A Timing Graph Based Approach to Mode Merging

Subramanyam Sripada
Synopsys Inc.
ssubram@synopsys.com

Murthy Palla
Synopsys India Pvt. Ltd.
murthyp@synopsys.com

ABSTRACT

With shrinking technologies and increasing design complexities, it is common to have a large number of modes (functional, scan, test and so on) and corners (PVT device and interconnect). This leads to an explosion in the number of scenarios $(\#modes \times \#corners)$ that need to be validated for timing. While multiple tactics are required to handle this problem, one essential way to address this is by reducing the number of modes by merging individual modes into superset modes. However, with the overriding necessity to maintain sign-off accuracy, mode merging with high merge-factor is very complex. In this paper, we propose a novel automated timing graph based approach to mode merging that is designed to meet these requirements. By construction, there is an inbuilt validation that the merged constraints correctly model the intent of original constraints. This technology is tested on large industrial designs and the results are provided.

1. INTRODUCTION

Modern designs often accommodate multiple functional blocks with the ability to selectively configure their behavior. For example, a chip may be designed to have a low power mode that might disable certain functionality to conserve power. In addition to these functional modes, it is a common practice to have several test modes and scan modes. All these modes are typically represented as different timing modes captured by their individual sets of timing constraints. Having separate constraints for each of these modes comes with several advantages like parallel development, opportunity to reuse and fine grained control over the sign-off process. However, having a large number of modes would mean either more time or hardware to perform timing analysis. With an exploded number of timing scenarios, neither of these is acceptable.

The performance of static timing analysis can be improved by using approaches like multi-corner multi-mode analysis [1], analyzing only dominant corners [2] and performing hierarchical timing analysis [3]. While these methods have their own merits and de-merits, they can all benefit from a reduced number of modes achieved by mode merging.

DAC'15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06 ...$15.00
http://dx.doi.org/10.1145/2744769.2744787.

A lot of design teams address the mode merging problem by manually merging the modes to create superset modes. However, this process is very tedious and error-prone. In addition, it is extremely difficult to debug wrong results due to erroneous constraints. An elegant solution to this problem is an automated process that ensures that the timing constraints of the merged modes are equivalent to the timing constraints of the individual modes. This paper proposes a timing graph based approach to automatically reduce modes into superset modes while assuring sign-off accuracy with the merged modes.

Despite being a problem of critical importance, mode merging is not well discussed in literature. To the best of our knowledge, this is the first paper divulging a detailed approach to mode merging. An approach to reduce the timing scenarios in implementation/ECO and sign-off phases is discussed in [4]. However, this approach is not comprehensive and cannot handle all the complex constraints and circuitry possible in today's designs.

The rest of this paper is organized as follows. section 2 provides the background required to introduce our solution to this problem. Our methodology to automatic merging of modes based on timing graph is proposed in section 3. The results of applying our methodology on multi-million gate industrial designs are provided in section 4. Finally, section 5 provides concluding remarks.

2. BACKGROUND

In this section, we will set the background required for the proposal of our mode merging approach by defining a timing relation and giving examples on timing relationship propagation and comparison. For definitions of basic terminologies such as timing graph, node, timing arc, startpoint, endpoint, tag etc., and a detailed explanation on tag propagation, please refer to [5].

Timing Relationship for a set of paths is defined by its launch clock, capture clock, timing endpoint/startpoint, rise/fall type, min/max path type and the constraint state (disabled, false path, multicycle path etc.) of the paths. Any arbitrary timing constraint in SDC [6] can be configured to have a constraint state. This is possible because the effect of any timing constraints can be captured at one or more endpoints in the form of some "state" which indicates the impact on the endpoint such as disabled, false path, multicycle path, constraint case, etc.

