OpenTimer v2: A Parallel Incremental Timing Analysis Engine

Tsung-Wei Huang
University of Utah

Chun-Xun Lin and Martin D. F. Wong
University of Illinois at Urbana–Champaign

Editor's notes:
This article introduces a high-quality open-source static timing analysis engine that is capable of parallel incremental timing and that provides an efficient API to facilitate development of complex EDA tools.

—Sherief Reda, Brown University
—Leon Stock, IBM
—Pierre-Emmanuel Gaillardon, University of Utah

Static timing analysis (STA) is a pivotal step in the overall chip design flow. It verifies the expected timing behaviors and prevents chips from malfunctioning after tape-out [1]. Of all timing analysis applications, *incremental timing* is imperative for the success of timing-driven optimization flows, such as placement, routing, logic synthesis, and physical synthesis [2]. Optimization tools often call a timer millions of times in their inner loop to evaluate a transform or an algorithm. The timer must quickly and accurately answer timing queries to ensure slack integrity and timing closure after the circuit experiences one or more changes. The capability of a timer on both speed and accuracy fronts is crucial for reasonable turnaround time and performance.

To this end, we developed *OpenTimer*, a high-performance timing analysis tool in 2015 [4]. OpenTimer is an award-winning tool in the ACM TAU Timing Analysis Contest (2014 through 2016) and has received many recognitions in the CAD community (golden timers in the IEEE/ACM ICCAD CAD Contests and the ACM TAU Contests [2]). OpenTimer is open-source, and we are committed to free sharing of our technical innovation to make EDA a better and open place to engage more talented people contributing to the community [3]. So far, OpenTimer has been used in many industrial and academic projects such as Qflow, VSDflow, CloudV, DARPA IDEA, OpenDesign, LGraph, and Ophidian [5]–[9]. After four years of development, we announced a major release OpenTimer v2 [3]. We rewrote the codebase in modern C++17 and developed a new software architecture to facilitate the parallelization of incremental timing. The overview of the OpenTimer v2 software stack is shown in Figure 1. We summarize our contributions as follows.

- *New parallel task programming model*: We developed a new task-based programming model that enables efficient implementations of parallel decomposition strategies. The new model allows us to go beyond the traditional loop-based parallelization of incremental timing, thereby leading to more asynchrony and faster runtime.
- *New software architecture and API concept*: We developed the core timing routines around three

Digital Object Identifier 10.1109/MDAT.2021.3049177
Date of publication: 5 January 2021; date of current version: 8 April 2021.

concepts, *builder*, *action*, and *accessor*. This separation allows OpenTimer v2 to exploit parallelism from both intra and inter operations, followed by efficient lazy evaluation.

- *New parallel incremental timing framework*: We developed a task-based incremental timing framework that propagates timing naturally with the structure of the timing graph. Our framework can simultaneously perform both graph-based analysis and path-based analysis in parallel while keeping accurate results without breaking complex dependencies between different timing propagation tasks.

Compared with the previous generation, OpenTimer v2 is faster and more scalable in increasing the graph size and the CPU count. The programming interface is also more succinct due to the new API concept. We have made many components modular to make OpenTimer v2 user-friendly and easier for developers to contribute to the codebase. These components include not only the core parallel incremental timing algorithms but also supporting readers/writers for SDC, liberty, and SPEF that can be beneficial for other EDA applications. We believe OpenTimer v2 stands out as a unique system considering the technical innovations and ensemble of software tradeoff and architecture decisions we have made. Recently, OpenTimer was selected as the Best Open-source EDA Tool Award in the 2018 WOSET at ICCAD (one out of 30) [10].

