OpenTimer: A High-Performance Timing Analysis Tool

Special Session Paper: Incremental Timing and CPPR Analysis

Tsung-Wei Huang* and Martin D. F. Wong†
*twh760812@gmail.com, †mdfwong@illinois.edu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—We introduce in this paper, OpenTimer, an open-source timing analysis tool that efficiently supports (1) both block-based and path-based timing propagations, (2) common path pessimism removal (CPPR), and (3) incremental processing. OpenTimer works on industry formats (e.g., .v, .spef, .lib, .sdc) and is designed to be parallel and portable. To further facilitate integration between timing and other electronic design automation (EDA) applications such as timing-driven placement and routing, OpenTimer provides user-friendly application programming interface (API) for inactive analysis. Experimental results on industry benchmarks released from TAU 2015 timing analysis contest have demonstrated remarkable results achieved by OpenTimer, especially in its order-of-magnitude speedup over existing timers.**

I. Introduction

The lack of accurate and fast algorithms for high-performance timing analysis tool with incremental capability has been recently pointed out as a major weakness of existing timing optimization flows [1]. In deep submicron era, timing-driven operations are imperative for the success of optimization flows. Optimization transforms change the design and therefore have the potential to significantly affect timing information. The timer must reflect such changes and update timing information incrementally and accurately in order to ensure slack integrity as well as reasonable turnaround time and performance [3]. However, such process requires extremely high complexity especially when path-based analysis is configured [4], [5], [6]. A high-quality incremental timer capable of path-based analysis is definitely advantageous in speeding up the timing closure.

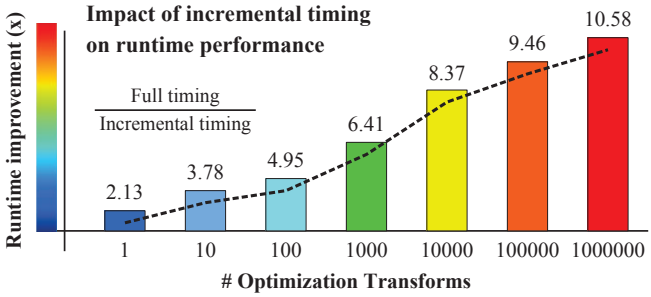


Figure 1. Performance improvement of incremental timing to full timing on one benchmark from [1].

The significance of incremental timing is demonstrated in Figure 1. It is observed that the runtime improvement keeps growing as the number of optimization transforms increases. One obvious reason is that once the critical paths in a design have been reported, the optimization tool would optimize the logic (e.g., gate sizing, buffer insertion) so as to overcome the timing violations. This subtle change can affect up to the majority of a circuit, whereas in reality, depending on the trace of critical paths, the timing update may only involve a small portion of the circuit. Since an optimization tool can perform millions of logic transformations, it is important that the timing profile is kept up-to-date in an incremental fashion. Otherwise, optimization tools cannot support fast turnaround for timing-specific improvement, which dramatically degrades the productivity.

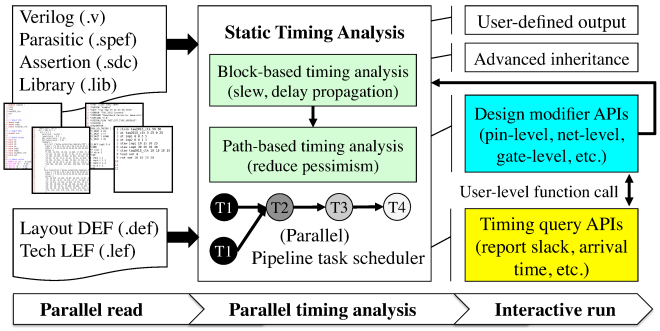


Figure 2. Program flow of OpenTimer.

Besides being incremental, one important feature of a practical timer is the capability of common path pessimism removal (CPPR). CPPR is a path-specific timing update that intends to remove redundant pessimism incurred by common segments between data paths and clock paths. Unwanted pessimism might force designers and optimization tools to waste a unnecessary yet significant amount of efforts on fixing paths that meet the intended clock frequency. This problem becomes even more critical when design comes to deep submicron era where data paths are shorter, clocks are faster, and clock networks are longer to accommodate larger and complex chips. However, the real problem is the amount of pessimism that needs to be removed is path-specific. Computational complexity and space requirements for CPPR typically grows exponentially as the design size increases, not to mention the challenge in conjunction with incremental timing analysis. Consequently, in this paper we introduce OpenTimer, an open-source high-performance timing analysis tool. An overview of OpenTimer is shown in Figure 2. We highlight three key features of OpenTimer as follows:

- **Parallel framework.** OpenTimer applies a pipeline task scheduler as the central engine. Critical tasks such as timing propagation and endpoint slack calculation are scheduled into the pipeline so as to overlap their runtimes.
- **Incremental capability.** OpenTimer precisely and minimally captures the features that are key to incremental timing. With lazy evaluation, we are able to keep computation as minimum as necessary.
- **Path-based analysis.** OpenTimer represents the path implicitly using efficient and compact data structure, yielding a significant saving in both search space and search time for CPPR.

The effectiveness and efficiency of our timer have been evaluated on a set of industry benchmarks released from TAU 2015 CAD contest. Compared to the top performers in TAU 2015 CAD contest, OpenTimer confers a high degree of differential in nearly all aspects. The source code of OpenTimer has been released to the public domain for promoting further research [2].

