OpenTimer v2: A New Parallel Incremental Timing Analysis Engine

Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong, *Fellow, IEEE*

***Abstract*—Since the first release in 2015, OpenTimer v1 has been used in many industrial and academic projects for analyzing the timing of custom designs. After four-year research and developments, we have announced OpenTimer v2—a major release that efficiently supports: 1) a new task-based parallel incremental timing analysis engine to break through the performance bottleneck of existing loop-based methods; 2) a new application programming interface (API) concept to exploit high degrees of parallelisms; and 3) an enhanced support for industry-standard design formats to improve user experience. Compared with OpenTimer v1, we rearchitect v2 with a modern C++ programming language and advanced parallel computing techniques to largely improve the tool performance and usability. For a particular example, OpenTimer v2 achieved up to 5.33× speedup over v1 in incremental timing, and scaled higher with increasing cores. Our contributions include both technical innovations and engineering knowledge that are open and accessible to promote timing research in the community.**

***Index Terms*—Computer-aided analysis, parallel programming.**

I. Introduction

STATIC timing analysis (STA) is a pivotal step in the overall chip design flow. It verifies the expected timing behaviors and prevents chips from malfunction after tapeout [2]. Among timing analysis applications, *incremental timing* is imperative for the success of timing-driven optimization flows, such as placement, routing, logic synthesis, and physical synthesis [3]. Optimization tools often call a timer millions of times in their inner loop to evaluate a design 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. Otherwise, optimization tools may be misguided to a wrong direction ending up with a huge waste of computing resources and timing violations. As a consequence, the capability of a timer on both speed and accuracy fronts is crucial for reasonable turnaround time and performance.

To this end, we have developed *OpenTimer v1*, a high-performance timing analysis tool for very large-scale integration (VLSI) systems in 2015 [1]. OpenTimer is an award-winning tool at ACM TAU Timing Analysis Contests in 2014–2016. It has received many recognitions in the computer-aided design (CAD) community, such as golden timers in 2015 IEEE/ACM ICCAD CAD Contests [4], golden timers of ACM TAU Timing Analysis Contests in 2016–2018 [3], and the Best Open-Source EDA Tool Award (one out of 30) in 2018 WOSET at ICCAD [5]. OpenTimer v1 has been open source and we are committed to free sharing of our technical innovations to make CAD a better and open place in order to engage more talented people to contribute to the community. So far, OpenTimer v1 has been used in many industrial and academic research projects, such as Qflow, VSDflow, CloudV, OpenDesign, LGraph, Ophidian, and more [6]–[12].

After several years of research and developments, we have announced a major release, *OpenTimer v2*, in 2019 DARPA IDEA/POSH Integration Exercise [14]. Since then, we have continued to enhance the capability of the timer. Compared with the previous generation, we rewrote the codebase of OpenTimer v2 from the ground up using modern C++17 and developed a new software architecture to facilitate the design of parallel incremental timing. Performance scalability is never an afterthought in the course of our developments. Our parallel decomposition strategy has delivered new performance scalability and programming productivity that were previously out of reach. Fig. 1 presents the overview of OpenTimer v2's software architecture. We summarize our contributions as follows.

1) *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.
2) *New Software Architecture and API Concept:* We developed the core timing routines around three concepts, *builder*, *action*, and *accessor*. The new application programming interface (API) concept defines a clear and concise logic for each operation of the timer. We leveraged this idea to exploit high degrees of parallelisms both inside and outside a sequence of operations.

Manuscript received January 27, 2020; revised April 7, 2020 and May 15, 2020; accepted June 21, 2020. Date of publication July 7, 2020; date of current version March 19, 2021. This work was supported in part by NSF under Grant CCF-1718883, and in part by the Defense Advanced Research Projects Agency under Grant FA 8650-18-2-7843. The preliminary version of this article has been presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, November 2015 [1]. This article was recommended by Associate Editor W. Yu. *(Corresponding author: Tsung-Wei Huang.)*

Tsung-Wei Huang is with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA (e-mail: twh760812@gmail.com).

Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong are with the Electrical and Computer Engineering Department, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA.

Digital Object Identifier 10.1109/TCAD.2020.3007319

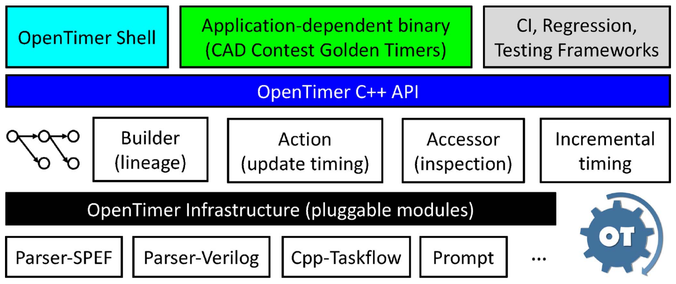


Fig. 1. OpenTimer v2 software architecture [13].

