RELATED WORK

Combinational cycles are generally associated with sequential circuit designs like asynchronous circuits. Cycles can also be present in combinational logic, and some cyclic combinational circuits have been shown to substantially reduce area [Riedel and Bruck 2003]. Since algorithms in EDA tools require acyclic timing graphs, the problem of finding cycles and analyzing the combinational nature of circuits with cycles has been investigated [Malik 1994]. Algorithms that generate an equivalent acyclic combinational circuit that reproduces all the combinational behavior of the original cyclic circuit have been developed [Shiple et al. 1996; Edwards 2003; Neiroukh et al. 2008]. These approaches cannot be applied to sequential circuits because they change the sequential behavior when state-holding feedback of a circuit are removed. In order to support general sequential circuits built as combinational logic with feedback, the cyclic circuit must be represented as a DAG without modifying its structure or behavior.

The work that is most closely related applies cycle cutting to the testing of digital circuits with feedback [Filippovich 1973]. This is formulated as a covering problem where the set of paths form the cycles, the solution is to find the minimal number of paths that cut all the cycles. The drawback of this approach is similar to the algorithms in current commercial CAD tools that also cut cycles. A set of cycle cuts, even if they are minimal, will create a DAG, but timing-driven optimizations cannot be performed because timing paths are cut.

A core function of the algorithms in this work is to identify cycles and paths in a circuit graph. Reconvergent paths and circuit cycles are particularly problematic. Indeed, it has been shown that a circuit can have an exponential number of paths based on the number of gates. This path explosion has proven to be particularly challenging for the delay fault testing community. While path explosion is problematic in some domains, it has not been demonstrated to be a problem in this application. The sequential modules being evaluated have normally been designed to minimize hazards that can often be a byproduct of reconvergent paths. Our algorithms are applied to single sequential controllers that contain fewer than 100 gates. We have applied our tool to the largest published sequential controller designs that tax the limits of what can be synthesized. For all but one circuit, the runtimes are less than 1 second in the exhaustive search mode. Pipeline controllers used in nearly all commercial and academic design are more closely represented by the 131 controller set used in our example set.

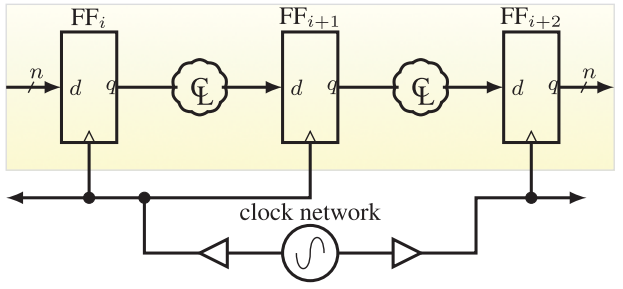

Fig. 1. Clocked design. Frequency and data path delay of first pipeline stage is constrained by $FF_i/clk\uparrow_j \mapsto FF_{i+1}/d + margin \prec FF_{i+1}/clk\uparrow_{j+1}$ Similar asynchronous bundled data pipeline shown in Figure 2.

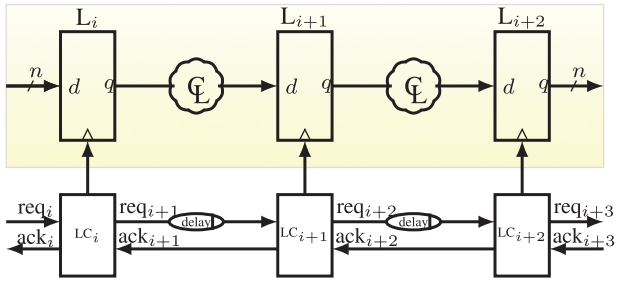

Fig. 2. Timed (bundled data) handshake design. Each $req_i\uparrow$ handshake on LC_i indicates new data is presented to pin d of L_i. Delay sized by relative timing constraint $req_i\uparrow \mapsto L_{i+1}/d + margin \prec L_{i+1}/clk\uparrow$.

3. BACKGROUND

3.1. Timed Asynchronous Design in Commercial EDA Tools

Delay-insensitive (DI) asynchronous design approaches ensure design correctness regardless of circuit delay. Thus, circuit timing for DI designs largely becomes a power, area, and performance tradeoff. However, in timed asynchronous design methodologies, circuit timing is critical to design correctness. Bundled data asynchronous design is one such approach. Traditional clocked combinational data path logic is employed much like in clocked design (as in Figure 1), and the handshake controllers are used to match the data path delays and sequence pipeline stages that support stalling.

The handshake controllers in a bundled data asynchronous design (the LC blocks in Figure 2) have three tasks. First, they perform flow control between communication channels. Second, they are used as the local timing reference to determine the operation frequency of the design. Each handshake channel implements a ring oscillator with an odd number of signal inversions that results in a particular oscillation frequency. In the absence of a stall, the delays through the controllers and communication channels determine the native cycle time of that pipeline stage. Third, these controllers produce the clock signal that drives the local latches and/or flip-flops in the data path. Delay differences between the clock drivers of adjacent handshake controllers result in wasted performance.

Delays through bundled data asynchronous handshake controllers comprise two of their three basic tasks. Unlike DI design, many of these bundled data timing constraints must hold for design correctness. For example, nearly all burst-mode controllers have internal timing constraints that must hold for correctness because the state variables can change concurrently with the outputs. Timing constraints internal to a controller are required to ensure correct circuit operation. Therefore, having a DAG graph representation where timing paths are not cut is a requirement for circuit optimization and validation for some asynchronous design styles that use commercial EDA tools.

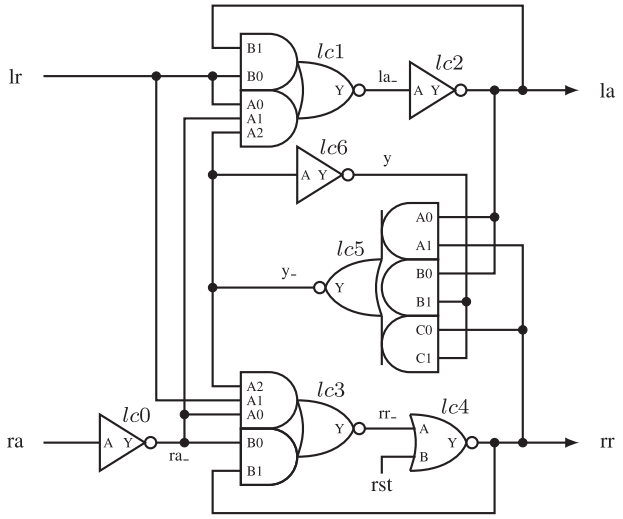

Fig. 3. LC circuit implementation.

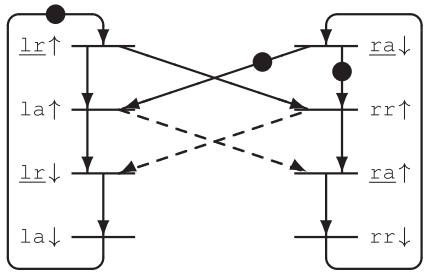

Fig. 4. Petri net specification of linear control.

LEFT	=	lr.c1.la.c2.lr.la.LEFT
RIGHT	=	c1.rr.c2.ra.rr.ra.RIGHT
SPEC	=	(LEFT | RIGHT) \ { c1, c2 }

Fig. 5. CCS specification of LC block.

We use the most challenging approach as the primary example in this paper by employing the burst-mode controller of Figure 3 in a bundled data pipeline. This is a four-phase return to zero handshake protocol with data valid on the rising edge of the request (lr) signal [Cortadella et al. 2002]. Figures 4 and 5 show the Petri net and CCS specifications of the protocol [Milner 1989; Chu 1987]. This is a timed protocol (the dashed arcs in Figure 4 constrain inputs), implementing a burst-mode specification [Coates et al. 1993]. This protocol is chosen as an example because it requires protocol-level burst-mode timing constraints between LC modules. The circuit in Figure 3 implements this protocol and is mapped to static gates from the Artisan 65nm library.

3.2. Circuit Representation

Timing analysis used in commercial EDA tools employ vectorless static timing analysis (STA) algorithms. This is accomplished by modeling the circuit as a directed acyclic graph, which largely ignores the functional behavior of the circuit. The algorithms

developed in this article also use only the structural netlist, identifying paths using graph traversal algorithms.

The circuit is represented as a directed graph $G = (V, E)$, where each gate, primary input, and primary output $v_i \in V$ is a vertex (node) of the graph, and edges $e_i = (v_x, v_y) \in E$ define connectivity between primary input and gate output vertices to primary outputs and gate input vertices of the design. A *simple path* $s_i = [v_j, \ldots, v_k] \in S$ is a sequence of vertices that are connected in G, where repetitions of vertices are not allowed. A *cycle* $c_i = [v_j, \ldots, v_k, v_j] \in C$ is a sequence of vertices that are connected in G, where the first and last vertex are the same, but no other vertex in the sequence is repeated. A *timing endpoint* $t_i = (v_x, v_y) \in T, v_x \neq v_y$ is a pair of vertices. In this article, we represent timing endpoints t_i by the pair of nodes (v_x, v_y) or the relationship $(v_x \rightarrow v_y)$.

Each primitive gate in a cell library used to build a design, such as an inverter or 2-input NAND gate, will have one or more input and one or more output. Each input for gate v_y will be associated with one or more edge $(v_x, v_y) \in E$. Each output for gate v_x will be associated to one or more edge $(v_x, v_y) \in E$. The path to edge function $P2E(G, S \vee C) \Rightarrow E$ translates a vertex-defined path (or cycle) set into a set of edges that are used to traverse the path (cycle) set.