978-1-4673-8388-2/15/$31.00 ©2015 IEEE

II. Incremental Timing Analysis and CPPR

Various stages of the design flow such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis [1]. During these stages, local operations such as gate sizing, buffer insertion, or net rerouting can modify small fractions of the design and significantly change both local and global timing landscape. As the example shown in Figure 3, a change on gate B3 has the potential to affect up to the majority of the circuit (downstream timing). Nevertheless, depending on the trace of critical paths, only a small portion of the timing would need to be updated. For instance, if such a change does not affect the arrival time at I1:o, then every downstream timing after I1:o is unaffected.

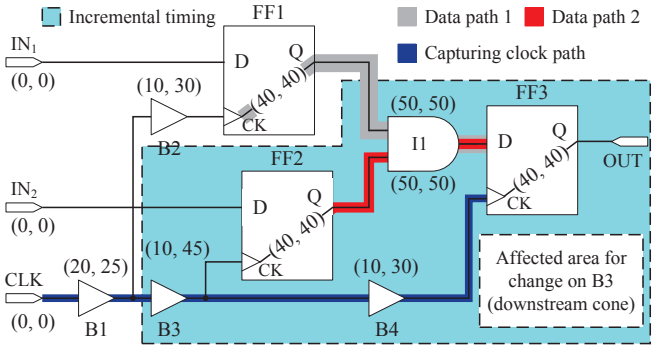


Figure 3. An example of sequential circuit network.

In addition to incremental processing, the capability of CPPR is another important component for modern timing analysis tools. Optimization transforms on the data network have no impact on CPPR credit (or CPPR adjustment) for any given launch-capture flip-flop (FF) pairs. Because the clock paths are not changed, any cached value for CPPR credit can be reused. However, in reality many optimization transforms are applied to the clock network, such as resizing a buffer or adding or deleting buffers on the clock tree in order to meet slack or skew targets. These changes can potentially affect a large number of data paths and slacks, and these data points must be recomputed with updated CPPR credits. Further, in some cases, changes on the clock network may not even impact CPPR for any data paths at all. As the example shown in Figure 3, the change on B3 can impact the CPPR credit for the launch-capture FF pair FF2 and FF3, while a change on B4 does not affect the CPPR credit for any FF pair. Therefore, the challenge of incremental CPPR is correctly identifying what data points are affected by which changes in an incremental manner.

III. Tool Configuration

The industry-standard format for timing analysis requests the following input files.

- **Two liberty (.lib) files** that defines the *early* and *late* characteristics of available cells in a given design, including pin capacitance, delay and slew look-up tables (LUTs), and setup/hold timing guard for sequential elements.
- **A verilog (.v) file** that defines the net list and circuit topology in gate level for a given design, including primary input/output ports and connections among gates.
- **A parasitics (.spef) file** that defines the design parasitics of a set of nets as a resistive-capacitive (RC) network, including the capacitance of internal nodes and wire resistance between internal nodes.
- **A Synopsys design constraint (.sdc) file** that defines the design operating conditions, including the clock port, clock period, initial timing on primary input ports, and load capacitance of primary output ports.

Given these input files, develop a CPPR-aware incremental timer that supports incremental timing update subject to a set of design modifiers and reports the timing with CPPR of any queried data path or timing point. In this paper, the slack prior to and after CPPR is referred to as *pre-CPPR slack* and *post-CPPR slack*, respectively. Particularly, the timer adheres to the following operations.

- **insert_gate**: adds an unconnected gate.
- **repower_gate**: changes the size of a gate.
- **remove_gate**: removes a disconnected gate.
- **insert_net**: creates an empty net.
- **remove_net**: removes a net from the design.
- **read_spef**: asserts parasitics on existing nets.
- **disconnect_pin**: disconnects a pin from its net.
- **connect_pin**: connects a pin to a net.
- **report_at**: reports the arrival time at a pin for any rise/fall transition and early/late split.
- **report_rat**: reports the required arrival time at a pin for any rise/fall transition and early/late split.
- **report_slack**: reports the worst post-CPPR slack at a pin for any rise/fall transition and early/late split.
- **report_worst_paths**: reports the worst post-CPPR path either in the design or through a specified pin.

The first eight operations describe the gate-level, net-level, and pin-level modifications on the design topology. The last four operations probe the design to report timing information. In order to collaborate with optimization tools, the timer should process these operations in an interactive or online manner. That is, advanced input disclosure or offline preprocessing is prohibited.

IV. Algorithm

The overall framework of OpenTimer is presented in Algorithm 1. It first initializes the circuit based on input liberty, verilog, parasitic, and Synopsys design constraint files. Then it enters the interactive while loop, reading the operation commands and processing each command accordingly.

Algorithm 1: OpenTimer(.lib, .v, .spef, .sdc)

Input: .lib, .v, .spef, .sdc files

```
1 initialize the circuit from input .lib, .v, .spef, and .sdc files;
2 while op ← GetOperationCommand do
3 |   process the operation command op accordingly;
4 end
```

A. State of the Art: UI-Timer

OpenTimer is built upon the state-of-the-art timer, UI-Timer (the winner of TAU 2014 CAD contest), which targets on one-time full timing update with CPPR [5]. OpenTimer succeeds the merits of UI-Timer, in particular its efficient data structures for pessimism retrieval and path search, and enhances it to be capable of incremental processing. For pessimism retrieval, we have implemented the LUT-based method by UI-Timer. Several LUTs are first built through the clock tree. Based on these LUTs, the amount of pessimism can be quickly retrieved by referring to the lowest-common ancestor (LCA) between tree nodes.

The second idea we borrowed from UI-Timer is the implicit representation of path. UI-Timer proposed two complementary data structures, namely suffix tree and prefix tree, to represent the search space of the path ranking. The suffix tree represents the shortest path tree rooted at a referenced node. The prefix tree is a tree order of non-suffix-tree edges such that each tree node represents the path being deviated on the corresponding edge from its ordinary trace in the suffix tree. Each path can be implicitly stored by the two data structures and the memory usage and the search time can be significantly reduced to constant time per path during the search. In the following sections, we shall focus on the major contributions of OpenTimer, while algorithmic details of pessimism retrieval and path ranking can be referred to [5].