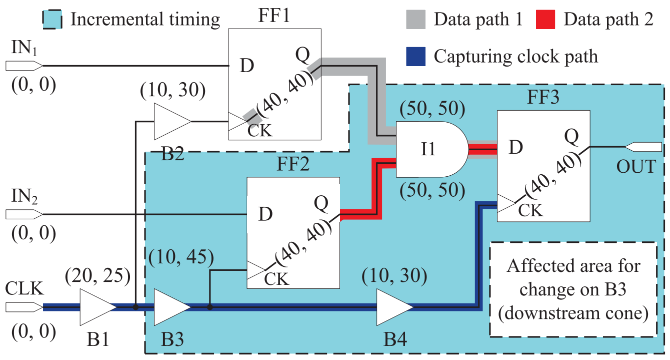


Fig. 2. Incremental timing example [1].

3) *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.
4) *Open Source and System Engineering:* OpenTimer v2 is open source under MIT license [13]. In addition to technical innovations that drive academic values, we have invested a lot in system engineering to make the tool open and accessible to the community. These efforts are crucial for both researchers and practitioners to more easily use OpenTimer v2 and develop derived work on top. We believe this is essential for a CAD project to be impactful.

Compared with the previous generation, OpenTimer v2 is faster, and is more scalable with increasing CPU numbers and problem sizes. The programming interface is more succinct and concise due to the new API concept. We have made many software components of OpenTimer v2 modular and reusable, such that users can quickly integrate OpenTimer v2 into their projects or contribute to our codebase. Other timing-driven CAD applications can benefit from our facility as well. 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. Still, different timers have their pros and cons, and deserve a particular reason to exist [15]–[19]. We would like to position OpenTimer v2 as a modern alternative to advance timing research through parallelism.

II. Incremental Timing

Various stages of the design flow, such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis [3]. 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 Fig. 2, a change on gate B3 has the potential to affect the majority of the circuit (downstream timing). 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 timing (e.g., slew and arrival time) at I1:o, the downstream timing after I1:o is unaffected. Likewise, the effect of modifying gate B3 is up-bounded by its downstream timing. Recomputing the timing outside this region, for instance, gates FF1 and B2, is unnecessary. A timer must act cleverly to quantify a small region effective enough for timing correction after the design is modified.

Fig. 2 is a simplified view of incremental timing. In practice, we incorporate various timing propagation tasks into the incremental timing update. Important tasks include graph-based analysis and path-based analysis, both of which can end up with a large amount of computations. For example, reaching timing closure on an industrial design of 2M gates can easily take several hours or days [3]. It is important to leverage the power of parallel computing on a multicore machine for performance reason.

A. Problem Statement

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

1) *Two liberty (.lib) files* that defines the *early* and *late* characteristics of available cells in a given design, including pin capacitance, delay and slew lookup tables (LUTs), and setup/hold timing guard for sequential elements.
2) *A verilog (.v) file* that defines the netlist and circuit topology in gate level for a given design, including primary input/output ports and connections among gates.
3) *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.
4) *A Synopsys design constraint (.sdc) file* that defines the design operating conditions, including the clock port, clock period, initial timing at primary input ports, and load capacitance at primary output ports.

A practical incremental timer supports the following operations for users to modify the design and query the timing quantities [3].

1) *insert_gate* adds a new gate to the design.
2) *repower_gate* changes the size of an existing gate.
3) *remove_gate* removes a disconnected gate from the design.
4) *insert_net* creates an empty net.
5) *remove_net* removes a net from the design.

6) *read_spef* asserts parasitics on existing nets.
7) *disconnect_pin* disconnects a pin from its net.
8) *connect_pin* connects a pin to a net.
9) *report_at* reports the arrival time at a pin under a rise/fall transition and an early/late split.
10) *report_rat* reports the required arrival time at a pin under a rise/fall transition and an early/late split.
11) *report_slack* reports the slack at a pin under a rise/fall transition and an early/late split.
12) *report_timing* reports the top-k worst critical paths in the design (k by default equals one).

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. Depending on the use case, the timer may need to support another set of operations. Considering the scope of this article, our definition is sufficient enough to represent generic use.

B. Three Big Challenges in Parallel Incremental Timing

Developing an efficient parallel incremental timing engine is never an easy job. It requires deep knowledge about circuit modeling, graph theory, parallel computing, optimization, and dynamic data structures. We highlight the challenge in three aspects as follows.