In the circuit of Figure 3, the two simple paths $[lr, lc3, lc4, rr]$ and $[lr, lc1, lc2, lc5, lc3, lc4, rr]$ connect the timing endpoint pair $(lr \rightarrow rr)$. This circuit contains several cycles, such as $[lc1, lc2, lc1]$. The set of edges covered by this cycle are $P2E(G, \{[lc1, lc2, lc1]\}) = \{(lc1, lc2), (lc2, lc1)\}$.

Two data structures Θ and Φ are sets of timing endpoint pairs. Set Θ contains timing endpoints that are used to define the true timing paths in the sequential circuit that need to be retained uncut. Set Φ holds timing endpoints that are used for cutting false paths, as well as local timing endpoints that will remove cycles formed by connectivity outside of the controller graph G. Mapping functions $GCP(G, T) \Rightarrow S$ and $P(G, T) \Rightarrow S$ are defined, which map from a set of timing endpoints to a set of paths. Function $GCP()$ creates the set of greatest common paths between the timing endpoint set (see Section 3.3). Function $P()$ creates the complete set of simple paths between the timing endpoints in graph G.

3.3. Greatest Common Path between Timing Endpoints

A GCP is a minimal structural path between a pair of timing endpoints. If the node sequence in a shorter path is contained in a longer path between the same endpoints, then the longer path is not a GCP. Let G represent the circuit of Figure 3. Graph G has two simple paths between timing endpoints lr and rr that do not contain cycles. Therefore, by setting Θ to $\{(lr, rr)\}, P(G, \Theta) = \{[lr, lc1, lc2, lc5, lc3, lc4, rr], [lr, lc3, lc4, rr]\}$. The greatest common path is calculated as $GCP(G, \Theta) = \{[lr, lc3, lc4, rr]\}$. The longer path is not a GCP because it is not minimal; it contains the shorter path. Note that paths that contain cycles will not be GCPs.

3.4. Path Identification from Timing Endpoints

Our specification approach does not identify or enumerate individual paths through a circuit; rather we employ timing endpoints to correctly identify paths that should not be cut (true paths) as well as false paths and paths that must be cut to remove cycles external to the controller. This approach is chosen because it is more compatible with the `sdc` path timing specifications `set_max_delay` and `set_min_delay` that use timing endpoints. It also allows multiple true or false paths to be identified with a single pair of timing endpoints.

3.5. Cutting the Timing Graph

The liberty timing file that is used by the commercial EDA tools defines the delay from inputs to outputs of each primitive gate. For example, the NOR gate in Figure 3 has a timing path from pin A to pin Y and from pin B to pin Y. These liberty gate-level timing

paths can be cut with the sdc command `set_disable_timing`. This is the mechanism we employ for cutting timing paths and cycles in a circuit, and the tool outputs its results in the sdc format.

Assume we want to cut the Figure 3 cycle $[lc3, lc4, lc3]$. This can be accomplished by cutting either of the two edges $P2E(G, \{[lc3, lc4, lc3]\}) = \{(lc3, lc4), (lc4, lc3)\}$. Removing edge $(lc3, lc4)$ is implemented by disabling the liberty timing edge (A, Y) in gate $lc4$. We write this out as the sdc constraint `set_disable_timing -from A -to Y` for the $lc4$ gate in the design. Using this mechanism, our tool can create a timing graph DAG that is supported by commercial EDA.

Note that edge $(lc4, lc3)$ is the preferable arc to cut in the above example to remove the cycle from the timing graph. If edge $(lc3, lc4)$ is cut, the circuit has no timing path from the primary inputs lr and ra to the primary output rr.

3.6. True and False Path Specification

3.6.1. True Path Specification. Many structural paths in a sequential design will be false paths that are not behaviorally sensitizable. True paths are the result of the logical sequential behavior of the circuit and how it responds to changes in the primary inputs and internal state. Since both STA and the algorithms employed in this tool are structural and ignore circuit behavior, a mechanism must be employed to specify the true and false paths in a design. True paths must be preserved, and false paths in a circuit must be cut. Otherwise, timing results will be incorrect and the quality of the timing optimizations performed by the commercial EDA tools will be significantly degraded. The timing endpoint sets Θ and Φ from Section 3.2 are used to identify the the true and false paths through these sequential circuits.

True timing paths are often directly identified as GCPs. As demonstrated in Section 3.3, there are two simple paths between the timing endpoints $(lr \rightarrow rr)$ for the circuit in Figure 3. However, only the shorter of the two paths, $[lr, lc3, lc4, rr]$, is a GCP. The GCP is also the true behavioral path through the circuit. Thus, placing the timing endpoints (lr, rr) into Θ correctly identifies the true path between lr and rr. The longer path is false because it cannot behaviorally occur in this design. Lowering rr is only sensitized by ra. For rr to rise, $ra\_$ and $y\_$ must be asserted and lr must rise. For the longer path to occur, $y\_$ would need to rise, which only occurs when la falls.

A GCP can identify multiple paths. Let G model the circuit of Figure 3, and set Θ to $\{(lr, lc5)\}$. Then $GCP(G, \Theta) = \{[lr, lc1, lc2, lc5], [lr, lc3, lc4, lc5]\}$. In this case, both of these paths are true paths through the circuit.

In some designs, a single GCP does not identify the true timing path(s) between the desired timing endpoints. In such cases, a set of timing endpoints must be used to identify the true path(s), where the transitive closure of the GCPs correctly covers the true path(s) and no other paths. For example, let G model the handshake controller of Figure 6, and set Θ to $\{(lr, rr)\}$. The $GCP(G, \Theta) = \{[lr, 3, 4, 8, rr], [lr, 0, 2, 4, 8, rr], [lr, 5, 6, 2, 4, 8, rr]\}$, but the last two paths are false. The timing endpoints $(lr \rightarrow rr)$, therefore, incorrectly identifies all three GCP paths as true. By selecting 3 as an intermediate timing endpoint, a set of two timing endpoints $\Theta = \{(lr, 3), (3, rr)\}$ now define the true path because $GCP(G, \Theta) = \{[lr, 3], [3, 4, 8, rr]\}$. Each of these endpoints identifies a single GCP path, whose transitive closure is the correct true path between lr and rr. (One can apply transitive closure on all true paths to reduce the number of paths, since each path will not be cut. Therefore, $\{[lr, 3], [3, 4, 8, rr]\} = \{[lr, 3, 4, 8, rr]\}$.

3.6.2. False Path Identification. Simply identifying true paths through a design is not sufficient to ensure correct timing evaluation of the design. The false paths must also be identified and then cut. For example, let G model the circuit in Figure 3, and set Θ to $\{(lr, rr)\}$. The $GCP()$ function identifies the shorter of the two paths as true. This path will not be cut in our algorithm. Assume that a DAG is formed to represent this circuit, and the longer simple path $[lr, lc1, lc2, lc5, lc3, lc4, rr]$ also remains uncut. Evaluating the maximum delay between timing endpoints lr and rr of this circuit using static timing analysis will return the delay of the longer false path, rather than the true

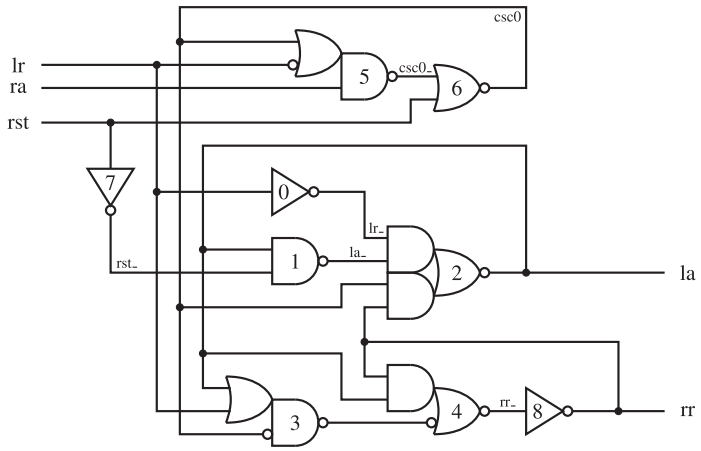

Fig. 6. Handshake Controller L222 ◦ R2242 synthesized with Petrify.

path $[lr, lc3, lc4, rr]$, resulting in an incorrect delay value. Thus, we must identify as false all simple paths between true timing endpoints that are not true paths. This is accomplished by placing the timing endpoints of the complete path into set Φ.

Only the first and last timing endpoints of the transitive closure of timing endpoint pairs in Θ will be included in Φ when a true path cannot be represented as a single GCP in Θ. For the circuit in Figure 6, two timing endpoint pairs are required to identify that path: $\Theta = \{(lr, 3), (3, rr)\}$. In this case, Φ is set to $\{(lr, rr)\}$, which identifies all false paths between lr and rr, and $P(G, \Phi)$ returns the path set $\{[lr, 3, 4, 8, rr], [lr, 0, 2, 4, 8, rr], [lr, 0, 2, 3, 4, 8, rr], [lr, 5, 6, 2, 4, 8, rr], [lr, 5, 6, 3, 4, 8, rr], [lr, 5, 6, 2, 3, 4, 8, rr]\}$.

3.6.3. False Path Specification. In our algorithm, true paths are specified by the GCPs of the timing endpoint pairs in Θ. False paths are identified by all paths in Φ. Let G model the Figure 3 circuit, set Θ to $\{(lr, rr)\}$, and Φ to $\{(lr, rr)\}$. The true path is specified as $GCP(G, \Theta) = \{[lr, lc3, lc4, rr]\}$. False paths are identified as all paths between the timing endpoints, so $P(G, \Phi) = \{[lr, lc1, lc2, lc5, lc3, lc4, rr], [lr, lc3, lc4, rr]\}$ are candidate false paths. Since the true path is also a valid path between the timing endpoints, we perform set subtraction to specify the correct set of false paths that must be cut: $P(G, \Phi) - GCP(G, \Theta) = \{[lr, lc1, lc2, lc5, lc3, lc4, rr]\}$.