B. Topological Ordering and Incremental Levelization

In timing analysis, the circuit is interpreted as a set of pin-to-pin connections or a directed acyclic graph (DAG) $G = \{P, E\}$, where P is the pin set and E is the edge set. Because of this special property, every pin p in the circuit graph can be levelized by a level index "*level*[p]" such that the topological order among different pins are maintained. The timing can thus be propagated level by level without destroying the circuit topology. In fact, we observe three major advantages of the topological levelization:

- Incremental timing can be achieved via the insertion of frontier pins from which the timing propagation originates.
- Using the level indices, timing can be propagated in a pipeline fashion as dependencies can be scheduled into different levels.
- Multi-threading is highly scalable since the timing in a given level can be propagated simultaneously.

As a result, we construct a bucket list as the core data structure for timing propagation. Each bucket is associated with a level index l and has a list storing those pins with level indices equal to l. The bucket list also records the minimum and maximum level indices of non-empty pin lists. Starting from the pin list in the lowest level, the function of incremental levelization is presented in Algorithm 2. In a rough view, Algorithm 2 iteratively levelizes a pin from the lowest level to the highest level (line 2:16). Once the pin is levelized, all its fanout pins are inserted to the bucket list (line 8:13).

Algorithm 2: IncrementalLevelization(B)

```
Input: bucket list B
Output: level indices of pins
1  l ← B.min_nonempty_level;
2  while l ≤ B.max_nonempty_level do
3      for p ∈B.pinlist(l) do
4          for p⁻ ∈ p.fanin_pins do
5              level[p] ← max(level[p⁻] + 1, level[p]);
6          end
7          B.insert(p);
8          for p⁺ ∈ p.fanout_pins do
9              if level[p] + 1 > level[p⁺] then
10                 level[p⁺] = level[p] + 1;
11             end
12             B.insert(p⁺);
13         end
14     end
15     l ← l + 1;
16 end
```

Lemma 1: *Denoting the downstream pin set of a pin as D_p^+, for every pin p in the bucket list B, we have $\{p' \in B \mid p' \in D_p^+\}$ after Algorithm 2.*

C. Forward Timing Propagation

Using the levelized bucket list, we develop the procedure of forward timing propagation. The forward timing propagation performs six tasks, *RC propagation*, *slew propagation*, *delay propagation*, *arrival time propagation*, *jump point propagation*, and *CPPR credit propagation*, for every pin in the bucket list level by level. RC propagation updates the RC parameters that are required for slew and delay propagations through a net. Slew propagation propagates the slew from an input cell pin to the output cell pin through a cell or an output cell pin to multiple input cell pins through a net. Delay propagation computes the edge delay through cells and nets. Similar to slew propagation, arrival time propagation propagates the arrival time through delay values on cell edges or net edges. In jump point propagation, we contract the graph in order to reduce the search space. CPPR credit propagation computes the amount of pessimism to be removed for a timing test.

1) RC Propagation: In this paper, we adopt the parasitic protocol by [1], where the output slew and delay through the RC network of a net are approximated by the symmetric of the value of the first and second moments of the impulse response. The approximation can be parameterized in a way such that the output slew and delay are functions of these RC parameters. Therefore, the goal of the RC propagation is to compute these RC parameters for any RC network. While the details are referred to [1], Algorithm 3 presents the procedure of RC propagation on the RC networks in a given level.

Algorithm 3: PropagateRC(l)

```
Input: level index l
1  B ← bucket list of the timer;
2  for p ∈ B.pinlist(l) do
3      if p.is_rc_network_root = true then
4          n ← p.net;
5          if n.is_rc_up_to_date = false then
6              update RC parameters for net n;
7          end
8      end
9  end
```

2) Slew and Delay Propagation: The propagations of slew and delay are carried out by Algorithm 4 and Algorithm 5, respectively. For each pin from a given level, the slew and delay to this pin are propagated from its fanin through either the RC network using pre-computed RC parameters (line 7:8 in Algorithm 4 and line 4:5 in Algorithm 5) or the cell timing arc where the values are obtained via extrapolation or interpolation on the corresponding slew and delay LUTs (line 10:12 in Algorithm 4 and line 7:9 in Algorithm 5).

3) Arrival Time Propagation: The propagation of arrival time is trivial once the delay value on each edge is ready. It has been shown that finding the earliest and latest arrival time in the circuit graph is equivalent to finding the shortest and longest paths in a DAG, which can be fulfilled using levelized propagation [3]. Algorithm 6 presents such propagation at a given level.

4) Jump Point Propagation: Reducing the size of timing graph is an effective way to speed up the path search. Because of intrinsic properties of cells, many paths are present in a tree form. To be more specific, for some pin pairs at certain transitions, the paths in between are uniquely defined. For instance, the AND gate in Figure 4 is unate-definite (i.e., either positive unate or negative unate), and hence any paths passing through are not diverged. Starting from pin FF3:D at any transition, there exists only one path back to pin FF1:Q

Algorithm 4: PropagateSlew(l)

```
Input: level index l
B ← bucket list of the timer;
for p ∈ B.pinlist(l) do
    if p.num_fanins = NULL then
        assign slew to p from the primary input;
    else
        for e ∈ p.fanin_edges do
            if e.is_net_edge = true then
                propagate slew to p through rc-timing on e.net;
            else
                if e.is_constraint_edge = false then
                    propagate slew to p through LUT on e;
                end
            end
        end
    end
end
```

Algorithm 5: PropagateDelay(l)

```
Input: level index l
B ← bucket list of the timer;
for p ∈ B.pinlist(l) do
    for e ∈ p.fanin_edges do
        if e.is_net_edge = true then
            update delay of e through rc-timing on e.net;
        else
            if e.is_constraint_edge = false then
                update delay of e through LUT on e;
            end
        end
    end
end
```

or pin FF2:Q. Consequently, we can construct a shortcut that allows the path search to jump over the subcircuit from FF2:Q or FF1:Q to FF3:D. In this case, the pin FF3:D is named as "*jump head*" and pins FF2:Q and FF1:Q are named as "*jump tail*".