1) *Complex Task Dependencies:* Updating a timing graph takes dependencies on load capacitance, parasitics, slew, delay, arrival time, required arrival time, and more. These quantities are dependent of each other and are not cheap to compute. The resulting task dependency graph in terms of encapsulated function calls is very large and complex. For example, in a million-gate circuit design, the graph can produce billions of tasks and dependency constraints. It has also been reported that many timing graphs are more connected and complex than that of social media and scientific computing [3].
2) *Irregular Compute Patterns:* Updating a timing graph involves extremely diverse computation patterns and sophisticated memory controls. Data can arrive *densely* and *sparsely* depending on the effect of a design transform. Unfortunately, the timer has little to no knowledge about the intent of an optimization strategy. We need to be prepared for different forms of timing data whether it is structured in a local block or is flat in the global scope. Computing these data typically spawns highly dynamic, iterative, and conditional patterns.
3) *Unknown API Practices:* Our user experience led us to believe that the *API concept* dominates the usability of a timer. According to our user experience, more than 30% efficiency was lost due to wrong use of OpenTimer v1's API. We believe the same hassle happens in other tools as well. 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 pitfalls due to the misuse of an API method.

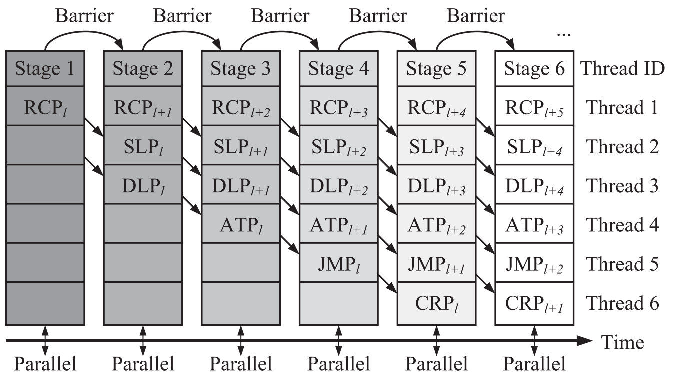


Fig. 3. Loop-based parallel timing propagations. Each level applies a `parallel_for` to update timing from the fanin of each node [1].

The extensibility and scalability to new technology is also an important consideration in unlocking the sustainable growth of our incremental timing engine. We are not only interested in technical innovations but also in the system engineering to keep each software component modular and extensible.

C. Bottleneck in OpenTimer v1 and Existing Timers

Similar to most existing timers, OpenTimer v1 dealt with incremental timing using *loop-based* parallelism [1]. In a rough view, we levelized the circuit into a topological order and maintain levels of nodes using a dynamic data structure, *level list*. Since all the nodes in the same level are independent on each other and can run in parallel, we applied language-specific "`parallel for`" to each list of nodes 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 [19]. Fig. 3 illustrates this strategy on an example of forward timing propagation [1]. For each node, we update a sequence of dependent tasks, including parasitics (RCP), slew (SLP), delay (DLP), arrival time (ATP), jump points (JMP), and common path pessimism reduction (CRP). We encapsulated the task dependency into a parallel pipeline and used one thread to run a particular type of task level by level. Details of each task can be referred to [1]. OpenTimer v1 carried out this strategy using Algorithm 1. It first calls `update_level` to update the level list for incremental timing (line 5). The timing propagation is then performed level by level in a parallel pipeline (lines 8–18 for forward timing propagation and lines 19–25 for backward timing propagation). By the end of each pipeline stage, a barrier is imposed to synchronize all spawned tasks (lines 16 and 23). The level list is reset after the timing propagation completes (line 26). Depending on applications, the timer may add more tasks to the pipeline for parallelism.

The loop-based pipeline strategy is simple and easy to implement using popular parallel programming libraries, such as OpenMP and Intel Threading Building Blocks (TBB) Flow Graph [20], [21]. However, it suffers from many performance drawbacks. For example, the number of nodes can vary from level to level, resulting in highly unbalanced computations and thread utilization. Also, there is a synchronization barrier between successive levels in order to keep task dependencies. The overhead can be large for graph with long data

Algorithm 1: OpenTimer_v1_update_timing()

```
B ← Level list of the timer;
if $B.num\_pins = 0$ then
    return;
end
update_level($B$);
$l_{min} \leftarrow B.min\_nonempty\_level$;
$l_{max} \leftarrow B.max\_nonempty\_level$;
# Parallel_Region {
# Master_Thread_do for $l = l_{min}$ to $l_{max} + 4$ do
    # spawn_task propagate_rc($l$);
    # spawn_task propagate_slew($l - 1$);
    # spawn_task propagate_delay($l - 1$);
    # spawn_task propagate_arrivel_time($l - 2$);
    # spawn_task propagate_jump_point($l - 3$);
    # spawn_task propagate_cppr_credit($l - 4$);
    # synchronize_tasks;
end
};
# Parallel Region {
# Master_Thread_do for $l = l_{max}$ to $B.min\_non\_empty\_level$ do
    # spawn_task propagate_fanin($l$);
    # spawn_task propagate_required_arrival_time($l$);
    # synchronize_tasks;
end
};
remove all pins from the level list $B$;
```

paths. Furthermore, it is difficult to extend the pipeline to efficiently include other types of compute-intensive tasks, such as advanced delay modeling, signal integrity, and cross-talk analysis. These tasks often expose dependency constraints across multiple layers of the timing graph that do not fit in a single level [2]. Neither can path-specific update be included to the pipeline without significant rewrite of the core data structure.