Challenges of incremental timing

Developing an efficient parallel incremental timing engine is a notoriously challenging job, requiring in-depth knowledge of circuit, graph theory, parallel programming, and software engineering. We highlight the three aspects of the challenge we face:

- *Complex task dependencies*: Updating a timing graph takes on load capacitance, parasitics, slew, delay, arrival time, required arrival time, and more. These quantities are interdependent and are not economical to compute. The resulting task dependency in terms of encapsulated function calls is very large and complex.
- *Irregular compute pattern*: Updating a timing graph involves highly diverse computation patterns. We need to capture different forms of timing data whether it is structured in a local block or is flat in the global scope, to implement different delay calculators and pruning heuristics.
- *Unknown API practices*: Our user experience led us to believe that the *API concept* dominates the usability of a timer. When things go incremental, users and developers are often confused by the effect of each operation, such as the per-call complexity, parallelism, and consistency. This can significantly lift up the turnaround time and result in performance pitfall due to misunderstanding of API.

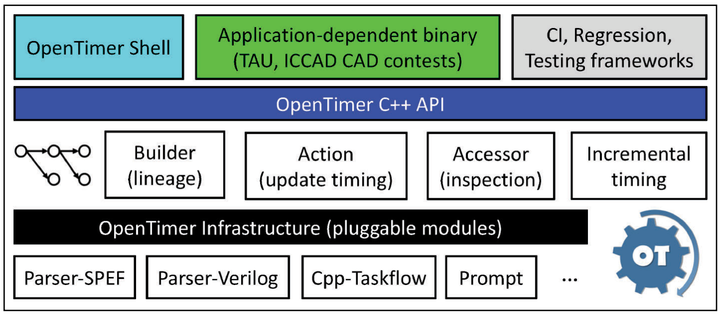

Figure 1. OpenTimer v2 software architecture [3].

The extensibility and scalability to new technology is also an important factor to take into consideration while developing a general incremental timing framework. We are not only interested in technical innovations but also in the modularity of the software to provide a better user experience.

Bottleneck in OpenTimer v1 and existing timers

One of the major differences between v1 and v2 is the parallelization of incremental timing. OpenTimer v1 and existing timers [11]–[13] dealt with incremental timing using *loop-based* parallelism [4]. In a rough view, we levelized the circuit into a topological order and applied the OpenMP "`parallel for`" directive to each node set level by level. This level-based decomposition is advantageous in its simple *pipeline* concept and is by far the most implementation in existing timers, including industrial tools. Figure 2 illustrates this strategy as an example of forward timing propagation. For each node, we update a number of dependent tasks including parasitics (RCP), slew (SLP), delay (DLP), arrival time (ATP), jump points (JMP), and pessimism reduction (RCP) [4]. However, this paradigm suffers from many performance drawbacks. For example, the

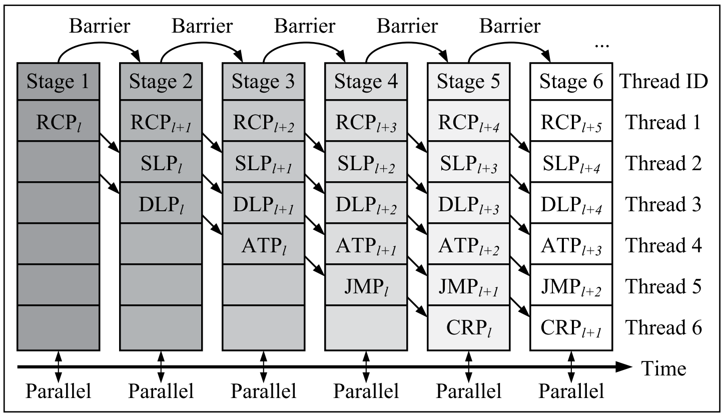

Figure 2. Loop-based parallel timing propagations. Each level applies a `parallel_for` to update timing from the fanin of each node [4].

number of nodes can vary from level to level, resulting in highly unbalanced thread utilization. Also, there is a synchronization barrier between successive levels to impose task dependencies. The overhead can be large for graphs with long data paths. Furthermore, we found it difficult to add to the pipeline other analysis frameworks that require diverse modeling techniques, for example, signal integrity and cross-talk analysis.