This condition also holds for paths constructed from the transitive closure of GCP constraints. Let G model the Figure 6 circuit, set Θ to $\{(lr, 3), (3, rr)\}$, and Φ to $\{(lr, rr)\}$. Now $GCP(G, \Theta) = \{[lr, 3], [3, 4, 8, rr]\}$, which can be simplified to $\{[lr, 3, 4, 8, rr]$ through transitive closure, and $P(G, \Phi) = \{[lr, 0, 2, 4, 8, rr], [lr, 0, 2, 3, 4, 8, rr], [lr, 5, 6, 2, 4, 8, rr], [lr, 5, 6, 3, 4, 8, rr], [lr, 5, 6, 2, 3, 4, 8, rr], [lr, 3, 4, 8, rr]\}$. The set of false paths that must be cut are $P(G, \Phi) - GCP(G, \Theta) = \{[lr, 0, 2, 4, 8, rr], [lr, 0, 2, 3, 4, 8, rr], [lr, 5, 6, 2, 4, 8, rr], [lr, 5, 6, 3, 4, 8, rr], [lr, 5, 6, 2, 3, 4, 8, rr]\}$.

3.7. Classification of Cycles and Cycle Cutting

Asynchronous bundled data design can be partitioned into two facets, as shown in Figure 2: the combinational data path logic and a clock control network. The clock control network implements handshaking protocols with sequential controllers (LC) that interact with upstream and downstream pipeline stages. These handshake controllers are small design blocks normally consisting of less than 20 logic gates. Most bundled data architectures typically use less than three or four different handshake protocols and controllers throughout the entire design.

Therefore, if we can isolate cycle cutting to the handshake controllers, then we can create a DAG for an entire design by performing cycle-cutting algorithms on a small

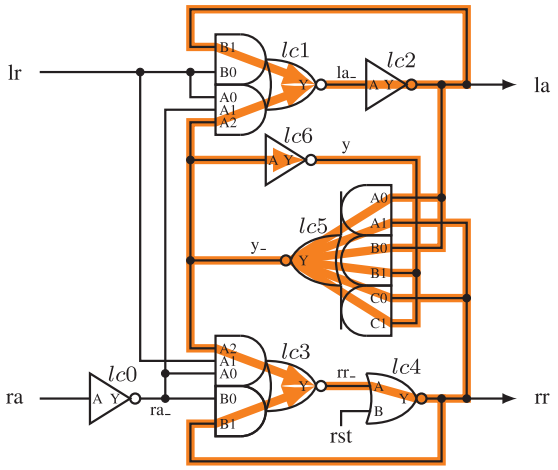

Fig. 7. LC circuit with the eight local cycles highlighted. Three cycles pass through $lc1$, three through $lc3$, and two cycles through $lc6$.

Fig. 8. External cycles formed from handshake channels are highlighted.

set of simple sequential designs. This can be accomplished by partitioning the three types of cycles described in Section 1 into two partitions as follows.

(1) *Local Cycles:* The first class of cycles are those that are entirely contained in a controller. These cycles are created in sequential controllers in order to implement state holding logic. The eight cycles for our example timed controller are highlighted in Figure 7.
(2) *External Cycles:* The other two cycles in an asynchronous architecture are classified as external because they are formed from interconnections outside of the handshake controller. These are of two types. Handshake channels create cycles for the request-acknowledge protocol. Figure 8 highlights some of these cycles such as $[LC_0/rr, LC_1/lr, LC_1/la, LC_0/ra, LC_0/rr]$. The third class of cycles are formed from cycles in architecture. If the leftmost channel and rightmost channel in Figure 8 are connected, an architectural cycle would be formed.

A point of convergence for all external cycles exists in the sequential handshake controller. Therefore, if we can support external cycle cutting in our algorithms, we can create a DAG for an entire asynchronous architecture regardless of its size by simply performing cycle cutting on the small set of handshake controllers used in the design. This must only be done once per controller, making this a very efficient operation.

The algorithms in this tool will automatically cut local cycles because they can be identified by evaluating the circuit graph G. External cycles can only be cut if they are identified by the user. This is performed by identifying all subsections of all external cycles that pass through the controller as false paths. These paths are identified by be placing the timing endpoints of the path subsections into the set Φ. For example, in a linear pipeline such as that shown in Figure 8, all external cycles can be removed by placing the timing endpoints (ra, rr) into set Φ. Such an approach allows a single cut path constraint local to the pipeline controller to remove all handshake channel cycles in a design. The location and expression of these cycles is dependent on the high-level design methodology employed.

Three observations can be made based on cycles external to the controllers being evaluated. First, placing a path that should be cut due to an external cycle into Φ does not *guarantee* the path will be cut. If there exists a path between the timing endpoints that is fully covered by GCPs in Θ, the path will not be fully cut. This will result in combinational cycles that remain in the timing graph of an architecture. However, this condition will be reported as an error by our tool.

Second, each handshake controller will have its cycle cut values generated independently. Therefore, if multiple different controllers are used in a single design and they employ a different handshake cut methodology, then cycles can remain in an architecture. (For instance, assume one controller cuts the handshake cycle with the (lr, la) endpoints and another with the (ra, rr) timing endpoints. If both are used in a design, the system can have cycles left uncut.) Thus, the application of this tool is dependent on correctly conforming to the system-level methodology employed, and ensuring that it is applied uniformly to the control modules.

Finally, while this method helps support different circuit and timing validation methodologies, it remains dependent on the high-level models employed. External cycles created by data path feedback may not be directly supported by adding a cut path in Φ for all timed asynchronous circuit and timing methodologies.

4. RULES FOR TIMING PATH-DRIVEN CYCLE CUTTING

This tool will create a DAG where true paths remain uncut and false paths and external cycles are cut when a correct set of timing paths are provided. If the set of paths are inconsistent and a DAG cannot be generated, or external cycle paths cannot be cut, an error is raised.

4.1. Rule Set

Paths are identified by a sets of timing endpoints. The timing paths consist of (a) a set Θ of timing endpoints that identify the true paths where the paths between the endpoints are GCPs, and (b) a set Φ of cut paths that cut all paths between the timing endpoints that are not GCPs in Θ.

RULE 1. All timing paths that are GCPs between timing endpoints in set Θ are considered true paths and must remain uncut.

RULE 2. All cycles in the controller must be cut.

RULE 3. All paths between timing endpoints in the cut path set Φ that are not GCPs from set Θ must be cut.

If at least one structural path exists between pairs of timing endpoints $\forall(v_x, v_y) \in \Theta$, then there exists at least one GCP between each pair of timing endpoints. If all timing paths covered by a GCP cannot be cut, then at least one timing path will be preserved for each pair of timing endpoints. The GCP(s) defines the true timing path(s) for a pair of timing endpoints. This rule takes precedence over the other rules.

4.2. Additional Information

4.2.1. True Path Identification. Providing the correct values for Θ that identify true paths through a sequential circuit is critical to the correct DAG generation, as it will directly impact design optimization. The method used to identify the true paths will be based on the particular design flow. In our work, we have a formal verification engine that automatically produces timing constraints that are required for the circuit to operate correctly [Xu and Stevens 2009]. This engine has been extended to support identification of true paths through the sequential circuits being verified for performance and timing.

4.2.2. Full Design Module Optimization. Best circuit optimization occurs with timing-driven design, as will be shown in Section 7. Ideally, every gate will have a timing path passing through it to define the intended delay bounds. An **orphan** is defined as a

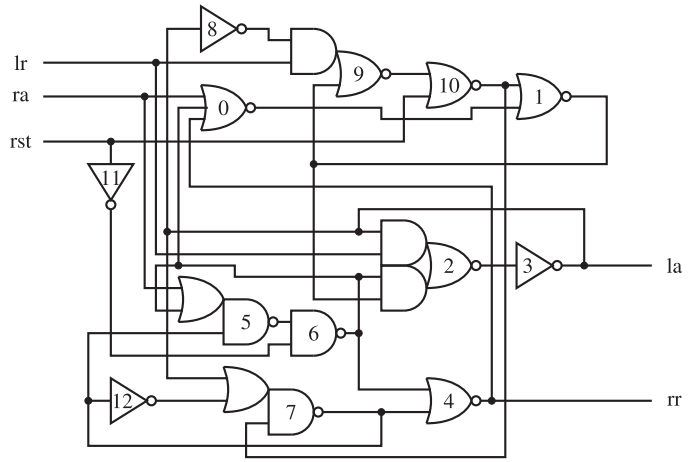

Fig. 9. Handshake Controller L400 ◦ R0000 synthesized with Petrify.

gate that does not have a true timing path passing through it defined by Rule 1. The optimization of handshake controllers will require multiple timing paths through the design to minimize or eliminate orphans and to create timing paths between primary inputs and outputs of a sequential controller.

4.2.3. Consistency and Rule Correctness. If set Θ consists of a single pair of timing endpoints, then rules 1–3 are necessary and sufficient to create a DAG with only true timing paths passing through the design for optimization. Unfortunately, these rules cannot all be guaranteed to hold when more than one pair of timing endpoints exist in Θ.