The examination of whether a pin is a jump head or a jump tail is presented in Algorithm 7–8. It can be analogized to a tree where the jump head is the root and the jump tail is the leave. As shown in Algorithm 7, a pin at any transition and timing split is referred to as a jump head only if its output signal is not branched. On the other hand, the jump tail is determined by whether its input signal is uniquely defined. Using Algorithms 7–8, the construction and propagation of jump points are given by Algorithms 9–10. In a rough view, Algorithm 9 induces the jump point connection through a recursive traversal to discover any tree-structured subcircuit. Algorithm 10 applies Algorithm 9 to each pin in a give level. Notice that jump point connections are only considered among data network.

Algorithm 6: PropagateArrivalTime(l)

```
Input: level index l
B ← bucket list of the timer;
for p ∈ B.pinlist(l) do
    if p.num_fanins = 0 then
        assign arrival time to p from the primary input;
    else
        for e ∈ p.fanin_edges do
            propagate arrival time to p through delay on e;
        end
    end
end
```

Algorithm 7: is_jump_head(p)

```
Input: an existing pin p
if p.num_fanouts = 0 or p.num_fanouts > 1 or p.is_in_clock_tree then
    return true;
end
e ← p.fanout_edges;
return e.timing_sense = non_unate;
```

Algorithm 8: is_jump_tail(p)

```
Input: an existing pin p
if p.num_fanins = 0 then
    return true;
end
for e ∈ p.fanin_edges do
    if e.is_constraint_edge = false then
        head ← is_jump_head(e.from_pin);
        if head = true then
            return true;
        end
    end
end
return false;
```

Algorithm 9: induce_jump_point(p, p', d)

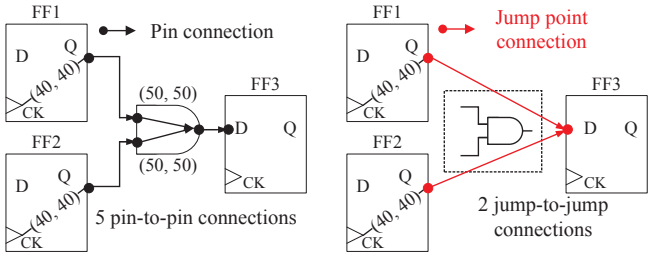
```
Input: two pins p and p' and a delay value d
p.jump_head ← p';
if is_jump_tail(p) = true then
    if p ≠ p' then
        insert a jump connection from p to p' with delay d;
    end
    return;
end
for e ∈ p.fanin_edges do
    p⁻ ← e.from_pin;
    if e.is_constraint_edge = true or p⁻.is_in_clock_tree = true then
        continue;
    end
    induce_jump_point(p⁻, p', d + e.delay);
end
```

Algorithm 10: PropagateJumpPoint(l)

```
Input: level index l
B ← bucket list of the timer;
for p ∈ B.pinlist(l) do
    if p.is_in_clock_tree = true or is_jump_head(p) = false then
        return;
    end
    induce_jump_point(p, p, 0);
end
```

Algorithm 11: PropagateCPPRCredit(l)

```
Input: level index l
B ← bucket list of the timer;
for p ∈ B.pinlist(l) do
    t ← p.timing_test;
    if t = NULL or t.is_sequential_test = false then
        continue;
    end
    # Fork_Thread_Task {
        path ← GetCriticalPath(t, 1) [5];
        t.cppr_credit ←path.cppr_credit;
    };
end
```



(a) Ordinary circuit graph (b) Contracted circuit graph

Figure 4. Graph contraction using jump-point connections.

5) CPPR Credit Propagation: For each data pin of a FF that is guarded by setup tests or hold timing tests, we need to discover the corresponding CPPR credit for slack adjustments [4]. The CPPR credit is defined as the numeric that is applied to skew the worst post-CPPR slack of a particular test [1]. As aforementioned, the state-of-the-art path tracing algorithm by UI-Timer [5] is our default engine for the investigation of CPPR credits for any timing tests. The algorithm of CPPR credit propagation is presented in Algorithm 11. In contrast to UI-Timer where the search graph is induced from the flattened circuit graph, we are able to reduce the search space with jump points which can lead to significant speedup. Because of the independence of timing tests, the path tracing can be performed in a parallel manner (line 7:10).

D. Backward Timing Propagation

In contrast to forward timing propagation, the backward timing propagation propagates the timing for every pin in the bucket list from the highest level to the lowest level by performing two major tasks, *fanin propagation* and *required arrival time propagation*. Fanin propagation inserts the fanin of each pin from the bucket list in order to construct the upstream cone. Required arrival time propagation propagates the timing constraint in a backward manner.

1) Fanin Propagation: In order to perform backward timing propagation, we need to construct the upstream cone of every pin in the bucket list. Considering the procedure in Algorithm 12 which inserts all fanin pins from a pin list in a given level, the upstream cone for backward timing propagation can be constructed by calling this procedure level by level.