Big idea 1: A new parallel task programming model using modern C++

After many years of research, we came to a conclusion that the biggest hurdle to a scalable parallel timer is a suitable *parallel programming model*. In addition to the traditional loop-based approach, the programming model must be capable of *task-based* parallelism. In fact, we have tried multiple options, such as OpenMP 4.5 tasking and Intel Threading Building Blocks (TBB), that are commonly used in EDA applications. We found them unsuitable to our workload for various reasons. For instance, OpenMP 4.5 tasking is *static*. Unfortunately, it is difficult to decide the timing graph at the time of programming. The problem of TBB is the programmability. Users need to understand complex task constructs and templates that are often at low level and hard to maintain. Similar reasons exist in other libraries as well. Therefore, we decided to develop a new parallel task programming model using modern C++ technology. Although the original purpose was for incremental timing, we later generalized it to a standalone open-source project called Taskflow to benefit generic C++ developers [14]. Note that the proposed parallel task programming model is different from that mentioned in [15], which relies on a specialized scheduler to insert tasks dynamically into shared work queues. We focus on *static* modeling that maps the entire timing propagation graph into a task computation graph. When the graph is ready, the scheduler can perform whole-graph optimization and schedule tasks using work-stealing to achieve dynamic load balancing.

Big idea 2: A new API concept and software architecture

With Taskflow in place, we develop a new software architecture in OpenTimer v2 to enable efficient parallel incremental timing. We group each timing operation into one of the three categories, *builder*, *action*, and *accessor*. A timing operation can be either a C++ method in the timer class or command in our shell. Hereafter, the term OpenTimer refers to v2 unless otherwise specified.

Builder: OpenTimer lineage

A builder operation builds up a timing analysis environment, for example, reading cell libraries and a verilog netlist. OpenTimer maintains a lineage graph of builder operations to create a *task execution plan* (TEP). A TEP starts with no dependency and keeps adding tasks to the lineage graph each time users call a builder operation. It records what transformations need to be executed when an action operation is called.

Figure 3 shows an example of OpenTimer lineage. The lineage is made of five builder operations, `read_celllib`, `read_verilog`, `read_sdc`, `enable_cppr`, and `insert_net`. Each time users call a builder operation, the timer adds one or

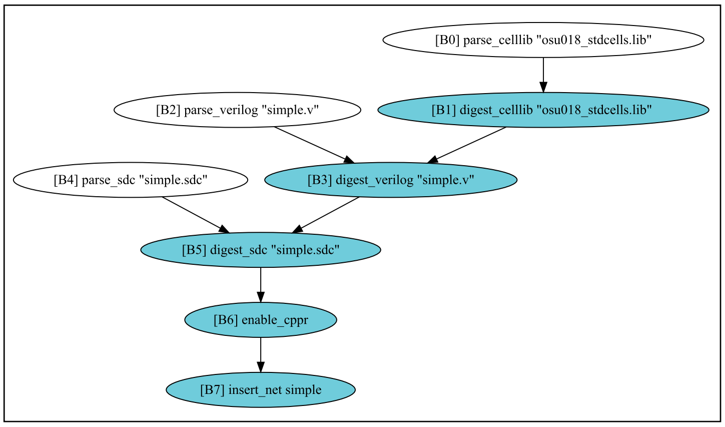
[B0] parse_celllib "osu018_stdcells.lib"
[B1] digest_celllib "osu018_stdcells.lib"
[B2] parse_verilog "simple.v"
[B3] digest_verilog "simple.v"
[B4] parse_sdc "simple.sdc"
[B5] digest_sdc "simple.sdc"
[B6] enable_cppr
[B7] insert_net simple

Figure 3. OpenTimer lineage example of five builder operations (cyan). Three parsing tasks run in parallel.

multiple tasks to the lineage graph. These operations are not evaluated until an action operation is issued. The advantage of this is *fine-grained* task parallelism. An operation is divided into several smaller tasks that can run in parallel with other counterparts. For example, reading an input file can be broken into two subtasks, parsing the file and digesting the data into OpenTimer's in-memory model. It is obvious the parsing part can run in parallel with others as long as it precedes its corresponding digesting task. Maintaining a lineage of builder operations enables us to exploit both intra and interoperation parallelism, followed by efficient lazy evaluation. Another side benefit of the lineage is the engineering change order (ECO) capability. We can easily keep track of the modifiers for state recovery or debugging.