Consider the circuit in Figure 1. A minimal set of timing constraints are shown in Constraint Set 1. The first constraint *create_clock*, specifies that clock $clkA$ which has a period of 10 units defined on port $clk1$ can clock all the six registers in the example. The second constraint specifies that all timing paths going through the output pin Z of the inverter $inv1$ should require two clock cycles to propagate the data. There are two paths going through $inv1/Z$: (i) $rA/Q \rightarrow inv1/Z \rightarrow rX/D$ and (ii) $rA/Q \rightarrow inv1/Z \rightarrow and1/Z \rightarrow inv2/Z \rightarrow rY/D$. D, Q and CP are the data input pin, data output pin and clock input pin of the registers, respectively. The third constraint specifies that all timing paths going through the output pin Z of the AND gate $and1$ need to be considered false. There are two paths that go through $and1/Z$: (ii) $rA/Q \rightarrow inv1/Z \rightarrow and1/Z \rightarrow inv2/Z \rightarrow rY/D$ and (iii) $rB/Q \rightarrow and1/Z \rightarrow inv2/Z \rightarrow rY/D$. Note that path (ii) has both the false-path and the multicycle-path constraints applied to it. Tools generally define a set of precedence rules for such overlaps. In this scenario, false-path overrides the multicycle-path.

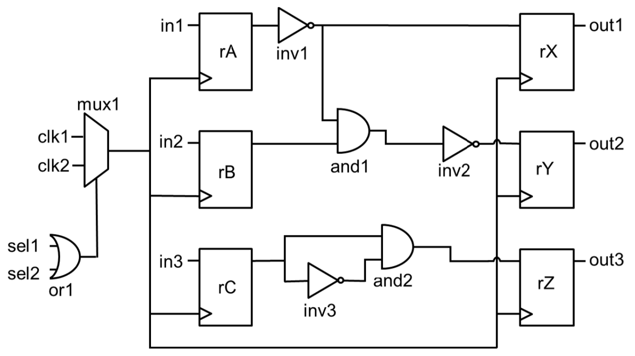

Figure 1: Example circuit

```
create_clock -name clkA -period 10 [get_ports clk1]
set_multicycle_path 2 -through [get_pins inv1/Z]
set_false_path -through [and1/Z]
```

Constraint Set 1: Constraints for demonstration timing relationship propagation

Start point	**End point**	**Launch clock**	**Capture clock**	**State**
*	rX/D	$clkA$	$clkA$	$MCP(2)$
*	rY/D	$clkA$	$clkA$	FP
*	rZ/D	$clkA$	$clkA$	-

Table 1: Timing relationships

If we bundle all paths reaching every endpoint, we get the *timing relationships* in Table 1. The first, second and third rows show the *timing relationships* for all the paths that end at rX/D, rY/D and rZ/D, respectively. This example also shows how constraints with different precedence can be modeled with this approach. Even though the multicycle-path constraint (MCP) affects some of the paths that reach rY/D, since the false-path constraint (FP) overrides the MCP, it doesn't figure in the timing relationships table for the endpoint rY/D. Constraint Set 1 does not have any constraints that affect rZ/D. So, no timing relation exists at this endpoint.

The timing relationships representation removes any dependency on how a timing constraint is specified and only models how it finally gets applied and affects the timing paths in the design.

Two sets of constraints are equivalent if and only if:

- Every timing relationship of the design obtained by applying the first constraint set is present by applying the second constraint set AND
- Every timing relationship of the design obtained by applying the second constraint set is present by applying the first constraint set

This is a powerful definition as it compares the effect of timing constraints on the design without comparing the constraints themselves. For example, if a multicycle path constraint described in previous section is rewritten so that it is specified on timing startpoints instead of timing endpoints, the effect of the constraints on the design might be the same although a simple comparison of constraints cannot determine the two constraint sets to be equivalent.

3. PROPOSED METHODOLOGY

The proposed methodology to mode merging is split into two steps. The first step, which we call *preliminary mode merging*, is designed to generate super set of Timing Relationships in the merged mode. With preliminary mode merging, we ensure that if a path is timed in any individual mode, the path is timed in its corresponding merged mode. The second step called *mode refinement* refines the Timing Relationships in the merged mode to ensure that the merged mode does not time any path unless it is timed by at least one individual mode. Because of this, there is an in-built, correct by construction validation step that ensures merged mode accurately represents the timing of individual modes.

