OpenTimer: A high-performance timing analysis tool

Special session invited paper

Tsung-Wei Huang and Martin D. F. Wong
Department of Electrical and Computer Engineering (ECE)
University of Illinois at Urbana-Champaign (UIUC), IL, USA
Outline

- Problem formulation recap
  - Incremental timing and incremental CPPR

- OpenTimer architecture
  - Tool overview and features
  - Core algorithm (IO, graph reduction, pipeline scheduling)

- Experimental results
  - TAU 2015 contest benchmarks

- Conclusion
Problem Formulation – TAU 2015 Contest

- A tool deals with incremental timing and incremental CPPR
  - Important for timing-driven applications
  - Fast full timing and incremental timing analysis
  - Capability of path-based CPPR analysis
  - Parallel programming and multi-threading

*CPPR stands for Common Path Pessimism Removal*
OpenTimer Architecture

- An open-source high-performance timing analysis tool
  - TAU14 (1\textsuperscript{st} place), TAU15 (2\textsuperscript{nd} place)
  - ICCAD15 (golden timer), TAU16 (golden timer)

Feature highlights
- C++11
- Industry format
- STA engine
- Block-based
- Path-based
- Incremental
- Lazy evaluation
- CPPR
- Multi-threaded

http://web.engr.illinois.edu/~thuang19/software/timer/OpenTimer.html
Initialize the Timer – Parallel IO

- A set of files in industry standard format
  - Verilog netlist, two libraries (early and late), parasitic spef, etc.
  - Time-consuming IO (e.g., file IO, complex parsing)

Parallel dependency generation using portable OpenMP

```
#pragma parallel ...
#pragma task ...
```

$x^2$ speedup by parallel read/parse!
Timing Graph Reduction

- Reduce the search space
  - Identify tree-structured subgraphs in the original timing graph
  - Merge every leaf-root path (transition-definite)

Every leaf-root path can be uniquely defined (given a transition at an endpoint)

- ~30% reduction on the graph size
- Apply to combinational circuits only
Key Components of Incremental Timing

- Full timing is just a special case of incremental timing
- Design modifiers
  - Pin-level operations, net-level operations, and gate-level operations
- Timing queries
  - Slack
  - Arrival time
  - Required time
  - TNS and WNS
  - Critical path report
  - CPPR
- Source of propagation
- Lazy evaluation
- Explore parallel incremental timing
Identify the Source of Incremental Timing (I)

- Source of incremental timing
  - Pins where the next timing propagation must originate from
  - Referred to as “frontier pins”
- Repower gate (gate sizing)
  - `repower_gate<gate_name, new_cell_name>`

(a) A circuit fragment
(b) Repower gate (X1→X2)
Identify the Source of Incremental Timing (II)

- Disconnect pin (disconnect a pin from its net)
  - `disconnect_pin <pin_name>`
- Affect RC network only
  - Incremental timing from the root

Find the root of the corresponding RC network (RC tree)

(a) A circuit fragment
(b) Connect pin FF3:D to net 1

Frontier pins (I1:o)

Frontier pins (I1:o and FF3:D)

Load cap changed
Identify the Source of Incremental Timing (III)

- Connect pin (connect a pin to a net)
  - `connect_pin <pin_name, net_name>`
- Affect RC network only
  - Incremental timing from the root

Find the root of the corresponding RC network (RC tree)

Frontier pins (I1:o)

Load cap changed

Frontier pins (I1:o and FF3:D)
Dealing with Design Modifiers

- Design modifiers can be built upon pin-level modification
  - remove_net: disconnect all pins and remove the net
  - insert_net: create an empty net
  - insert_gate: insert a gate and create pins
  - remove_gate: disconnect all input and output pins, and remove the gate
- …

- How to store frontier pins for correct timing propagation?
  - Timing graph is a directed acyclic graph (DAG)
  - Efficient data structure, bucketlist, to maintain the dependency
    - Apply topological sort (longest path finding)
    - Level index of each pin to keep track of dependency
    - Every frontier pin is inserted into the corresponding bucket
    - Incremental topological sort for incremental levelization
Pipeline-based Parallel Timing Propagation

- Timing propagation has several linearly dependent tasks
  - RC update → Slew & Delay → Arrival time → Jump point → CPPR
  - Pipeline scheduling with multiple threads

We use the following paper for dealing with CPPR
*UI-Timer: An ultra-fast clock network pessimism removal algorithm, T.-W. Huang, P.-C. Wu, and Martin D. F. Wong, ICCAD14
Experimental Results – Environment Setup