Action: Update timing

A TEP is materialized and evaluated when users request the timer to perform an action operation, for example, reporting the arrival time and the slack value of a pin. Calling an action operation triggers a timing update from the earliest task to the one that produces the result of the action call. Internally, we create a task dependency graph and update timing in parallel, including forward propagation (slew and arrival time) and backward propagation (required arrival time). Figure 4 shows an example of task dependency graph to update a timer. The bottom-most call of every action operation is the method `update_timing`. The method explores a minimum set of nodes in the timing graph as propagation candidates and constructs a task dependency graph to carry out the timing update. Our tasking model can incorporate different types of timing propagation into a task. Unlike the level-based approach in v1, a task can start immediately after all its preceding tasks finish. This largely enhances asynchrony, giving rise to higher CPU utilization, and faster runtime.

Accessor: Inspect OpenTimer

An accessor operation lets users inspect the timer status and dump *static* timing information, for example, dumping the timing graph for visualization purposes or dumping the design statistics. All accessor operations are declared as *constant* methods in the timer class. Calling an accessor method does not alter any internal data structures of a timer.

Big idea 3: Parallel incremental timing analysis algorithms

We discuss, in this section, how OpenTimer performs graph-based analysis and path-based analysis.

Graph-based analysis

At the bottom of every action operation, OpenTimer calls `update_timing` to perform graph-based timing updates. The timer first evaluates the lineage (e.g., Figure 3) and discovers a list of *frontier* pins from which incremental timing should begin after a modification is applied [4]. We then identify the