To identify mergeable modes, we perform a mock run of preliminary mode merging through which we figure out modes that cannot be merged. For example, if two modes have clocks, which if merged will result in blocking of one or other clock, we mark them as non-mergeable. Similarly, if the constraints of two modes have incompatible values, these modes are also marked non-mergeable. To identify the sets of individual modes that can be merged into super-set modes, we create a mergeability graph as shown in Figure 2. The vertices of the mergeability graph represent the modes of the problem and an edge is added between two modes if they are mergeable. The maximal sets of mergeable individual modes are identified by finding cliques of this graph. This is done by using greedy algorithm as the number of modes is small. In Figure 2, the mergeable modes are highlighted by cliques $M1$, $M2$ and $M3$.

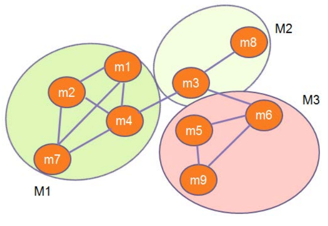

Figure 2: Mergeability graph

Due to space restrictions, we are unable to provide full details of this mergeability determination process. We only discuss the problem of reducing N mergeable modes into 1 superset mode.

We use the example circuit in Figure 1 to explain the proposed methodology. Wherever required in the rest of this paper, we define a new set of constraints for this circuit to demonstrate the concepts.

3.1 Preliminary mode merging

Through preliminary mode merging, we create preliminary versions of the superset or merged modes in the below steps.

3.1.1 Creating union of clocks

To create the union of clocks, we iterate through all the clocks of each of the individual modes and add each non-duplicate clock to the merged mode. A clock is treated as duplicate if there already exists a clock in the merged mode that has the same source(s) and waveform. During this process, we create a two way map between the individual mode clocks and the merged mode clocks. These maps are used to identify the matching between other clock based individual and merged mode constraints.

Consider the constrains of Constraint Set 2. Mode A has two clocks and mode B has three clocks. A comparison of the clock waveforms and sources indicates that clock *clkB* of mode A and clock *clkC* of mode B are identical, and all the other clocks are unique. Hence, the superset mode $A + B$ is created with four clocks that are the union of the clocks from modes A and B. If the modes have conflicting clock names, we make the merged mode clock names unique by adding unique suffixes. For example, clock *clkB* of mode B is renamed as clock *clkB_1* in the superset mode $A + B$.

```
Mode A:
 CLK1: create_clock -name clkA -period 10 -waveform {0 3} [get_port clk1]
 CLK2: create_clock -name clkB -period 20 [get_port clk2]
 CSTR1: set_clock_latency 3 [get_clock clkA]
 CSTR2: set_clock_latency -rise -min 1.1 [get_clock clkB]
 CSTR3: set_clock_latency -rise -max 1.3 [get_clock clkB]
Mode B:
 CLK1: create_clock -name clkA -period 10
 CLK2: create_clock -name clkB -period 16 -waveform {0 8} [get_port clk1]
 CLK3: create_clock -name clkC -period 20 -waveform {0 10} [get_port clk2]
 CSTR1: set_clock_latency 1.5 [get_clock clkA]
 CSTR2: set_clock_latency 2 [get_clock clkB]
 CSTR3: set_clock_latency -rise -min 1.11 [get_clock clkC]
 CSTR4: set_clock_latency -rise -max 1.31 [get_clock clkC]
Mode A + B:
 CLK1: create_clock -name clkA -period 10 -waveform {0 3} [get_port clk1]
 CLK2: create_clock -name clkB -period 20 -waveform {0 10} [get_port clk2]
 CLK3: create_clock -name clkA_1 -period 10
 CLK4: create_clock -name clkB_1 -period 16 [get_port clk1] -add
 CSTR1: set_clock_latency 3 [get_clock clkA]
 CSTR2: set_clock_latency -rise -min 1.1 [get_clock clkB]
 CSTR3: set_clock_latency -rise -max 1.31 [get_clock clkB]
 CSTR4: set_clock_latency 1.5 [get_clock clkA_1]
 CSTR5: set_clock_latency 2 [get_clock clkB_1]
```

Constraint Set 2: Constraints for demonstration of preliminary mode merging of clocks and clock based constraints

3.1.2 Merging clock based constraints

Once the union of all clocks is created, the next step is to add clock based constraints like set_clock_transition, set_clock_latency, set_clock_uncertainty, set_propagated_clock etc., to the merged mode. For this purpose, we check whether the clock constraints across different modes are common and are within a certain tolerance limit. In case the constraint values are within the tolerance limit, but not identical, we pick the minimum of *min* values and maximum of *max* values to be added to the merged mode.