Algorithm 12: PropagateFanin(l)

Input: level index l

```
1  B ← bucket list of the timer;
2  for p ∈ B.pinlist(l) do
3      for p⁻ ∈ p.fanin_pins do
4          B.insert(p⁻);
5      end
6  end
```

2) Required Arrival Time Propagation: The propagation of required arrival time in a given level is shown in Algorithm 13. Algorithm 13 exerts similar procedure as Algorithm 6 but in a reversed direction (line 8:10). For constrained pin, the required arrival time is assigned by the constraint value from the corresponding timing test (line 4:5) and is adjusted by the CPPR credit in case of sequential timing tests (line 6).

Algorithm 13: PropagateRequiredArrivalTime(l)

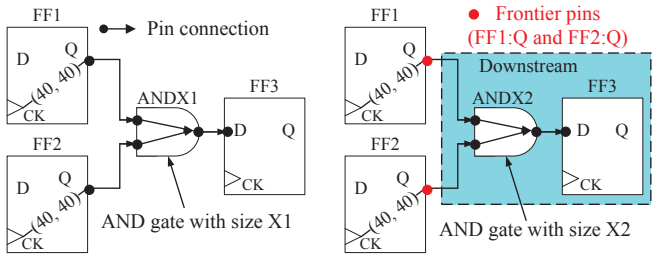
Input: level index l

```
1  B ← bucket list of the timer;
2  for p ∈ B.pinlist(l) do
3      if p.num_fanouts = 0 then
4          t ← p.timing_test;
5          assign required arrival time to p from t;
6          adjust required arrival time with CPPR credit from t;
7      else
8          for e ∈ p.fanout_edges do
9              propagate required arrival time to p through delay on e;
10         end
11     end
12 end
```

E. Design Modification

Based on the levelized bucket list, the objective of dealing with design modifiers is to identify the set of "*frontier pins*" from which the incremental timing update originates. Starting at the frontier pins, Algorithm 2 constructs a downstream cone of affected area which will be used for incremental timing update. We consider the design modifiers at gate level, net level, and pin level.

1) Gate-Level Modifications: The operations that modify the design at gate level are 1) insert_gate, 2) remove_gate, and 3) repower_gate. Recall that the operation insert_gate creates a new gate in the design and the operation remove_gate removes a disconnected gate from the design. It is obvious that the two operations introduce no frontier pins as the gate being inserted or removed is not connected to the current circuit. Therefore, for gate-level design modifiers we only deal with the operation repower_gate.



(a) A circuit fragment (b) Repower gate (X1→X2)

Figure 5. A design modification by repowering the gate with another size (repower_gate).

An example of the operation repower_gate is shown in Figure 5. The AND gate in the data network is repowered from size X1 (cell ANDX1) to size X2 (cell ANDX2). Repowering a gate changes the cell timing and the pin capacitance. The affected area should be traced back by one level where the pins connecting the gate originate the incremental timing. In this example, the incremental timing propagation is captured by two frontier pins FF1:Q and FF2:Q. Using this fact, our solution to the operation repower_gate is presented in Algorithm 14. Algorithm 14 first replaces the cell that was attached to the gate with the new cell (line 1). Afterward the frontier pins, which are fanin pins of each input pin of the gate, are inserted into the bucket list (line 3:9) for incremental timing update.

2) Net-Level Modifications: There are three operations that modify the design at net level: 1) insert_net, 2) remove_net, and 3) read_spef. Similar to gate-level modifications, the operation insert_net creates an empty (disconnected) net for the design and the operation remove_net

Algorithm 14: repower_gate(g, c)

Input: an existing gate g, a new cell c

```
1 remap the gate g to the new cell c;
2 B ← bucket list of the timer;
3 for p ∈ g.input_pins do
4     for p⁻ ∈ p.fanin_pins do
5         B.insert(p⁻);
6         n ← p⁻.net;
7         n.is_rc_up_to_date ← false;
8     end
9 end
```

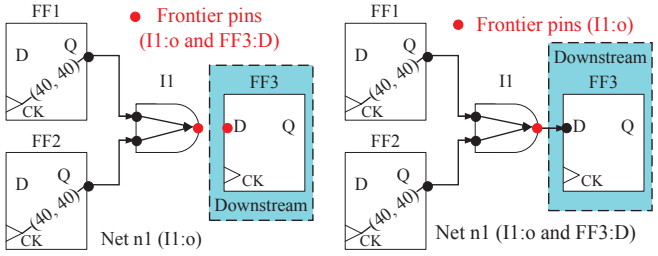
deletes an empty net from the design. Due to the isolation, both operations have no impact on current timing profile. The net-level design modifier read_spef is the only operation that could affect the timing. Our solution to read_spef is presented in Algorithm 15. Algorithm 15 first parses the given .spef file into an object (line 1). Then it iterates each net that was parsed from the .spef file and asserts the new parasitics to it (line 3:4). Whenever the parasitics of a net change, the incremental timing update is captured by the root of the corresponding RC network (line 5:7).

Algorithm 15: read_spef(.spef)

Input: a .spef file

```
1 O ← parse .spef file into an object;
2 B ← bucket list of the timer;
3 for net n ∈ O do
4     update the parasitics of net n through O;
5     n.is_rc_up_to_date ← false;
6     p_r ← n.rc_network_root_pin;
7     B.insert(p_r);
8 end
```

3) Pin-Level Modifications: The pin-level design modifiers are the most crucial operations since they directly alter the connectivity in the design. There are two operations that modify the design at pin level: 1) disconnect_pin and 2) connect_pin. The operation disconnect_pin disconnects the pin from the net it is connected to and the operation connect_pin connects the pin to a given net. Both operations alter the structure of the design and directly affect the timing. Consequently, we need to identify the frontier pins that capture the incremental timing update for such changes.