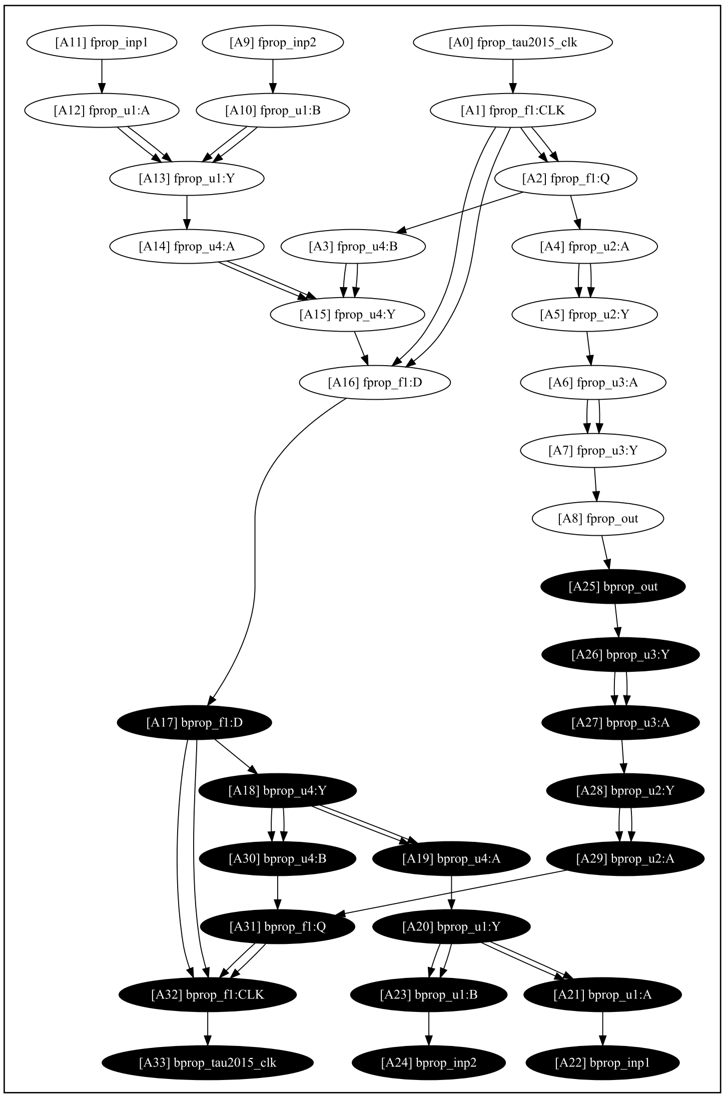

Figure 4. Example task dependency graph to carry out an action operation. The graph consists of forward propagation tasks (white) and backward propagation tasks (black).

Table 1. Accuracy comparison between OpenTimer v1 and v2 on TAU15 contest benchmarks [2].

Circuit	#Gates	#Nets	#Ops	v1	v2
b19	255.3K	255.3K	5641.5K	99.95%	100%
cordic	45.4K	45.4K	1607.6K	98.88%	100%
des_perf	138.9K	139.1K	4326.7K	99.73%	100%
edit_dist	147.6K	150.2K	3368.3K	98.30%	100%
fft	38.2K	39.2K	1751.7K	99.77%	100%
netcard	1496.0K	1497.8K	11594.6K	99.99%	100%
softusb_navre	6.9K	7.0K	427.8K	99.97%	100%
tip_master	37.7K	38.5K	1300.4K	97.04%	100%

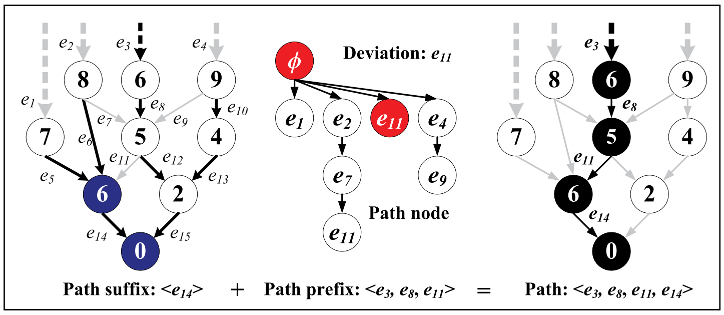
Deviation: e_{11}
Path node
Path suffix: $<e_{14}>$ + Path prefix: $<e_3, e_8, e_{11}>$ = Path: $<e_3, e_8, e_{11}, e_{14}>$

Figure 5. OpenTimer applies implicit path representation based on a suffix tree and a prefix tree data structures per query to perform path-based analysis [16].

propagation candidates (downstream and upstream of frontier pins) and derive a task dependency graph for graph-based timing update (e.g., Figure 4). Executing the task dependency graph autonomously triggers a parallel incremental timing update.

Path-based analysis

We developed our path-based analysis using the path generation algorithm by Huang and Wong [16]. To our best knowledge, this is by far the fastest algorithm in the literature. The algorithm consists of two complementary data structures, *suffix tree* and *prefix tree*. Each path is transformed into an implicit representation that takes constant space and time. The suffix tree represents the shortest path tree rooted at a given endpoint of the design. The prefix tree is a tree order of timing arcs each representing a unique path deviated from a timing arc. Generating the top-k critical paths across all endpoints is extremely efficient under this data structure. It also largely facilitates the parallelization as each pair of suffix tree and prefix tree is independent of each other at different endpoints. An example of the implicit path representation is shown in Figure 5.