If true paths cover a cycle, rule 2 will fail, permitting the cycle to remain uncut. For example, the controller in Figure 9 contains 18 cycles, 26 cut paths, and 3 GCP paths. The true path for timing endpoints (lr, rr) is $[lr, 9, 10, 1, 2, 3, 7, 5, 6, 0, 1, 9, 10, 7, 4, rr]$ (which contains a cycle), while the true path for (lr, la) is $[lr, 9, 10, 1, 2, la]$. Setting Θ to $\{(lr, 10), (10, 1), (1, la), (3, 5), (5, 0), (0, 10), (10, rr)\}$ is sufficient to represent the true paths. However, when this is done, the algorithm reports that three cycles remain uncut including [0, 1, 2, 3, 7, 5, 6, 0], [0, 1, 2, 3, 7, 5, 6, 0], and [1, 9, 10, 1]. This is due to GCPs that overlap to cover all 3 cycles in the true path.

Likewise, a false path in a design may be covered by a set of GCPs. Therefore, any uncut path between timing endpoints in Φ that are not true paths are reported as an error by the tool.

4.2.4. Providing Correct Endpoint Sets. The generation of a complete and consistent set of timing endpoint sets for Θ and Φ that has no orphans, correctly specifies all true timing paths using GCPs, and correctly identifies all false paths, is outside of the scope of this article. The implementation of the algorithms in this article reports on the quality of the timing endpoint sets provided. If there is no solution that adheres to all of the rules, then a report is generated noting the violations.

5. ALGORITHM

An algorithm that automates the process of generating cycle cuts for sequential circuit modules is presented. It obeys the rules in Section 4.1. Four inputs are provided to this tool: (1) A sequential circuit specified in structural Verilog that has been technology mapped to a set of library gates, (2) a data structure L that defines the input and output pin names and mapping for each gate in the structural Verilog library, (3) timing endpoint set Θ that defines true paths, and (4) Timing endpoint set Φ that defines false paths and external cycle cuts.

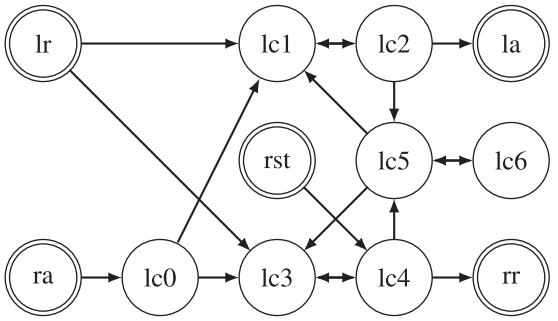

Fig. 10. Graph representation for LC circuit.

The structural Verilog circuit description and cell library data structure L are parsed and stored as an adjacency list $G = (V, E)$. Figure 10 demonstrates a graph representation of the Figure 3 circuit. The true paths are calculated by $GCP(G, \Theta) = \Theta_p$, and the false paths are calculated as $P(G, \Phi) - \Theta_p = \Phi_p$. All cycles present in the circuit module are identified using a depth-first search algorithm on G and stored in data structure C.

A set of edges E_c that are cut candidates are generated from the paths in Φ_p and C as $P2E(\Phi_p \cup C) - P2E(\Theta_p) = E_c$. A covering table is then created that maps paths in Φ_p and C to cut candidates E_c. A quality metric is defined for the final DAG solution, with the best solution having all cycles cut, all false paths cut, with no gate orphans. Two algorithms are now applied to determine the edges to be cut: a polynomial time greedy heuristic and an exponential algorithm, which finds the best quality solution.

A set of cycle cut constraints in the sdc format are output as well as a list of violations to the rules. These constraints can then be passed through the synthesis and place and route flows to allow the circuits to be automatically power and performance optimized by commercial EDA tools and then validated for timing correctness.

More details on certain specific steps of the algorithm are described.

5.1. Finding All the Cycles Present in the Circuit

A brute-force algorithm is implemented to find all the local cycles in a circuit. A depth-first search is performed for each vertex $v_i \in V$ in the adjacency list G to find paths that return to the vertex v_i. If such a path exists, then the stack that stores the trace is recorded as a cycle.

The LC controller in Figure 7 highlights the eight circuit cycles shown below.

Cycle 0	[$lc1$ $lc2$ $lc1$]	Cycle 4	[$lc3$ $lc4$ $lc5$ $lc3$]
Cycle 1	[$lc1$ $lc2$ $lc5$ $lc1$]	Cycle 5	[$lc3$ $lc4$ $lc5$ $lc3$]
Cycle 2	[$lc1$ $lc2$ $lc5$ $lc1$]	Cycle 6	[$lc5$ $lc6$ $lc5$]
Cycle 3	[$lc3$ $lc4$ $lc3$]	Cycle 7	[$lc5$ $lc6$ $lc5$]

Note that Cycles 1 2 have the same gate sequence because the output of gate $lc2$ goes into two separate inputs of gate $lc5$. This can be verified from the adjacency list with gate $lc5$ appearing twice as successor of gate $lc2$. Similarly, Cycles 4 and 5 and also Cycles 6 and 7 have the same gate sequence but are two separate cycles. This implies that two timing cuts may be necessary to cut a single cycle.

The fanout and fanin of paths through the pins of the gate (or through independent gates) can result in an exponential number of paths and cycles in a design. Thus, the complexity of finding paths and cycles is theoretically exponential based on the number of vertices (gates) in a design. Fortunately, sequential controllers generally have few gates (typically less than 20). Finding the paths and cycles for the circuits under investigation result in small run times, even with fanin and fanout as observed above.

5.2. True Timing Path Generation

A depth-first search is performed to generate all paths between the timing endpoints in sets Θ and Φ specified by the user. The paths from sets of timing endpoints in set Θ are then pruned to the GCPs.

A set of four timing endpoints are employed as true timed paths for the circuit of Figure 3 and stored in Θ. The constraints and GCPs for Θ are shown here:

$$\begin{array}{ll} \text{lr} \rightarrow \text{la} & [lr, lc1, lc2] \times 2 \\ \text{lr} \rightarrow \text{rr} & [lr, lc3, lc4] \\ \text{ra} \rightarrow \text{la} & [ra, lc0, lc1, lc2] \\ \text{lr} \rightarrow \text{y} & [lr, lc1, lc2, lc5, lc6] \times 4 \quad [lr, lc3, lc4, lc5, lc6] \times 2 \end{array}$$

5.3. Generating Cycle and False Path Cuts

Two algorithms have been implemented to generate cycle cuts:

—**V1:** This is a polynomial-time greedy approach in terms of number of cycles. The solution is created by cutting maximum occurring edges in the covering table.
—**V2:** This is an exponential time approach in terms of number of cycles, which searches through the complete list of solutions possible to find the highest-quality solution.

Quality metrics for the tool flow report the status whether all cycles are cut, if any false paths remain uncut, and if there are any orphaned gates without a timing path passing through them. There is no need to create a minimal set of cuts; what matters is that the "right" set is created that does not violate any of the rules in Section 4.1.

The basis of both algorithms is the same. Up to this point, all false paths, cycles, and true timing paths have been defined. The problem of generating the false path and cycle cuts is converted into a covering problem for which a covering table is generated. In this implementation, the cycles are cut first, then the false paths. They could be processed concurrently as well.

Paths in the cycle set C become rows of the table. The columns are the edges in set $P2E(C) - P2E(\Theta_p)$. Only the edges present in the cycles that can be cut are considered. All the edges that are present on a GCP are excluded because they cannot be cut due to Rule 1 precedence.

Edges that have the same source and destination gates are combined into a single column for the covering table even though they might generate multiple `set_disable_timing` constraints. Shorthand is used in the following table for the local cycles present in the circuit of Figure 3. Columns la_0, y_{-0}, y_{-1}, y, and rr_0 represent the edges $(lc2, lc1)$, $(lc5, lc1)$, $(lc5, lc3)$, $(lc6, lc5)$, and $(lc4, lc3)$, respectively.

	la_0	rr_0	y_{-0}	y_{-1}	y
[$lc1$ $lc2$ $lc1$]	√				
[$lc1$ $lc2$ $lc5$ $lc1$]			√		
[$lc1$ $lc2$ $lc5$ $lc1$]			√		
[$lc3$ $lc4$ $lc3$]		√			
[$lc3$ $lc4$ $lc5$ $lc3$]				√	
[$lc3$ $lc4$ $lc5$ $lc3$]				√	
[$lc5$ $lc6$ $lc5$]					√
[$lc5$ $lc6$ $lc5$]					√

After the generation of the covering table, the V1 algorithm selects the edge that cuts the maximum number of cycles (rows). Each selected edge is stored in the cut edge set X and removed from future consideration by removing that column in the covering table. One or more `set_disable_timing` constraints are written out, and all rows representing cycles that get cut by this edge are removed. The algorithm iterates through the table to find a solution by repeatedly selecting an edge that is present in the most rows, writing out the sdc constraint(s), and updating the table. The generation of the local cycle cuts end when there are either no more edges (some cycles were not cut) or there are no more rows (all cycles have been cut) in the table.

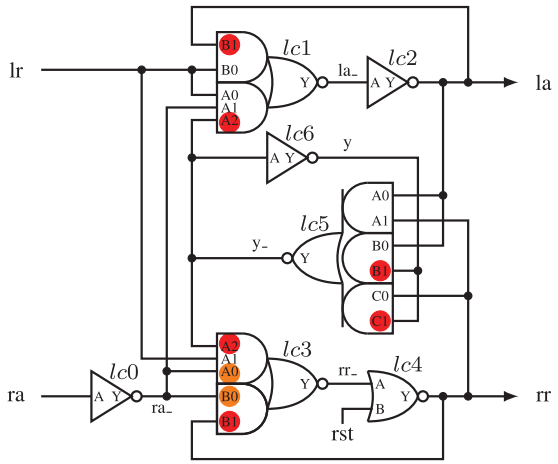

Fig. 11. LC circuit implementation showing local (red) and external (orange) timing arc cuts through the marked gates.