(a) Disconnect pin FF3:D from net n1 (b) Connect pin FF3:D to net n1

Figure 6. A design modification by disconnecting/connecting a pin from/to a net (disconnect_pin/connect_pin).

An example for operations disconnect_pin and connect_pin are given in Figure 6. It can be seen from (a) disconnecting the pin I1:o from its net cuts off the connection from I1:o to FF3:D. This change affects the timing at the pins I1:o and FF3:D as well as the downstream cone of the pin FF3:D. Therefore, disconnecting a pin introduces two frontier pins that are the two end points at the connection to or from which the pin is connected. On the other hand, connecting a pin to a given net establishes a new connection. In (b), connecting the pin I1:o to the net n1 produces a new connection from the pin I1:o to the pin FF3:D. This change has impact on the timing profile in the downstream cone of pin I1:o. As a result, connecting a pin introduces one frontier pin which is the tail of this connection. Algorithms 16–17 present our solutions to pin-level operations. Notice that a pin is considered either a root of the RC network where we need to remove or insert all possible connections, including the jump point connection that covers such a change (line 4:8 in Algorithm 16 and line 2:7 in Algorithm 17), or the terminal of the RC network in which case we deal with the only one connection (line 10 in Algorithm 16 and line 9:11 in Algorithm 17).

Algorithm 16: disconnect_pin(p)

Input: an existing pin p

```
1  n ← p.net ;
2  p_r ← n.rc_network_root_pin ;
3  B ← bucket list of the timer;
4  if p = p_r then
5      for p′ ∈ n.pinlist −{p_r} do
6          B.insert(p′);
7          disconnect pin p′ from the net n;
8      end
9  else
10     B.insert(p_r);
11 end
12 B.insert(p);
13 disconnect all jump point connections to p.jump_head;
14 disconnect the pin p from the net n;
```

Algorithm 17: connect_pin(p, n)

Input: an existing pin p and an existing net n

```
1  B ← bucket list of the timer;
2  if p.is_rc_network_root_pin = true then
3      for p′ ∈ n.pinlist do
4          establish the connection from p to p′;
5          disconnect all jump point connections to p′.jump_head;
6      end
7      B.insert(p);
8  else
9      p_r ← n.rc_network_root_pin ;
10     establish the connection from p_r to p;
11     B.insert(p_r);
12 end
13 disconnect all jump point connections to p.jump_head;
14 connect the pin p to the net n;
```

F. Incremental Timing Update

Based on Algorithms 2–17, we are able to deliver the key procedure for incremental timing update. In order to guarantee correct timing results, the task dependency among different timing propagations needs to be carefully addressed. For backward timing propagation in a given level, the procedures of fanin propagation and required arrival time propagation are apparently independent to each other. However, for forward timing propagation in a given level, the following dependency should be satisfied: 1) RC propagation (RCP) precedes the slew propagation (SLP) and delay propagation (DLP); 2) DLP precedes the arrival time propagation (ATP); 3) ATP precedes the jump point propagation (JMP); 4) JMP precedes the CPPR credit

propagation (CRP). As the timing propagation is conducted level by level, the task dependency can be efficiently encapsulated by a parallel pipeline. Figure 7 illustrates this concept (subscript delineates the level index).

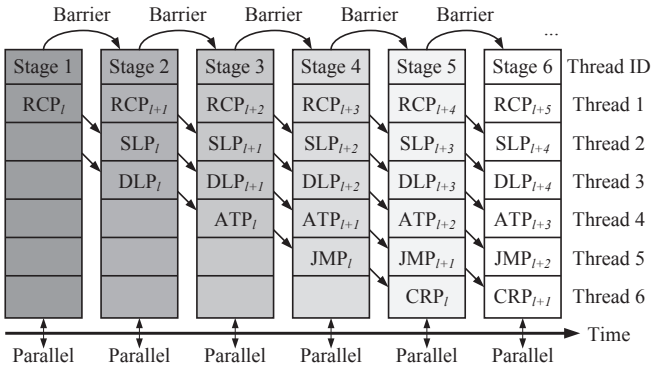


Figure 7. Parallel forward timing propagation using pipeline.

Algorithm 18: update_timing()

```
1  B ← bucket list of the timer;
2  if B.num_pins = 0 then
3  |  return;
4  end
5  IncrementalLevelization(B);
6  l_min ← B.min_nonempty_level;
7  l_max ← B.max_nonempty_level;
8  # Parallel_Region {
9  # Master_Thread_do for l = l_min to l_max + 4 do
10 |  # Fork_Thread_Task PropagateRC(l);
11 |  # Fork_Thread_Task PropagateSlew(l − 1);
12 |  # Fork_Thread_Task PropagateDelay(l − 1);
13 |  # Fork_Thread_Task PropagateArrivalTime(l − 2);
14 |  # Fork_Thread_Task PropagateJumpPoint(l − 3);
15 |  # Fork_Thread_Task PropagateCPPRCredit(l − 4);
16 |  # Synchronize_Thread_Tasks;
17 end
18 };
19 # Parallel Region {
20 # Master_Thread_do for l = l_max to B.min_non_empty_level do
21 |  # Fork_Thread_Task PropagateFanin(l);
22 |  # Fork_Thread_Task PropagateRequiredArrivalTime(l);
23 |  # Synchronize_Thread_Tasks;
24 end
25 };
26 remove all pins from the bucket list B;
```

Algorithm 18 presents our solution to incremental timing update. It first calls Algorithm 2 to construct the downstream cone of all frontier pins in the bucket list (line 5). The timing propagation is then performed level by level in a parallel pipeline fashion (line 8:18 for forward timing propagation and line 19:25 for backward timing propagation). By the end of each pipeline stage, a barrier is imposed to synchronize all forked threads (line 16 and line 23). The bucket list is reset after the timing propagation is accomplished (line 26).