Consider the constraints of the Constraint Set 2. The clock latency constraints CSTR2 of mode A and CSTR3 of mode B correspond the clock B of the merged mode $A + B$ and have a slight deviation in their values. As these are *min* constraints, we add constraint CSTR2 to mode $A + B$ with a value that is the minimum of values from modes A and B.

3.1.3 Create union of external delay constraints

To merge the set_input_delay and set_output_delay constraints, we iterate through these constraints of each individual mode and add all the unique constraints to the merged mode.

3.1.4 Creating intersection of case_analysis constraints

To merge the case_analysis constraints, we iterate through these constraints of each individual mode and check if they are present in all the individual modes with non-conflicting values. If so, we add these constraints directly to the merged mode. If not, we check if it is possible to translate the case_analysis statement to a false path constraint and then add it. If this is not possible, we drop the particular case statements from the merged mode. This might temporarily result in additional valid paths in the preliminary merged mode. However, in the *merged mode refinement* process described in section 3.2, we add additional false paths to the merged mode to precisely disable any unwanted paths from the preliminary merged mode.

3.1.5 Creating intersection of disable_timing constraints

To merge the disable_timing constraints, we iterate through these constraints of each individual mode to ensure that they are present in all the individual modes, and then add them to the merged mode.

3.1.6 Merging drive and load constraints

Similar to other constraints, to merge the drive and load constraints (such as set_input_transition, set_driving_resistance, set_load etc.) we iterate through the individual modes to ensure that these constraints are same across all the individual modes with all the values within the tolerance limit, and then add them to the merged mode.

3.1.7 Determining clock exclusivity

With the creation of a union of clocks from all the individual modes in the merged mode, it is not possible to directly take the clock exclusivity constraints from the individual modes. To identify clock exclusivity rules, we first iterate through all the individual modes and collect pairs of clocks that can co-exist within at least one individual mode. We then iterate through all the merged mode clocks and add clock exclusivity constraints between all the pairs of clocks that cannot co-exist in at least one individual mode.

3.1.8 Clock Refinement

Clock refinement is done to ensure that the merged mode does not have any extra clocks propagated through the clock network than the individual modes. For this purpose, we perform a breadth first traversal through each node/arc in the clock network and compare the clocks present in the merged mode to the clocks on the individual modes that are propagating through this node/arc. If any clock is found on a given node/arc in the merged mode, but not on any of the individual modes, we add a constraint to the merged mode that stops the propagation of the clock from that point onwards.

Consider the constraints from Constraint Set 3 written for the example circuit of Figure 1. It can be observed that the case statement for the inputs $sel1$ and $sel2$ have conflicting values between the two individual modes. So, we drop these case statements from the merged mode while performing the preliminary mode merging. During the clock network refinement, we observe that these inputs never change in any of the individual modes and add disable timing constraints CSTR1 and CSTR2 of the merged mode $A + B$. Further, due to the case statements, the 'select' input of $mux1$ is always fixed at 1. So, the merged mode clock $clkA$ does not propagate through $mux1$. To take this into account, we add constraint CSTR3 to the merged mode $A + B$.

```
Mode A:
 CLK1: create_clock -period 10 -name clkA [get_port clk1]
 CLK2: create_clock -period 20 -name clkB [get_port clk2]
 CSTR1: set_case_analysis 0 sel1
 CSTR2: set_case_analysis 1 sel2
Mode B:
 CLK1: create_clock -period 10 -name clkA [get_port clk1]
 CLK2: create_clock -period 20 -name clkB [get_port clk2]
 CSTR1: set_case_analysis 1 sel1
 CSTR2: set_case_analysis 0 sel2
Mode A + B
 CLK1:  create_clock -name clkA -period 10 -add [get_ports clk1]
 CLK2:  create_clock -name clkB -period 20 -add [get_ports clk2]
 CSTR1:  set_disable_timing [get_ports sel1]
 CSTR2:  set_disable_timing [get_ports sel2]
 CSTR3:  set_clock_sense -stop_propagation -clock [get_clocks clkA] [get_pins mux1/Z]
```