There are three edges that can result in cutting two cycles for the covering table of the circuit of Figure 3. Assume the rightmost edge y is selected. Cycles 6 and 7 are cut by removing this edge. This leads to the cycle count for the y edge to become zero, and hence that column is also removed. Continuing this process leads to the following cut set, which cuts all the cycles with six sdc constraints, graphically shown as red circles in Figure 11.

```
set_disable_timing -from A2 -to Y [find -hier cell *lc1]
set_disable_timing -from A2 -to Y [find -hier cell *lc3]
set_disable_timing -from B1 -to Y [find -hier cell *lc5]
set_disable_timing -from C1 -to Y [find -hier cell *lc5]
set_disable_timing -from B1 -to Y [find -hier cell *lc1]
set_disable_timing -from B1 -to Y [find -hier cell *lc3]
```

Algorithm V2 also creates the covering table, but goes one step further. It generates the complete list of solutions. A solution cost heuristic is employed to select the best solution. After generating each new solution, its cost is calculated and compared against the previous best solution. The solution with the minimum cost is selected as the best solution. Following is the cost heuristic, which gives priority to remaining uncut cycles. The constant K gives different weights for *number of uncut cycles* and *number of orphaned gates*. A orphaned gate exists when no timing path passes through the gate. This article assigns $K = 3$ to give higher cost for uncut cycles.

$$\begin{aligned} cost \;=\; & K \times number\ of\ uncut\ cycles \\ & + number\ of\ orphaned\ gates \end{aligned} \tag{1}$$

The search ends when the first solution with zero cost is found or when all the partial solutions have been generated. In the latter case, the partial solution with the lowest cost is returned.

After generating cycle cut constraints for all the local cycles, the cut path constraints are applied to remove false paths and external cycles. There are 24 simple false paths and external cycle cut paths generated from $P(G, \Phi)$. Ten of these paths are removed as being identical to true paths in $GCP(G, \Theta)$, leaving 14 remaining paths. Twelve of these are removed because a cycle cut previously performed removes an edge in these paths (six each by y_0 and y_1). The remaining uncut paths are stored in path set variable Φ_c. In this example, all of the false paths are cut as part of the cycle-cutting phase. The two remaining paths in Φ_c belong to the ($ra \rightarrow rr$) constraint used to cut external cycles.

The covering table is constructed with the rows being the remaining false path and external cycle cut paths that have not been cut as part of the cycle-cutting phase. The columns are calculated as $P2E(\Phi_c) - (P2E(\Theta_p) \cup X)$: the edges in the remaining false paths and external cycle cut paths that are not on a true path and have not already been removed in the cycle-cutting phase. This results in the following table, where $(lc0, lc3)$ is signal $ra_{\_0}$.

	$(lc0, lc3)$
$[ra, lc0, lc3, lc4]$	$\surd$
$[ra, lc0, lc3, lc4]$	$\surd$

The V1 algorithm, which eagerly selects cuts based on the number of paths cut, is employed. One or more `set_disable_timing` timing cut constraints are written out, and all rows representing covered by this edge are removed. The algorithm ends when there are either no more edges (some cut paths could not be cut) or there are no more rows (all cut paths have been cut) in the table. This results in the following two sdc constraints being applied to remove the two rows in the table.

```
set_disable_timing -from A0 -to Y [find -hier cell *lc3]
set_disable_timing -from B0 -to Y [find -hier cell *lc3]
```

These two cut points are highlighted in orange in Figure 11. The algorithm has now generated a complete DAG for the system with all of the true paths intact, all of the cycles cut, all of the false paths cut, and without leaving any orphaned gates because a true path passes through each gate in the design. Timing targets can be placed across each of the timing endpoints in Θ to optimize the sizing and placement of the gates to ensure that hazards to not occur in the design resulting in failure, and the performance of the circuit can be optimized.

6. EVALUATION APPROACH

This tool is evaluated on asynchronous pipelines using a bundled data or micropipeline approach [Sutherland 1989]. A micropipeline consists of a traditional Boolean logic data path and an asynchronous control path. The data path contains acyclic combinational logic (CL) and registers (L). The control path consists of linear control blocks (LC), which perform via handshaking the role of the clock. Handshake clocking generates the appropriate timing and sequencing for the design. It is elastic in nature and can stall if required. The minimum latency through the control logic plus a margin must be greater than the maximum delay of the combinational logic in order to fulfill the setup and hold time constraints at the register bank. Thus, delay elements may be required between LC blocks.

The C-element used in the Sutherland micropipeline is replaced with each controller in the full family of four-cycle handshake controllers with data valid on the rising request [Nagasai et al. 2010; Birtwistle and Stevens 2014]. To simplify the evaluation and to focus on the controllers, the data path of the micropipeline has been removed. Since there is no data path, this implements a token FIFO where the handshake controllers are allowed to operate at maximum frequency. Pipelines of depth four are employed, rather than the three deep example shown in Figure 2. The simulations employ controllable interfaces on the input and output of the pipeline that operate faster than the response time of the controller under test.

The evaluation uses the methodology in Stevens et al. [2009], which is based on relative timing [Stevens et al. 2003]. A set of relative timing constraints specific to each controller are generated that must hold for the circuit to perform hazard free. These are mapped onto each controller as a set of path-based constraints that are provided to the tool in the true path set Θ and the cut path set Φ. Three additional path constraints are added to reduce the cycle time of the controller: a constraint from lr of the current controller to lr of the downstream controller, lr of the downstream controller to ra of the current controller, and ra of the downstream controller to ra of the current controller. These endpoints are identified as red dots and black arrows in Figure 8.

Those endpoints are mapped onto the local controller constraints $\{(lr, rr)(lr, la)(ra, la)\}$ that are passed to the tool in Θ and Φ (adjusting timing endpoints to identify true paths in each of the specific controllers). The handshake channel cycles are cut by adding timing endpoint (ra, rr) to the cut set Φ.

The sdc constraints generated by the tool are employed in synthesis and simulation, where delays are attached to each of the three identified timing paths. The delays are customized to each specific controller design based on its response time such that negative slack does not occur.

The pipelines are synthesized with a commercial CAD tool employing optimization for power and performance using the Artisan 65nm library. This tool automatically cuts cycles in the design based on a minimum cut algorithm that does not respect timing paths provided to the design. The designs are then placed and routed with SoC Encounter to determine layout area and parasitics. The numbers for forward latency, backward latency, and cycle time are generated by simulating the post-routed design using sdf (standard delay format) back annotation. The designs are tested for 50 handshake cycles to generate a vcd (value change dump) file that reports node activity. The vcd file is used with PrimeTime to generate power and simulation time numbers on the post-APR (automatic place and routed) design using back-annotated parasitics extracted from the layout.

7. PRELIMINARY RESULTS

7.1. Benefits of Correct Cycle Cutting

Path-based timing-driven optimizations are not able to be performed if timing paths through the circuit have been cut. The LC circuit in Figure 3 is used as a preliminary example. Optimization results are significantly degraded when commercial EDA tools cut cycles without respecting timing paths because the tools must perform untimed circuit optimization. We show the power, area, and performance benefits of applying timing-driven optimizations on an asynchronous handshake controller when timing paths are ensured to remain uncut through the application of the algorithms in this article. However, performance, power, and area degradation is a secondary problem. When true timing paths are cut, the circuit cannot be optimized nor evaluated for that path. If this is a path that is required for correct operation, the circuit may fail.

The three timing constraints used for the micropipeline performance analysis described in Section 6 are employed. Verification identifies additional constraints that must hold in this design for correct operation. The relative timing constraint $lr\uparrow \mapsto y\uparrow \prec rr\downarrow$ states that y must rise before rr falls. Thus, we also include a correctness timing endpoint pair (lr, y) for the design. These four timing endpoints are added to both Θ and Φ. The external cycle cut constraint (ra, rr) is added to Φ. The tool is passed these constraints and the Verilog design. A set of sdc constraints are generated that create a DAG. All false paths and cycles are cut, and at least one timing path specified in Θ passes through every gate in the design. The design has 10 true paths identified as GCPs, cuts 8 cycles, and 14 false paths or external cycles. This is accomplished with 8 timing graph cuts.

The design is evaluated under the following four scenarios.

—**No TPCC Constraints:** The commercial CAD tool algorithm cuts all cycles in the design not respecting timing paths.
—**External Cut Constraints:** The external cut constraints are provided by our tool, but local cycles and false paths are left to the commercial tool to cut.
—**Local Cut Constraints:** The algorithms described here are used to cut local cycles and false paths, respecting the true timing paths provided in set Θ. The external cycles are cut by the commercial EDA algorithms, not respecting timing paths
—**All TPCC Constraints:** All cycles and false paths are cut using our timing path cycle-cutting algorithms that retain the true paths uncut.

Table I shows that there is a substantial improvement in circuit quality obtained by employing cuts from the timing path constrained algorithms in this article when

Table I. Comparison of Performance Metrics on Figure 3 Pipeline Using Timing Path Cycle Cutting (TPCC) Versus Commercial EDA Algorithm

	No TPCC Constraints	External Cut Constraints	Local Cut Constraints	All TPCC Constraints
Forward Latency (ps)	98	128	85	108
Backward Latency (ps)	328	348	305	233
Cycle Time (ps)	520	540	460	390
Area (um^2)	362	384	237	146
Power (mW)	2.30	2.31	1.72	1.01
Simulation Time (nS)	32.5	34.3	29.3	27.8
Energy/token (fJ)	374	395	253	141
No. of Cut Timing Paths	2	2	0	0

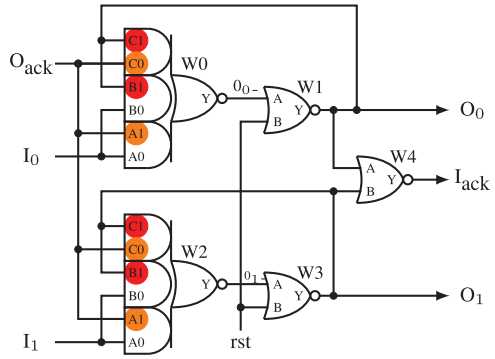

Fig. 12. WCHB Circuit. Red and orange circles denote locally and externally cut timing arcs.

compared to cuts automatically generated by the commercial CAD tool. Applying our algorithm for this circuit over letting the commercial EDA tool create the DAG results in improvements of 1.3× for cycle time, 2.5× for area, and 2.7× for energy per token. The table also points out the importance of including both the local and external cycle cut constraint paths. If only the external cycle cut path constraints are generated by our tool, the results are generally worse than having the commercial CAD tool perform all cycle cutting. Simply employing the local constraints helps, as this reduces energy by 1.5× over the commercial CAD tool. However, there still remains a penalty of 1.8× in energy per token over our algorithm if one allows the commercial CAD tool to perform external cycle cutting.