G. Timing Query

Using Algorithm 18 as the infrastructure, the value-based timing queries, for example, reporting the arrival time, can be implemented as Algorithm 19. Queries for required arrival time and slack can be mimicked in a similar manner. The path-based query is presented in Algorithm 20. Algorithm 20 takes two arguments, one pin p and a path count K, and reports the top K post-CPPR critical paths through p. If p is nil, the paths are searched across the entire circuit graph G (line 2). Otherwise, the search graph is limited to the region of downstream cone D_p^+ and upstream cone D_p^- of p such that every path discovered in the search graph passes through p (line 3:5). Then we apply the path ranking algorithm by [5] to peel out the top K critical tests (line 6). Finally we iteratively extract the top K critical paths from each of the top K critical tests and maintain the globally top K critical paths using a priority queue (line 7:15).

Algorithm 19: report_at(p, s, m)

Input: an existing pin p and targeted transition s and timing split m
Output: arrival time at p for s and m

```
1  update_timing();
2  return p.arrival_time(s, m);
```

Algorithm 20: report_worst_path(p, K)

Input: an existing pin p and a path count K
Output: top K critical paths through p in the design

```
1  update_timing();
2  G' ← G;
3  if p ≠ NULL then
4  |  G' ← D_p^- ∪ D_p^+;
5  end
6  extract a sorted set T of the top K post-CPPR critical tests from G' [5];
7  Q ← priority queue keyed on post-CPPR slack values;
8  for t ∈ T do
9  |  if Q.size = K and t.slack ≥ Q.top_max then
10 |  |  break;
11 |  end
12 |  Q ← Q∪ GetCriticalPath(t, K) [5];
13 |  Q.maintain_top_k_min(K);
14 end
15 return Q;
```

V. Experimental Results

OpenTimer is implemented in C++ language on a 2.20 GHz 64-bit Linux machine with 128 GB memory. The application programming interface (API) provided by OpenMP 3.1 is used for our multi-threaded programming. Our machine can execute a maximum of 8 threads concurrently. Experiments are undertaken on a set of industry benchmarks released from TAU 2015 CAD contest [1]. The golden reference is generated from an industry timer and the design modifiers are wrapped in a .ops file which contains tens of millions of operations. Table I lists the benchmark statistics and the performance of OpenTimer compared to the top performers, "iTimerC 2.0" and "iitRACE," from TAU 2015 CAD contest [1].

We begin by comparing OpenTimer with iitRACE. The strength of OpenTimer is clearly demonstrated in the accuracy and runtime values. We have seen a significant performance gap where our timer is much more accurate and far faster than iitRACE. Even though iitRACE achieves better memory usage, such data are less meaningful when accuracy is considered the top priority. Next we compare OpenTimer with iTimerC 2.0. In general, OpenTimer outperforms iTimerC 2.0 across nearly all circuit benchmarks in any aspects. We reach the goal by $\times$2.3 (edit_dist) to $\times$9.7 (cordic_core) faster and consume less memory for most benchmarks. In addition, our accuracy is higher than iTimerC 2.0 by 7% in average. Unfortunately, we are unable to compare the data on the benchmark softusb_navre because iTimerC 2.0 encountered execution fault.

TABLE I
PERFORMANCE COMPARISON BETWEEN OPENTIMER AND TOP-RANKED TIMERS IITRACE AND ITIMERC 2.0 FROM TAU 2015 CAD CONTEST [1].

Circuit	#Gates	#Nets	#OPs	iitRACE			iTimerC 2.0			OpenTimer		
				accuracy	runtime	memory	accuracy	runtime	memory	accuracy	runtime	memory
b19	255.3K	255.3K	5641.5K	63.03 %	629 s	3.0 GB	99.95 %	215 s	5.8 GB	99.95 %	52 s	4.6 GB
cordic	45.4K	45.4K	1607.6K	61.83 %	100 s	0.9 GB	98.88 %	80 s	1.3 GB	98.88 %	18 s	1.3 GB
des_perf	138.9K	139.1K	4326.7K	67.43 %	299 s	4.2 GB	97.02 %	92 s	3.1 GB	99.73 %	30 s	3.0 GB
edit_dist	147.6K	150.2K	3368.3K	64.83 %	857 s	2.0 GB	98.29 %	98 s	3.8 GB	98.30 %	42 s	3.8 GB
fft	38.2K	39.2K	1751.7K	89.66 %	70 s	0.5 GB	98.45 %	49 s	1.2 GB	99.77 %	11 s	1.2 GB
leon2	1616.4K	1517.0K	8438.5K	72.34 %	16832 s	9.9 GB	100.00 %	787 s	27.2 GB	100.00 %	282 s	22.8 GB
leon3mp	1247.7K	1248.0K	8405.9K	62.99 %	4960 s	8.2 GB	100.00 %	609 s	19.8 GB	100.00 %	163 s	17.9 GB
mgc_edit_dist	161.7K	164.2K	3403.4K	64.29 %	1578 s	1.9 GB	100.00 %	135 s	4.1 GB	100.00 %	41 s	3.1 GB
mgc_matrix_mult	171.3K	174.5K	3717.5K	67.93 %	1363 s	2.0 GB	100.00 %	157 s	4.3 GB	100.00 %	31 s	3.1 GB
netcard	1496.0K	1497.8K	11594.6K	87.63 %	6662 s	9.4 GB	99.99 %	691 s	22.9 GB	99.99 %	192 s	20.8 GB
cordic_core	3.6K	3.6K	226.0K	59.42 %	21 s	0.3 GB	95.19 %	29 s	0.2 GB	95.19 %	3 s	0.1 GB
crc32d16N	478	495	28.9K	57.15 %	3 s	0.1 GB	100.00 %	5 s	0.1 GB	100.00 %	1 s	0.1 GB
softusb_navre	6.9K	7.0K	427.8K	40.17 %	21 s	0.1 GB	0.00 %	-	-	99.97 %	4 s	0.5 GB
tip_master	37.7K	38.5K	1300.4K	82.95 %	64 s	0.6 GB	96.42 %	47 s	1.0 GB	97.04 %	9 s	0.8 GB
vga_lcd_1	139.5K	139.6K	2961.5K	99.65 %	260 s	1.6 GB	100.00 %	94 s	2.2 GB	100.00 %	31 s	2.9 GB
vga_lcd_2	259.1K	259.1K	12674.7K	98.57 %	1132 s	13.3 GB	100.00 %	156 s	5.0 GB	100.00 %	65 s	3.9 GB