Constraint Set 3: Constraints for demonstration of clock refinement and inference of disable_timing constraints

3.1.9 *Creating intersection of exceptions*

To merge exceptions like set_false_path, set_multicycle_path, set_min_delay and set_max_delay, we iterate through the exceptions of all the individual modes and if the exceptions are found in all the modes, then we add them directly to the merged mode. If any exception is found only in a few modes, then we try to uniquify the exception by making it unique to just the modes in which it is present by the way of restricting the exception to a few clocks as explained in section 3.1.10.

The false path exceptions that cannot be uniquified are dropped from the merged mode. This might temporarily result in additional valid paths in the preliminary merged mode. However, in the *merged mode refinement* process described in section 3.2, we add additional false paths to the merged mode to precisely disable any unwanted paths from the preliminary merged mode.

3.1.10 *Exception uniquification*

Most of the times, it is possible to write the same exception in different forms. In cases where exceptions are present only in some, but not all the modes being merged, we may not be able to add the exceptions "as is" to the merged mode because the exceptions can potentially disable any paths that are otherwise valid in some individual modes. To overcome this, we use a technique called *exception uniquification* where we add additional information like $from$ or *to* clocks to the exception that makes it unique to the particular mode in which it is present.

We demonstrate exception uniquification with an example using the circuit in Figure 1. Consider the constraints of the Constraint Set 4. The multicycle path constraint MCP1 of mode A does not have an equivalent constraint in mode B. If it is added "as is" to the merged mode, it will set multicycle path also on the paths from rA/CP that are clocked by clock $clkB$ of the merged mode, which does not even exist in mode A.

```
Mode A:
 CLK1: create_clock -name clkA
 CSTR1: set_case_analysis 0 [mux1/S]
 MCP1: set_multicycle_path 2 -from [rA/CP]
Mode B:
 CLK1: create_clock -name clkB
 CSTR1: set_case_analysis 1 [mux1/S]
Mode A':
 CLK1: create_clock -name clkA
 CSTR1: set_case_analysis 0 [mux1/S]
 MCP1: set_multicycle_path 2 -from [get_clocks clkA] -through [rA/CP]
Mode A' + B:
 CLK1: create_clock -name clkA
 CSTR1: create_clock -name clkB
 MCP1: set_multicycle_path 2 -from [get_clocks clkA] -through [rA/CP]
```

Constraint Set 4: Constraints for demonstration of exception uniquification

By modifying MCP1 of mode A to MCP1 of mode A', which does not change the behavior of the constraint, we make mode A' mergeable with mode B as the modified MCP1 of A' does not affect any valid paths of mode B. So, we can create the merged mode $A' + B$ as listed in the Constraint Set 4.

3.2 Refinement of preliminary merged mode

The preliminary merged mode can potentially have extra paths that are not valid in any of the individual modes that can skew the results when directly used. To make the merged mode exactly identical in behavior to that of the individual modes, we employ the timing graph based equivalency checking to perform data refinement explained in this section.

Data refinement is done in two steps. In the first step, we traverse through the data network of the circuit to ensure that the clocks at any node/arc in the preliminary merged mode are present on that node/arc in at least one individual mode. If any extra clock is found, we add a constraint to the merged mode to stop the propagation of that clock from that particular node/arc onwards.

Consider the constraints from Constraint Set 5 written for the example circuit of Figure 1. The merged mode clock $clkB$ exists only in mode B where the pin rB/Q is set to constant 0. Because of this, the merged mode clock $clkB$ does not propagate beyond the pins rB/Q and $and1/Z$. To this effect, we add constraint CSTR6 to the merged mode $A + B$.