7.2. Generality of Approach

This tool can used with any design methodology and applied to any controller design. This is illustrated by applying this to a well-known quasi–delay-insensitive controller. Figure 12 shows the implementation of a weak-condition half-buffer (WCHB) [Lines 1998]. This design has been mapped to the same cell library and characterization flow described in Section 6. The constraint sets Θ and Φ for this circuit include timing endpoints $\{(I_0, O_0),\ (I_1, O_1),\ (I_0, I_{ack}),\ (I_1, I_{ack})\}$. Two cut path endpoints $\{(O_{ack}, O_0), (O_{ack}, O_1)\}$ are used to remove cycles in the handshake channel are added to Φ. The algorithm generates the cycle cuts shown graphically in Figure 12.

Post-layout results for a four-deep pipeline are generated. Table II shows that using both local and external cycle cut constraints clearly results in the best design. This timing-optimized implementation has nearly a 2× improvement in forward and backward latency and cycle time, a 1.4× area advantage, and a 3.2× energy-per-token advantage.

Table II. Comparison Using Timing Path Cycle Cutting (TPCC) Versus the Algorithm in a Commercial CAD Tool for Delay-Insensitive WCHB

	No TPCC Constraints	External Cut Constraints	Local Cut Constraints	All TPCC Constraints
Forward Latency (ps)	163	160	153	83
Backward Latency (ps)	273	270	253	145
Cycle Time (ps)	510	520	460	270
Area (um^2)	1270	1214	1233	891
Power (muW)	717	607	646	344
SimTime (nS)	53.8	54.3	52.5	34.5
Energy/token (fJ)	770	659	678	237
No. of Cut Timing Paths	2	2	2	0

Table III. The "Total Cycles Found / $\text{Cycles}_{\text{false paths}}$ Left Uncut/Orphans" for the V1 Algorithm Applied to the Full 4-Phase Family of Controllers

LoR	L000	L200	L400	L220	L420	L222	L422	L440	L442	L444
R0000	–	–	$18/3_2/0$	35/0/2	–	–	11/0/2	$24/3_0/7$	$7/1_0/0$	$5/1_0/0$
R0020	38/0/1	–	$17/2_2/1$	14/0/0	11/0/0	13/0/0	$9/1_1/0$	13/0/0	9/0/0	8/0/0
R0040	20/0/0	23/0/1	15/0/0	14/0/0	44/0/6	19/0/8	8/0/0	5/0/0	8/0/0	10/0/0
R0022	25/0/0	59/0/9	$7/1_1/0$	10/0/4	19/0/9	8/0/0	7/0/0	3/0/0	4/0/0	$4/1_1/0$
R0042	39/0/15	13/0/1	14/0/0	–	16/0/0	35/0/7	7/0/0	$10/1_0/0$	6/0/0	$5/1_0/0$
R2022	22/0/11	30/0/6	44/0/12	7/0/0	12/0/5	12/0/3	8/0/0	6/0/0	4/0/0	.
R2042	50/0/6	20/0/1	10/0/0	5/0/3	7/0/0	8/0/0	6/0/0	4/0/0	4/0/0	.
R0044	10/0/0	7/0/0	10/0/1	4/0/0	5/0/5	10/0/6	4/0/0	6/0/0	3/0/3	$3/1_0/0$
R2044	7/0/1	9/0/0	7/0/0	4/0/6	6/0/7	6/0/1	4/0/0	2/0/0	3/0/5	.
R4044	18/0/0	7/0/1	.	3/0/0	.	$5/1_0/0$	.	.	.	.
R2222	19/0/8	7/0/3	5/0/0	3/0/0	5/0/0	5/0/3	4/0/0	4/0/0	2/0/1	.
R2242	17/0/0	9/0/0	8/0/0	7/0/3	6/0/5	5/0/2	4/0/0	3/0/0	3/0/0	.
R2262	7/0/2	7/0/0	10/0/0	4/0/0	$5/2_1/0$	.	.	3/0/0	.	.
R2244	3/0/1	4/0/0	4/0/1	1/0/0	1/0/0	1/0/0	1/0/0	1/0/0	1/0/0	.
R2264	5/0/0	6/0/0	5/0/0	1/0/0	2/0/0	.	.	1/0/0	.	.
R4244	5/0/4	6/0/3	.	1/0/0	.	2/0/0	.	.	.	.
R4264	4/0/1	4/0/0	.	1/0/3	.	.	.	.	.	.

8. RESULTS

The algorithm described in this article is written in C++. The results are reported for runs on a Core i7 processor with 4GB memory. Sequential control circuits are relatively small, so this problem is not constrained by runtime or memory.

8.1. Complete Family of Four-Cycle Handshake Controllers

This example set consists of the complete family of 131 untimed four-cycle handshake controllers with data valid at the rising edge of request (lr) [Nagasai et al. 2010; Birtwistle and Stevens 2014]. The specifications are generated from concurrency reduction rules, and they are synthesized with Petrify. This creates a rich set of controller modules with various properties, such as half and full data buffered pipelines. The concurrency reduction rules were applied to the most concurrent protocol to generate the complete set of untimed (speed-independent and delay-insensitive) protocols.

The evaluation approach described in Section 6 is employed. This includes the three paths ($lr \rightarrow la$), ($lr \rightarrow rr$), and ($ra \rightarrow la$). Additional timing paths specific to each controller that are required to remove hazards from the design are also included. The external cycle cut constraint ($ra \rightarrow rr$) is included to remove cycles on the handshake channel.

Data is included in tabular and graphical form. Table III shows some results for each individual controller. Some of the handshake controllers failed synthesis from Petrify and are marked as "–" in the table. Those that deadlock due to too much concurrency reduction are marked with ".". Graphical results collect data into sets for each of the left cut (Lxxx). Thus, from 6 to 16 controllers are included for each Lxxx value in the graphs. The table and graphs generally display data from higher to lower concurrency. (The

Table IV. Comparison of Performance Metrics Using the Algorithm in Commercial CAD Tool Versus Timing Path-Based Cycle Cutting (V1) (Commercial CAD Tool Number/V1 Number)

	Minimum Value	Maximum Value	Average
Forward Latency	0.76×	3.56×	1.60×
Backward Latency	0.33×	2.92×	1.55×
Cycle Time	0.77×	2.32×	1.52×
Area	1.46×	4.49×	2.96×
Power	1.0×	6.19×	2.81×
SimTime	0.81×	2.16×	1.42×
Energy/token	1.43×	6.34×	3.84×
$e\tau$/token	1.16×	13.69×	5.45×

improved left cut nomenclature of Birtwistle and Stevens [2014] is employed.) Larger Lxxx numbers represent increased concurrency reduction on the output rr/ra channel. Likewise, Rxxxx cuts for each Lxxx set create orthogonal concurrency reduction on the input lr/la channel. All concurrency reduction values for the input channel are included in the Lxxx value, so the numbers can have significant variance. The average value for the left cut set is identified by the line with the maximum and minimum values identified with the error bars.

Table III shows the total cycles, cycles and false paths left uncut, and orphans for the design. The total number of cycles in these controllers is identified in the table. The amount of concurrency in a design is directly proportional to the number of state variables required [Birtwistle and Stevens 2008]. As expected, the most concurrent protocols contain the largest number of cycles due to more state-holding feedback signals. Table III also shows the number of cycles and false paths left uncut, and the number of orphaned gates. The uncut false paths are reported next to the uncut cycles because false paths are removed when cycles are cut. Six designs have one false path that was left uncut, and two designs have two uncut false paths. All cycles were removed in 118 of the 131 test cases employing the given constraint paths. A number of gates were orphaned. Gates are orphaned for two reasons: first, there is no timing path through the gate, and second, all input to output timing arcs get cut.

Further investigation reveals that all the orphan gates have no timing constraint path passing through them in both the V1 and V2 algorithms. These gates are primarily associated with reset and the local state variable logic. Thus, these gates can be sized by applying constraint paths that are specific to the state logic of each design.

Table IV summarizes the performance results. It presents a comparison of the performance values employing cycle cutting done by a commercial CAD tool and by the V1 algorithm for latency, cycle time, area, power, and energy. The average aggregate improvement of forward latency × area × $e\tau$/token results in a 25.8× improvement over a commercial CAD tool across the protocol set.

Figure 13 shows the energy delay product comparing cycle cutting being performed by the V1 algorithm and a commercial CAD tool. The benefit of the V1 algorithm ranges from an improvement of 13.7× to 1.2× over a commercial CAD tool. Figure 14 shows the forward latency, Figure 15 shows the backward latency, and Figure 16 shows the cycle time of each module with cycle cutting performed by a commercial CAD tool and our timing path-driven cycle cutting. A comparison of these graphs show that the commercial CAD tool typically generates a slower circuit except for a few cases where it used big gates that improved performance by expending substantially more energy.