#Gates: number of gates. #Nets: number of nets. #OPs: number of operations. accuracy: average of path accuracy and value accuracy (%). -: program crash.

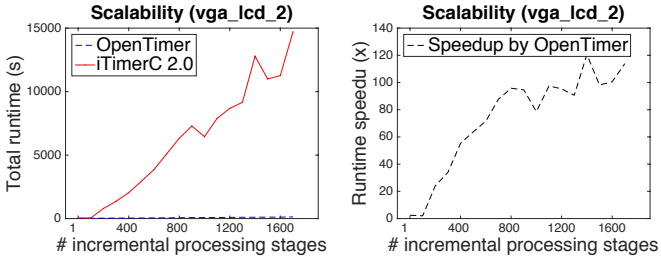


Figure 8. Scalability comparison between OpenTimer and iTimerC 2.0.

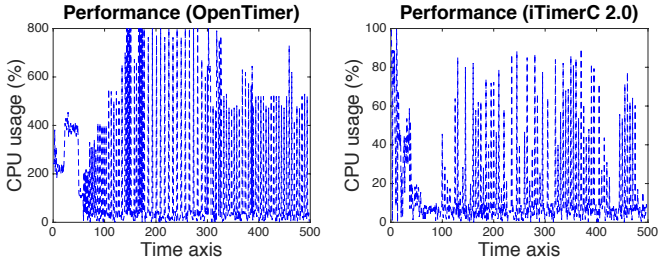


Figure 9. Parallelism comparison between OpenTimer and iTimerC 2.0.

Finally we investigate the scalability of our timer and iTimerC 2.0 on accommodating the depth of incremental processing. We omit the comparison with iitRACE because its low accuracy might result in unfairness. In this experiment, we refer to a set of design modifiers followed by at least one timing query as "one stage" of incremental processing. We have divulged, unfortunately, all benchmarks from TAU 2015 contest have less than 10 incremental processing stages, which is not sufficient to reveal the performance bottleneck. Therefore, we modified the benchmark vga_lcd_2 by inserting a path-based timing query after each complete design modification. The comparison of runtime scalability between OpenTimer and iTimerC 2.0 is demonstrated in Figure 8. It can be clearly seen that our runtime scales extremely well as the number of incremental processing stages increases. For instance, OpenTimer accomplished the goal by $\times 95.8$ faster (66 seconds vs 6324 seconds) than iTimerC 2.0 at the 800th stage. Similar trends can be observed on other stage numbers. We further reveals the cpu usage for both programs in Figure 9. It is observed OpenTimer is highly parallel, using up to the hardware-limited thread number, while iTimerC 2.0 does not support any multi-threaded feature. To sum up, these experiments have justified the practical viability of OpenTimer.

VI. CONCLUSION

In this paper we have presented OpenTimer, a high-quality incremental timing analysis algorithm with CPPR. We have not only captured the key features that achieve incremental capability, but also parallelized the incremental timing update in a pipeline fashion. Our framework is very flexible and scalable as many critical tasks such as timing propagation and CPPR are scheduled into the pipeline so as to overlap their runtimes. These advantages confer OpenTimer a high degree of differential over existing methods. Comparatively, experimental results have demonstrated the superior performance of OpenTimer in terms of accuracy, runtime, and memory over the top performers from TAU 2015 CAD contest.

ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation under Grant CCF-1320585 and CCF-1421563. The authors acknowledge Jin Hu, Myung-Chul Kim, and Pei-Yu Lee for helpful discussions and program debugging, and teams iitRACE and iTimerC 2.0 for sharing their binary in TAU 2015 CAD contest.

REFERENCES

[1] TAU 2015 Contest: Incremental Timing Analysis and Incremental CPPR, http://sites.google.com/site/taucontest2015

[2] OpenTimer: http://web.engr.illinois.edu/~thuang19/index.html

[3] J. Bhasker and R. Chadha, "Static Timing Analysis for Nanometer Designs: A Practical Approach," *Springer*, 2009.

[4] J. Hu, D. Sinha, and I. Keller, "TAU 2014 Contest on Removing Common Path Pessimism during Timing Analysis," *Proc. ACM ISPD*, pp. 153–160, 2014.

[5] T.-W. Huang, P.-C. Wu, and Martin D. F. Wong, "UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm," *Proc. IEEE/ACM ICCAD*, pp. 758–765, 2014.

[6] Y.-M. Yang, Y.-W. Chang, and Iris H.-R. Jiang, "iTimerC: Common Path Pessimism Removal Using Effective Reduction Methods," *Proc. IEEE/ACM ICCAD*, pp 600–605, 2014.