As the second step of data refinement we perform 3-pass comparison of timing relationships at startpoints and/or endpoints of the circuit to identify any extra timing paths and disable them by adding appropriate constraints to the merged mode. The 3-pass algorithm described in this section forms the basis for the relation comparison used to refine the data paths of the preliminary merged mode. It is used to check the equivalency of two different sets of constraints by the way of propagating their constraints through the timing graph and comparing the timing relations at the endpoints and/or startpoints.

To compare individual mode constraints and the preliminary merged mode constraints comprehensively, we need to

```
Mode A:
 CLK1: create_clock -name ClkA -period 2 [get_port clk1]
 CSTR1: set_input_delay 2.0 -clock ClkA [get_port in1]
 CSTR2: set_output_delay 2.0 -clock ClkA [get_port out1]
Mode B:
 CLK1: create_clock -name ClkB -period 1 [get_port clk1]
 CSTR1: set_input_delay 2.0 -clock ClkB [get_port in1]
 CSTR2: set_output_delay 2.0 -clock ClkB [get_ports out1]
 CSTR3: set_case_analysis 0 rB/Q
Mode A + B
 CLK1: create_clock -name ClkA -period 2 -add [get_ports clk1]
 CLK2: create_clock -name ClkB -period 1 -waveform -add [get_ports clk1]
 CSTR1: set_input_delay 2 -clock [get_clocks ClkA] [get_ports in1]
 CSTR2: set_input_delay 2 -clock [get_clocks ClkB] -add_delay [get_ports in1]
 CSTR3: set_output_delay 2 -clock [get_clocks ClkA] [get_ports out1]
 CSTR4: set_output_delay 2 -clock [get_clocks ClkB] -add_delay [get_ports out1]
 CSTR5: set_clock_groups -physically_exclusive -name ClkA_1 -group [get_clocks
                                              ClkA] -group [get_clocks ClkB]
 CSTR6: set_false_path -from [get_clocks ClkB] -through [get_pins rB/Q and1/Z]
```

Constraint Set 5: Constraints for demonstration of data refinement by stopping clock propagation

essentially determine *timing relationship* of every path in the design with both the sets of constraints and compare them. Doing this by brute force on each path can be very expensive. The 3-pass algorithm addresses this problem by performing comparison on sets of timing paths and refining the path selection only if necessary.

We use the constraint from Constraint Set 6 written for the circuit in Figure 1 to demonstrate the 3-pass algorithm. It can be observed that no false path constraints of modes A and B are in common. So, it is not possible to add any false paths to the merged mode at the time of creating the preliminary merged mode.

```
Mode A:
 CLK1: create_clock -p 10 -name clkA [get_port clk1]
 CSTR1: set_false_path -to rX/D
 CSTR2: set_false_path -to rY/D
 CSTR3: set_false_path -through inv3/Z
Mode B:
 CLK1: create_clock -p 10 -name clkA [get_port clk1]
 CSTR1: set_false_path -from rA/CP
 CSTR2: set_false_path -to rZ/D
Preliminary Merged Mode
 CLK1: create_clock -name clkA -period 10 -add [get_ports clk1]
Mode A + B
 CLK1: create_clock -name clkA -period 10 -add [get_ports clk1]
 CSTR1: set_false_path -to [get_pins rX/D]
 CSTR2: set_false_path -from [get_pins rA/CP] -to [get_pins rY/D]
 CSTR3: set_false_path -from [get_pins rC/CP] -through [get_pin inv3/A] -to
                                                          [get_pins rZ/D]
```

Constraint Set 6: Constraints for demonstration of data refinement using the 3-pass algorithm

The path selection and refinement are performed in the following three passes.

Pass 1

In this pass, we determine the *timing relationships* at all timing endpoints of the design. These *timing relationships* model the timing constraints affecting all the paths that end at a particular endpoint. This is done on one side by propagating the timing relationships of all the individual modes and on the other side by propagating the timing relationships of their corresponding merged mode from all startpoints of the circuit to all the endpoints.