Figure 17 shows post-layout area. Gates are substantially oversized when the commercial CAD tool performs cycle cutting. Four of the five cases with smallest area advantage (1.9× times or less) are for the circuits with uncut cycles. In these five cases, the commercial CAD tool creates additional cycle cuts after our timing-driven algorithm is employed, as it synthesizes and optimizes the design.

The same conclusion can be made by comparing the power consumption and completion time reported in Figures 18 and 19. There are five designs that have higher performance when the commercial CAD tool performs cycle cutting, but the performance

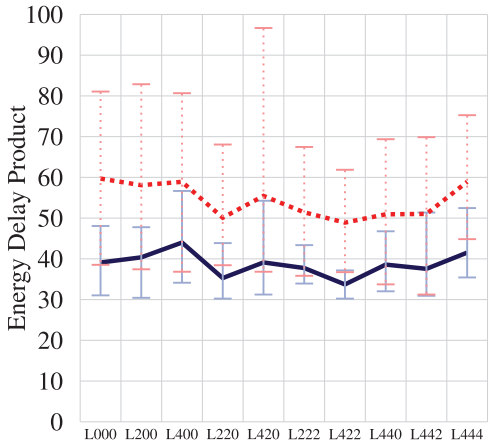

Fig. 13. $e\tau$ ratio averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

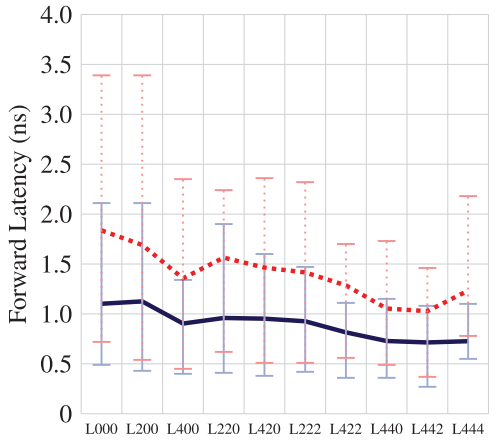

Fig. 14. Forward latency averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

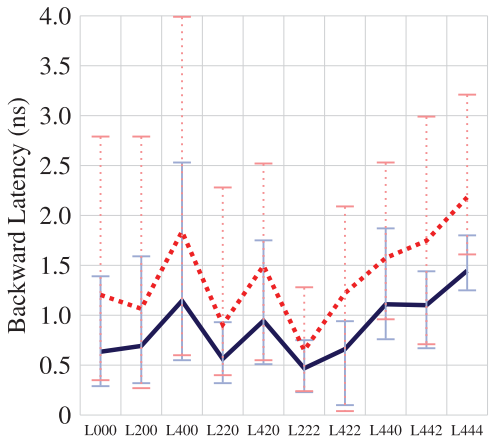

Fig. 15. Backward latency averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

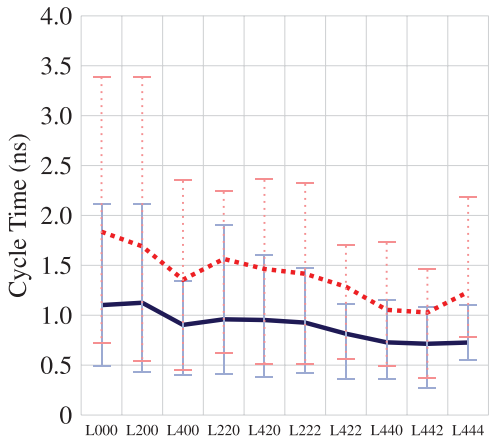

Fig. 16. Cycle time (post APR) averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

improvement again comes at the cost of expending lots more power. Energy per token is shown in Figure 20. The Energy numbers for the pipeline design average 3.84× larger for the case when the commercial CAD tool performs cycle cutting.

Results of the V1 and V2 algorithms were compared for forward latency, backward latency, and cycle time. The average variation was found to be 0.3%. Designs using the V2 algorithm were generally faster except for seven cases with more than ±10% variation. Similarly, in terms of power consumption and performance, the results were pretty much the same except in six cases where the variation was more than ±10% and ±5%, respectively. Similarly, the average energy variation is around 0%, but four of the the previous outlying designs show results with more than ±10% variation.

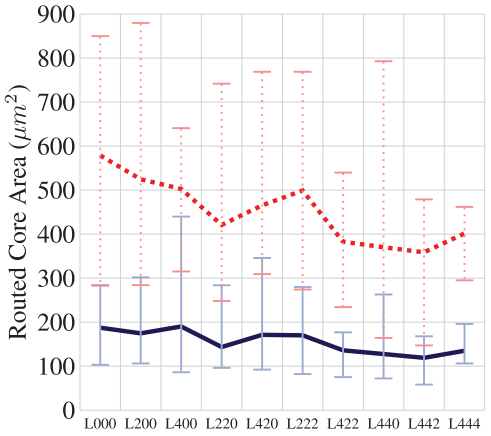

Fig. 17. Core area averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

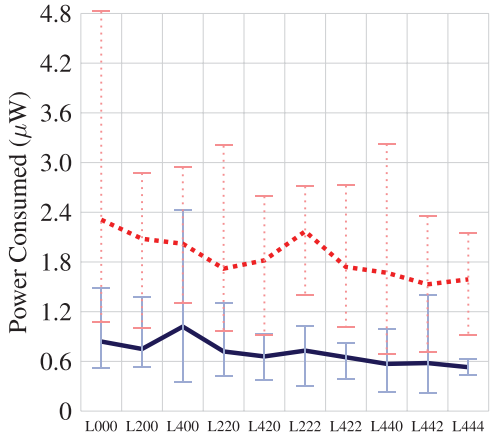

Fig. 18. Power consumption averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

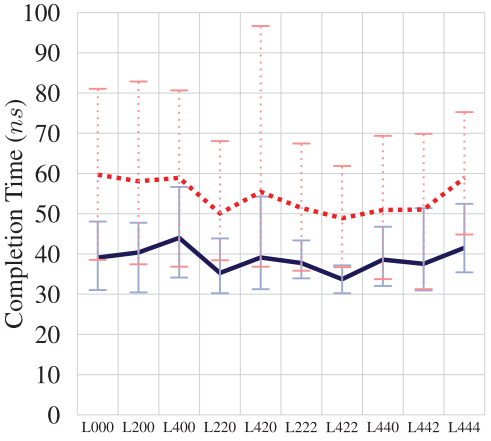

Fig. 19. Computation time averaged across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

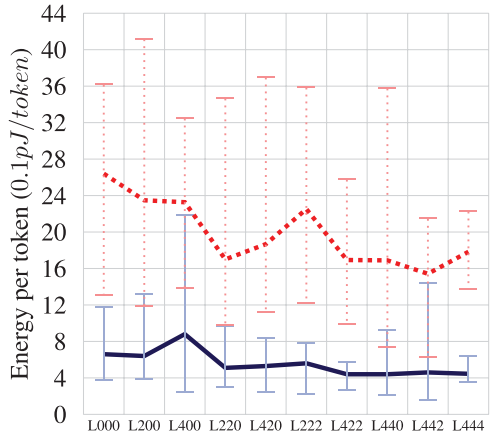

Fig. 20. Averaged energy consumed across left cuts. Interval shows largest and smallest values for the cut, line passes through the mean. Dotted line depicts using commercial CAD tool, while solid line our algorithm.

8.2. Benchmark Circuits

A number of benchmark circuits of varying complexity were also evaluated, including the largest published asynchronous finite-state machines that sequential synthesis tools are able to create. These benchmarks include a GCD design and modules from the Post Office [Stevens et al. 1986] and PSCSI [Yun and Dill 1999]. These designs were synthesized using Petrify to generate a gate-level netlist to which reset was added by hand.

The test setup for these designs is similar to that for the examples in Section 7. The notable exception is that the circuits were not formally verified for correctness against the specification in order to generate the true timing paths in the design. Instead,

manual analysis of these designs was performed to identify the true paths taken for each input to output path in the design. Only one true path from each primary input to each primary output was identified, if one exists. These true paths are supplied to the algorithm. Also, the external connectivity of these designs is ignored, so no external cut paths were employed to cut handshake channel cycles. Using this information, the cycle cuts were generated for these circuits using the V1 and V2 algorithms and compared against a commercial CAD tool.

TableVI shows the complexity of each of these designs and a comparison of generating the cycle cuts using the V1 and V2 algorithms. Design complexity is based on the number of paths, cycles, and gates. The two algorithms are compared based on the number of unsized gates and algorithm runtime. Analysis of the pscsi-isend design provides a good picture of the exponential nature of the exhaustive V2 algorithm. The algorithm considers edges that are not GCPs of specified true timing paths as cut candidates. There are over 6,000 candidates in the pscsi-isend design. The V2 algorithm performs an exhaustive search to find a zero-cost solution on all possible cut combinations to cut all false paths. This search is expensive and consumes significant runtime. For the pscsi-isend design, the exhaustive algorithm does not improve the results over the eager V1 algorithm.

Table V gives a comparison for the designs generated by the algorithms with respect to a commercial CAD tool in terms of timing paths that have been cut, area, energy per token, runtime performance, and energy delay product $e\tau$. These numbers give a comparison on the effectiveness of applying the greedy approach (V1) and the exhaustive approach (V2) to finding the cycle cut points in a circuit. The commercial tool preserves all timing paths in four out of seven designs. The average results for the designs generated using the V1 algorithm compared to a commercial CAD tool are less than half the size and energy, with a 5% performance improvement. This gives a $2.9\times$ $e\tau$ benefit. The benefits for the V2 algorithm are similar to the V1 algorithm, with the only difference being that the designs are very slightly smaller.