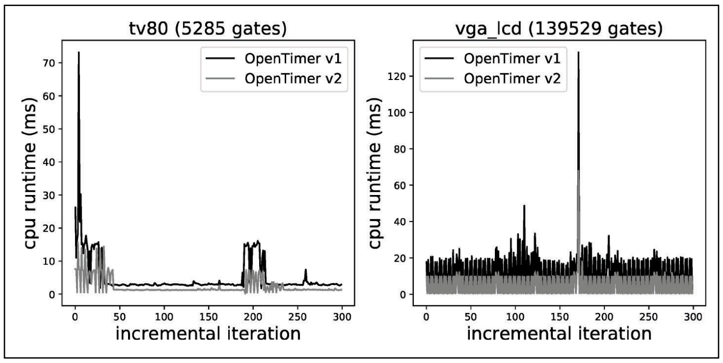

Figure 6. Runtime comparison of incremental timing.

Experimental results

OpenTimer v2 is implemented in C++17 on a 40-core 3.2-GHz 64-bit Linux machine with 64–GB memory. We used G++ 8.0 with `-std=c++17` to compile the source. Experiments are undertaken on the TAU15 contest benchmarks with a golden reference generated by IBM Einstimer under static mode [2]. Table 1 compares the accuracy between OpenTimer v1 and v2 on a set of TAU15 contest benchmarks [2]. These benchmarks are where OpenTimer v1 failed to achieve full accuracy due to an implementation compromise between path generation and parallelization. The new software architecture in v2 lets us manage to resolve these issues and we are able to match the golden results completely. We did not observe too much runtime and memory difference between v1 and v2 on these benchmarks.

The TAU15 contest benchmarks have fewer than ten incremental timing iterations, making it hard to profile the performance. Therefore, we modified two circuits tv80 and vga_lcd, on which both v1 and v2 acquire full accuracy, to incorporate 300 incremental timing iterations. In each iteration, we randomly modify the designs (`e.g., repower_gate`) and call `report_timing` to trigger incremental timing updates. As shown in Figure 6, v2 is consistently faster than v1 (2.14× on tv80 and 2.19× on vga_lcd). About 64% of the speed-up came from replacing the pipeline-based parallelism with the new tasking framework. Figure 7 plots the runtime scalability of v1 and v2 over an increasing number of cores. Regardless of the core count, v2 is always faster than v1. Both saturates at about 8–12 cores.

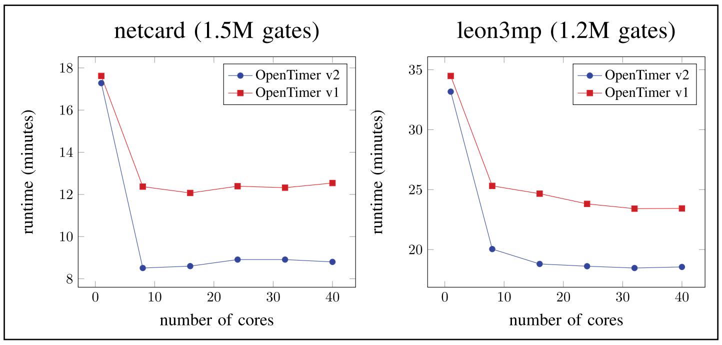

Figure 7. Runtime scalability with increasing number of CPU cores on two large circuits, netcard, and leon3mp.

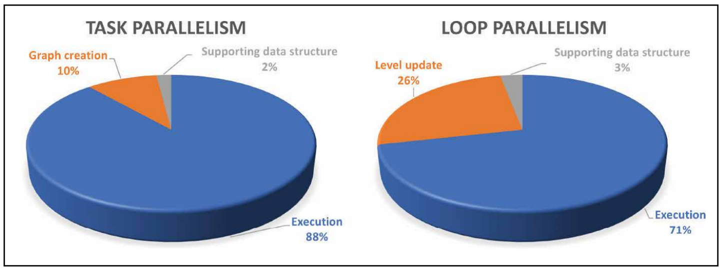

Figure 8. Runtime profiling for task parallelism in OpenTimer v2 and loop parallelism in v1.

The scalability is affected by many factors such as the graph structure and the size of incremental timing. A primary reason that prevents v2 from scaling beyond 12 cores is the data size. Most data for incremental timing are sparse. They do not span across large cones, as full timing, which produces a large amount of data for higher parallelism.