Table 2 shows the timing relationships for the Constraint Set 6 being compared in pass 1. The first column in Table 2 indicates that all the paths ending at rX/D and having $clkA$ as both launch and capture clock are false in both the

Start point	End point	Launch clock	Capture clock	Individual mode state	Merged mode state	Pass1 result
*	rX/D	$clkA$	$clkA$	FP	V	X
*	rY/D	$clkA$	$clkA$	FP, V	FP, V	A
*	rZ/D	$clkA$	$clkA$	FP, V	FP, V	A

Table 2: Timing relationship comparison table for pass 1 [FP: False Path, V: Valid, M: Match, X: Mismatch, A: Ambiguous]

individual modes, but valid in the merged mode indicating a timing relationship mismatch. To address the mismatch, we add the constraint CSTR1 to the merged mode $A + B$ in Constraint Set 6. For the second and third columns representing all the paths ending at rY/D and rZ/D, respectively, we can observe that there is an ambiguity due to the presence of multiple timing relationships at these endpoints.

Ambiguity in pass 1 indicates that the paths ending with a specific endpoint are not guided by the same set of constraints and hence it is required to identify which startpoints these timing relations are propagating from to gain more clarity. To perform this, all the endpoints with ambiguity in timing relationship comparison are forwarded to pass 2 for a mode detailed analysis.

Pass 2

In pass 2, we compare *timing relationships* to determine matches/mismatches for all paths between a start point and an endpoint. The *timing relationships* in pass 2 model the timing constraints affecting all the paths that start at a particular startpoint and end at a specific endpoint. Pass 2 analysis is done only on selective endpoints that are found to have an ambiguous timing relationship comparison in pass 1. In pass 2, the timing relations of all the selected endpoints are back propagated towards the startpoints and the timing relations that reach the startpoints are compared.

Start point	End point	Launch clock	Capture clock	Individual mode state	Merged mode state	Pass2 result
rA/CP	rY/D	$clkA$	$clkA$	FP	V	X
rB/CP	rY/D	$clkA$	$clkA$	V	V	M
rC/CP	rZ/D	$clkA$	$clkA$	FP, V	FP, V	A

Table 3: Timing relationship comparison table for pass 2 [FP: False Path, V: Valid, M: Match, X: Mismatch, A: Ambiguous]

Table 3 shows the timing relationships for the sample constraints being compared in pass 2. The first column indicates that all the paths starting at rA/CP and ending at rY/D and having $clkA$ as both launch and capture clock have mismatch in timing relationship. We add constraint CSTR2 of mode $A+B$ in Constraint Set 6 to fix the mismatch. The second column indicates that all the paths starting at rB/CP and ending at rY/D and having $clkA$ as both launch and capture clock are valid in both the individual and merged modes. The third column of the table indicates that the paths starting at rC/CP and ending at rZ/D and having $clkA$ as both launch and capture clock have an ambiguous timing relationship comparison and hence it cannot be conclusively determined whether there is a match or mismatch at this stage.

Ambiguity in pass 2 indicates that the paths starting at a specific startpoint and ending at a specific endpoint are not guided by the same set of constraints and hence it is required to identify which reconvergence points these timing relations are propagating through to gain more clarity. To perform this, all the startpoint-endpoint pairs with ambiguity in timing relationship comparison are forwarded to pass 3 for a mode detailed analysis.

Pass 3

Finally, in pass 3, we identify all the re-convergent points between the given startpoint-endpoint pairs and compare timing relationships to determine matches/mismatches at these points. No ambiguity is expected at this phase as this is the finest level of comparison that can be made at path level. For any missing false paths encountered in the merged mode, we add the required false path constraint to the merged mode to resolve the mismatch.

Table 4 shows the timing relationships for the sample constraints being compared in pass 3. It can be observed that there is a mismatch in timing relationships for the path from rC/CP through $inv3/A$ to rZ/D. To address this, we add the constraint CSTR3 to the merged mode $A + B$ in the Constraint Set 6.

Start point	**Thro-ugh**	**End point**	**Lau-nch clock**	**Capt-ure clock**	**Indiv. mode state**	**Merged mode state**	**Pass3 re-sult**
rC/CP	$and2/A$	rZ/D	$clkA$	$clkA$	V	V	M
rC/CP	$inv3/A$	rZ/D	$clkA$	$clkA$	FP	V	X

Table 4: Timing relationship comparison table for pass 3 [FP: False Path, V: Valid, M: Match, X: Mismatch, A: Ambiguous]

4. RESULTS

The proposed approach has been implemented with a multi-threaded engine in C++ and tested in an industrial environment. Although the merged modes can be used in various design phases such as place & route, we present results on using our merged modes to perform STA.