- Implementation
  - C++11 with GCC 4.8
  - Linux machine (8 cores)*

- Benchmark suite
  - TAU15 contest benchmarks

- Baseline on TAU15 winners
  - iTimerC 2.0 (1st place)
  - iitRACE (3rd place)

<table>
<thead>
<tr>
<th>Design</th>
<th>Pls</th>
<th>POs</th>
<th>Gates</th>
<th>Nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>vga_lcd</td>
<td>89</td>
<td>109</td>
<td>139.5K</td>
<td>139.6K</td>
</tr>
<tr>
<td>cordic_ispd</td>
<td>34</td>
<td>64</td>
<td>45.4K</td>
<td>45.4K</td>
</tr>
<tr>
<td>des_perf_ispd</td>
<td>234</td>
<td>140</td>
<td>139.8K</td>
<td>139.1K</td>
</tr>
<tr>
<td>edit_dist_ispd</td>
<td>2.6K</td>
<td>12</td>
<td>147.6K</td>
<td>150.2K</td>
</tr>
<tr>
<td>fft_ispd</td>
<td>1.0K</td>
<td>2.0K</td>
<td>38.2K</td>
<td>39.2K</td>
</tr>
</tbody>
</table>

- clock tree has inverters and buffers

7 based on released benchmarks (∼10^2 – ∼10^5 gates)
3 based on Cadence benchmarks (∼10^2 – ∼10^5 gates)
6 based on ICCAD 2014 benchmarks (∼10^5 – ∼10^6 gates)

*Campus cluster, University of Illinois at Urbana-Champaign (UIUC)
https://campuscluster.illinois.edu/
Experimental Results – Overall Performance Comparison

* **iTimerC 2.0**: 1\textsuperscript{st} place in TAU contest 2015 (binary from authors)

* **iitRACE**: 2\textsuperscript{nd} place in TAU contest 2015 (binary from authors)

Program crashes
(Unexpected error in iTimerC 2.0)
Experimental Results – Performance Highlights

- **OpenTimer performance highlights**
  - Achieved the highest accuracy (*both path-based and value-based*)
  - Achieved the fastest runtime (*x2 to x9 speedup*)
  - Robust and reliable (*no crash*)
  - Small memory usage*

- **Comparison to the *newest* results of iTimerC 2.0**
  - netcard (1.5M gates and 1.5M nets)
    - OpenTimer: 192s, 20GB
    - iTimerC 2.0: 213s, 21GB
  - leon3mp (1.2M gates and 1.2M nets)
    - OpenTimer: 163s, 18GB
    - iTimerC 2.0: 186s, 19GB

*iTimerC 2.0 runs in 3.5GHz CPU! OpenTimer runs in 2.2GHz CPU!*

* Our contest version has higher memory usage is due to the different system environment settings between IBM machine and UIUC machine

**iTimerC 2.0: Fast incremental timing and CPPR analysis, ICCAD15**
Experimental Results – Scalability of Incremental Timing

- **Optimization or synthesis tools**
  - Call an incremental timer millions of times to optimize the timing objective

- **One incremental processing stage**
  - A set of design modifiers followed by a timing query

- **Insufficiency of TAU15 benchmarks**
  - Only 5~10 incremental processing stages...

![Graphs showing scalability](image)

- **x115 speedup** by OpenTimer (at 1459th stage)
- **X2.7 speedup** by OpenTimer (at 1th stage, i.e. full timing)
Experimental Results – Parallelism Comparison

- Capability of multi-threading
  - Higher parallelism translates to higher cpu usage
  - Better resource utilization reflects on higher cpu usage

Performance (OpenTimer)

- Parallel IO by OpenTimer
- Parallel timing by OpenTimer

Performance (iTimerC 2.0)

- Single thread by iTimerC 2.0 (no parallelism)
Conclusion

- Developed a high-performance timing analysis tool
  - Free software and open-source under GPL v3.0
  - Industry format (.v, .spef, .lib, .lef, .def, etc.)
  - Fast, accurate, and robust
  - Multi-threaded and CPPR by default

- Recognition
  - 1st prize in TAU14 contest (full timing with CPPR)
  - 2nd prize in TAU15 contest (incremental timing with CPPR)
  - Golden timer in ICCAD15 CAD contest
  - Golden timer in TAU16 contest

- Acknowledgment
  - Jin, Billy, M.-C., team iTimerC, team iitRACE, and the UIUC CAD group!