9. CONCLUSIONS

Timing paths must be cut to represent the timing graphs of sequential circuits as DAGs in the current state-of-the-art EDA tools. An algorithmic approach is presented for automating the timing path-driven generation of these cycle cuts so that the EDA tools can properly perform gate sizing for performance, area, and energy optimizations on sequential circuits without modifying the underlying structural netlist. Timing information is specified as an input to the algorithm. It is passed as timing endpoints of two forms: those that identify the true timing paths in the circuit and thus cannot be cut to preserve necessary timing paths, and those that identify false paths and external cycles that must be cut. A method based on the greatest common path is provided for specifying the correct timing paths in the sequential circuits based on timing endpoints composition. The CAD tool reports on the quality metrics of the results, consisting of the number of cycles left uncut, the number of false paths that were not cut, and the number of gates that do not have a timing path passing through them. Two versions of the algorithm are presented: a faster greedy search as well as an exhaustive algorithm that returns a result of the highest quality.

The CAD tool developed generates cycle-cutting constraints in the sdc format. This timing path-driven cycle-cutting algorithm is a key component of any design and CAD flow that enables asynchronous design to be synthesized, placed and routed, power and performance optimized, and validated for post-layout timing correctness using commercial EDA tools. The input is the structural Verilog design of sequential controllers that implement handshake protocols and sequencing in asynchronous designs.

The algorithms are general to any sequential circuit. The algorithm is demonstrated on a test bench of 131 four-cycle bundled data asynchronous controllers, one delay-insensitive design and a set of large gate count benchmark circuits. Circuits in this example set have as many as 325 cycles and over 6,000 paths in the implementation. The general runtime for each example is very small with the worst case for the large

Table V. Results Comparison for Benchmark Circuits

	Commercial CAD tool				V1 Algorithm			V2 Algorithm			V1 Benefits			V2 Benefits		
	Area (um^2)	Energy/ token (pJ)	Comp time (ns)	False Cuts	Area (um^2)	Energy/ token (pJ)	Comp time (ns)	Area (um^2)	Energy/ token (pJ)	Comp time (ns)	Area	Energy/ token	Comp time	Area	Energy/ token	Comp time
rcv-setup	68.6	0.05	86.78	0	42.0	0.03	91.15	42.0	0.03	91.15	1.63	1.67	0.94	1.63	1.67	0.94
sbuf-send-ctl	224.6	0.78	316.81	0	108.0	0.29	316.09	103.2	0.29	316.09	2.07	2.68	1.00	2.17	2.69	1.00
pscsi-trcv-bm	309.5	0.66	143.91	0	96.0	0.15	149.78	100.8	0.16	149.78	3.22	4.42	0.96	3.07	4.05	0.96
pscsi-tsend-bm	347.2	0.79	223.41	0	120.0	0.23	199.47	115.2	0.22	199.47	2.89	3.43	1.12	3.01	3.65	1.12
pscsi-tsend	309.5	0.61	219.42	2	127.2	0.22	188.61	122.4	0.21	188.61	2.43	2.77	1.16	2.53	2.92	1.16
pscsi-isend	495.5	0.98	296.47	7	204.0	0.44	258.79	204.0	0.44	258.79	2.42	2.25	1.15	2.42	2.25	1.15
gcd	642.0	1.25	314.46	4	348.6	0.61	299.97	348.6	0.61	299.97	1.95	2.04	1.05	1.84	2.04	1.05
Average Benefit											2.35	2.74	1.05	2.38	2.74	1.05

Table VI. Benchmark Circuits Design Comparison (Number of Gates with All the Input to Output Paths Cut)

	No. of Cycles	Gate Count	Unsized Gates		Algorithm Runtime (s)		No. of Paths
			V1	V2	V1	V2	
rcv-setup	1	8	2(0)	2(0)	<0.01	<0.01	4
sbuf-send-ctl	12	28	16(1)	16(0)	<0.01	0.01	120
pscsi-trcv-bm	6	26	11(1)	11(0)	<0.01	<0.01	32
pscsi-tsend-bm	10	33	7(2)	7(0)	0.02	0.03	377
pscsi-tsend	10	35	7(2)	7(0)	0.04	0.12	1,819
pscsi-isend	325	43	19(2)	19(0)	0.17	49,710	6,122
gcd	22	72	34(0)	34(0)	<0.01	0.01	175

benchmark design pscsi-isend being 0.17s for the greedy algorithm, and 13.8 hours for the exhaustive algorithm. Results are reported with an identical set of timing paths on these examples.

It is shown that allowing EDA tools to generate a DAG employing algorithms that do not respect timing paths passing through the sequential circuit leads to issues in timing optimization and validation, as well as producing inferior circuits. The circuits generated when cycle cutting is performed by a commercial CAD tool are, on average, 2.96× larger, operate 1.42× slower, have a forward latency 1.60× greater, and consume 3.84× more energy. The average aggregate benefit of this approach is a 25.8× improvement using these metrics. However, even more importantly, true timing paths of the circuit cannot be evaluated using static timing analysis unless a DAG is generated that respects the timing paths. If some timing paths do not meet specified delays, the circuit will fail to operate correctly.

REFERENCES

Peter A. Beerel, Georgios D. Dimou, and Andrew M. Lines. 2011. Proteus: An ASIC flow for GHz asynchronous designs. *IEEE Design and Test of Computers* 28, 5 (2011), 36–51.

Graham Birtwistle and Kenneth S. Stevens. 2008. The family of 4-phase latch protocols. In *Proceedings of the 14th International Symposium on Asynchronous Circuits and Systems*. IEEE, 71–82.

Graham M. Birtwistle and Kenneth S. Stevens. 2014. Modelling mixed 4 phase pipelines: Structures and patterns. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*. IEEE, 27–36.

Tam-Anh Chu. 1987. *Synthesis of Self-Timed VLSI Circuits From Graph-Theoretic Specifications*. Ph.D. Dissertation. Massachusetts Institute of Technology.

William S. Coates, Alan L. Davis, and Kenneth S. Stevens. 1993. Automatic synthesis of fast compact self-timed control circuits. In *Proceedings of the IFIP Working Conference on Design Methodologies*. 193–208.

J. Cortadella, M. Kishinevsky, S. M. Burns, A. Kondratyev, L. Lavagno, K. S. Stevens, A. Taubin, and A. Yakovlev. 2002. Lazy transition systems and asynchronous circuit synthesis with relative timing assumptions. *IEEE Transactions on Computer-Aided Design* 21, 2 (Feb 2002), 109–130.

S. A. Edwards. 2003. Making cyclic circuits acyclic. In *Proceedings of the 40th Conference on Design Automation*. ACM, New York, NY, 159–162.

V. V. Filippovich. 1973. Transforming a cyclic directed graph into an acyclic graph. *Cybernetics and Systems Analysis* 9, 2 (March 1973), 348–351.

Alex Kondratyev and Kelvin Lwin. 2002. Design of asynchronous circuits using synchronous cad tools. *IEEE Design & Test of Computers* 19, 4 (July-Aug. 2002), 107–117.

Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. 2000. Asynchronous design using commercial HDL synthesis tools. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 114–125.

Andrew M. Lines. 1998. *Pipelined Asynchronous Circuits*. Master's thesis. California Institute of Technology, Pasadena, CA.

Sharad Malik. 1994. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 7 (1994), 950–956.

Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall International, London.

Santosh Nagasai, Kenneth S. Stevens, and Graham Birtwistle. 2010. Concurrency reduction of untimed latch protocols – theory and practice. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*. IEEE, 26–37.

Osama Neiroukh, Stephen A. Edwards, and Xiaoyu Song. 2008. Transforming cyclic circuits into acyclic equivalents. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 10 (2008), 1775–1787.

Marc D. Riedel and Jehoshua Bruck. 2003. The synthesis of cyclic combinational circuits. In *Proceedings of hte Design Automation Conference*. ACM/IEEE, 163–168.

T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovnni-Vincentelli. 1996. Analysis of combinational cycles in sequential circuits. In *Proceedings of the 1996 IEEE International Symposium on Circuits and Systems,* Vol. 4.

A. B. Smirnov. 2009. *Asynchronous Micropipeline Synthesis System*. Ph.D. Dissertation. Boston University.

Kenneth S. Stevens, Ran Ginosar, and Shai Rotem. 2003. Relative timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 1, 11 (Feb. 2003), 129–140.

Kenneth S. Stevens, Shane V. Robison, and Alan L. Davis. 1986. The post office—communication support for distributed ensemble architectures. In *Proceedings of 6th International Conference on Distributed Computing Systems*. 160–166.

Kenneth S. Stevens, Yang Xu, and Vikas Vij. 2009. Characterization of asynchronous templates for integration into clocked CAD flows. In *Proceedings of the 15th International Symposium on Asynchronous Circuits and Systems*. IEEE, 151–161.

Ivan E. Sutherland. 1989. Micropipelines. *Communications of the ACM* 32, 6 (June 1989), 720–738. Turing Award Paper.

A. Taubin, J. Cortadella, L. Lavagno, A. Kondratyev, and A. Peeters. 2007. Design automation of real-life asynchronous devices and systems. *Foundations and Trends® in Electronic Design Automation* 2, 1 (2007), 1–133.

Yang Xu and Kenneth S. Stevens. 2009. Automatic synthesis of computation interference constraints for relative timing. In *Proceedings of the 26th International Conference on Computer Design*. IEEE, 16–22.

Kenneth Y. Yun and David L. Dill. 1999. Automatic synthesis of extended burst-mode circuits: Part II (automatic synthesis). *IEEE Transactions on Computer-Aided Design* 18, 2 (Feb. 1999), 118–132.

Received September 2015; revised January 2016; accepted February 2016