Table 5 shows the results on mode reduction and the mode merging runtime on various industrial designs. Table 6 shows the STA runtime for these designs with the individual modes versus the merged modes and provides the runtime improvement and QoR conformity. The mode merging and STA on all the designs were run on a single machine with 4 cores. The delay calculations in STA were performed using wire load model approach. The number of modes of these designs have reduced by an average of 67.5% and with this reduction a runtime improvement of 62.52% was observed for performing STA. Note that the mode merging runtime adds as a one-time overhead, but the significant reduction in STA runtime overweighs this as it is often required to perform STA multiple times in a design cycle, for example in an ECO flow.

The benefit of mode merging is shown here in terms of runtime reduction. In a parallel environment, this can be translated into resource saving by the way of reducing the number of machines required to perform STA.

The quality of results of the merged modes is validated by comparing the worst slacks on all the endpoints on the designs obtained using the merged modes versus the individual modes. As shown in Table 6, it is observed that an average of 99.82% endpoint slacks have a deviation of within 1% of the capture clock period from the worst individual mode slacks.

Design	Size	# of Modes		% Reduction	Merging Runtime
		Individual	Merged		
A	0.2	95	16	83.1	6205
B	0.2	3	1	66.6	85
C	0.3	12	1	75.0	890
D	1.4	3	1	66.6	450
E	1.6	5	1	80.0	459
F	2.8	3	2	33.3	1424
			Average	67.5	

Table 5: Mode reduction and runtime on industrial designs [Units: Size → Millions of cells, Time → Seconds]

Design	Overall STA Runtime		% Reduction	Conformity
	Individual	Merged		
A	5584	875	84.3	99.89
B	339	140	58.7	100.00
C	820	398	51.5	99.91
D	1003	419	58.2	99.18
E	846	329	61.1	99.93
F	2593	1004	61.3	100.00
		Average	62.52	99.82

Table 6: Reduction in overall STA runtime and QoR of merged modes [Units: Time → Seconds; Conformity: % of EndPoints that have slack deviation within 1% of capture clock period]

5. CONCLUSION

In this paper, we have presented a novel technique to perform mode merging using timing relationship comparison on timing graph. With the behavior comparison we perform between the individual and merged constraints, we guarantee that the modes merged with this technique will be of sign-off accuracy. With an exhaustive validation of our approach on several industrial designs, we prove that our method is very practical and can be applied on complex real world designs to gain significant runtime and resource advantages.

6. REFERENCES

[1] Jing-Jia Nian; Shih-Heng Tsai; Shao-Lun Huang, ”A unified Multi-Corner Multi-Mode static timing analysis engine,” ASP-DAC, 2010

[2] Onaissi, S.; Taraporevala, F.; Jinfeng Liu; Najm, F., ”A fast approach for static timing analysis covering all PVT corners,” DAC, 2011

[3] Rajagopal, K.A.; Sivakumar, R.; Arvind, N.V.; Sreeram, C.; Visvanathan, V.; Dhuri, S.; Chander, R.; Fortner, P.; Sripada, S.; Qiuyang Wu, ”A comprehensive solution for true hierarchical timing and crosstalk delay signoff,” VLSI Design, 2006.

[4] Subrangshu K. Das; Ajay J. Daga; Aishwarya Singh; Vikas Sachdeva, ”The Automatic Generation of Merged-Mode Design Constraints”, DAC, 2009 User Track

[5] Shuo Zhou; Bo Yao; Hongyu Chen; Yi Zhu; Chung-Kuan Cheng; Hutton, M.; Collins, T.; Srinivasan, S.; Chou, N.; Suaris, P., ”Improving the efficiency of static timing analysis with false paths,” ICCAD-2005.

[6] ”Using the Synopsys Design Constraints Format”, Application Note, Version 2.0, 2013, Synopsys Inc.