Figure 8 shows the runtime profiling for task-based approach in OpenTimer v2 and loop-based levelization in v1. We measure the time each significant portion of `update_timing` takes in a piechart. Creating a task graph occupies about 10% of the entire runtime and executing the graph takes the majority of 88%. On the other hand, the loop-based approach spent up to 26% on updating the level list and the parallel execution of tasks across all levels takes 71%.

In this article, we presented OpenTimer v2—a new parallel incremental timing analysis tool. We have developed a new parallel task programming model and applied it to design an efficient parallel incremental timing framework. Also, we have introduced a new API concept that defines a clear operation effect on top of our parallelization framework. The source of OpenTimer v2 is available at [3]. ■

Acknowledgments

We would like to thank all OpenTimer users in providing their feedback, suggestions, and requests.

References

[1] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, 2009th ed. New York, NY, USA: Springer, 2009, ISBN-13: 978-0387938196.

[2] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 895–902.

[3] *OpenTimer*. Accessed: 2020. [Online]. Available: https://github.com/OpenTimer/OpenTimer

[4] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 895–902.

[5] *Qflow*. Accessed: 2020. [Online]. Available: http://opencircuitdesign.com/qflow/

[6] *CloudV*. Accessed: 2020. [Online]. Available: https://cloudv.io/

[7] J. Jung et al., "DATC RDF: An academic flow from logic synthesis to detailed routing," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 37:1–37:4.

[8] *LGraph*. Accessed: 2020. [Online]. Available: https://github.com/masc-ucsc/lgraph

[9] *Ophidian*. Accessed: 2020. [Online]. Available: https://gitlab.com/eclufsc/ophidian

[10] *WOSET*. Accessed: 2020. [Online]. Available: https://github.com/woset-workshop/woset-workshop.github.io

[11] *OpenSTA*. Accessed: 2020. [Online]. Available: https://github.com/abk-openroad/OpenSTA

[12] P.-Y. Lee, I. H.-R. Jiang, and T.-C. Chen, "FastPass: Fast timing path search for generalized timing exception handling," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2018, pp. 172–177.

[13] K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2018, pp. 110–117.

[14] T.-W. Huang et al., “Cpp-taskflow: Fast task-based parallel programming using modern C++,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2019, pp. 974–983.

[15] A. Mark Lavin et al., “Decentralized dynamically scheduled parallel static timing analysis,” U.S. Patent 2 012 0311 514 A1, Jul. 8, 2014.

[16] T.-W. Huang and M. D. F. Wong, “UI-timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 11, pp. 1862–1875, Nov. 2016.

Tsung-Wei Huang is currently an Assistant Professor with the Electrical and Computer Engineering (ECE) Department, University of Utah, Salt Lake City, UT. His current research interests focus on timing analysis and parallel processing. Huang has a BS and an MS from the Department of Computer Science, National Cheng Kung University (NCKU), Tainan, Taiwan (2010 and 2011, respectively), and a PhD in ECE from the University of Illinois at Urbana–Champaign (UIUC), Champaign, IL.

Chun-Xun Lin is currently pursuing a PhD with the Department of Electrical and Computer Engineering (ECE), University of Illinois at Urbana–Champaign (UIUC), Champaign, IL. His research interests include VLSI CAD and parallel processing. Lin has a BS in electrical engineering from National Cheng Kung University, Tainan, Taiwan (2009) and an MS in electronics engineering from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan (2011).

Martin D. F. Wong is currently the Dean of the Faculty of Engineering, Chinese University of Hong Kong (CUHK), Hong Kong. Wong has a BS in mathematics from the University of Toronto, Toronto, ON, Canada, and an MS in mathematics and a PhD in computer science (1987) from the University of Illinois at Urbana–Champaign (UIUC), Champaign, IL.

■ Direct questions and comments about this article to Tsung-Wei Huang, Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA; tsung-wei.huang@utah.edu.