III. OpenTimer v2: New Breakthrough

With years of research, we decided to rearchitect the core of OpenTimer v1 from the ground up to a comprehensively parallel target based on three big ideas: 1) a new parallel task programming model; 2) a new API concept and software architecture; and 3) a new task-based parallel incremental timing algorithm. We discuss these ideas as follows.

A. New Parallel Task Programming Model

Over the past years, we have studied many parallel processing options with our industrial partners, and 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 libraries, such as OpenMP 4.5 tasking and TBB. We found them not easy to suit our workload for various reasons. For example, OpenMP task dependency clause relies on *static*

```
tf::Taskflow taskflow;
tf::Executor executor;

auto [A, B, C, D] = tf.silent_emplace(
  [] () { std::cout << "Task A\n"; },
  [] () { std::cout << "Task B\n"; },
  [] () { std::cout << "Task C\n"; },
  [] () { std::cout << "Task D\n"; }
);

A.precede(B, C);   // A runs before B and C
D.succeed(B, C);   // D runs after  B and C

executor.run(taskflow).wait();
```

Listing 1. Example of OpenTimer v2's parallel task programming engine, Cpp-Taskflow [22].

task annotations. Developers must explicitly specify a valid order of tasks consistent with the sequential execution. This restriction makes it very difficult to handle dynamic flows, where the graph structure is unknown at programming time. On the other hand, TBB is disadvantageous from an ease-of-programming standpoint. Its task graph description language is very complex and often results in large source lines of code (LOC) that are hard to read and debug. These issues combined to make it difficult to go beyond the loop-based approach and prevent computations from flowing naturally along the timing graph. Therefore, we decided to develop a new parallel task programming model using modern C++ technology. While the original purpose was for OpenTimer v2, we have generalized it to an open-source general-purpose parallel task programming library, Cpp-Taskflow, for generic C++ developers [22], [23].

Listing 1 presents an example of a Cpp-Taskflow program. The code is *self-explanatory*. The program creates a task dependency graph of four tasks, A, B, C, and D, where task A runs before task B and task C, and task D runs after task B and task C. Each task is described as a *callable object*, which can be either a lambda, a functor, a binding expression, or an operator. Cpp-Taskflow provides an abstraction over difficult concurrency controls, such as threading and scheduling. Users describe an application in terms of *tasks* rather than threads. They do not need to manage threads or locks, and can focus on high-level developments. In fact, delegating task scheduling to Cpp-Taskflow lets the core of OpenTimer v2 remain small and flexible. Individual libraries can evolve at a faster pace for each problem domain. Due to the space limit, we refer the readers to our official repository to learn more about Cpp-Taskflow [23]. The project has gained much attention in gaming, animation, and parallel programming communities. In 2019, ACM Multimedia Conference awarded Cpp-Taskflow the second prize in open-source software competition. A key difference between Cpp-Taskflow and existing tools is our experience of real-world VLSI timing applications. Cpp-Taskflow is the backbone technology of OpenTimer v2.

B. New API Concept and Software Architecture

With Cpp-Taskflow in place, we develop a new software architecture in OpenTimer v2 to enable efficient parallel decomposition strategies of incremental timing. We do not

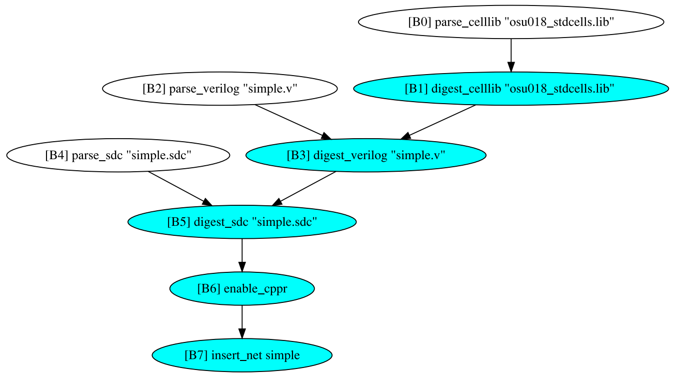

[B0] parse_celllib "osu018_stdcells.lib"
[B1] digest_celllib "osu018_stdcells.lib"
[B2] parse_verilog "simple.v"
[B3] digest_verilog "simple.v"
[B4] parse_sdc "simple.sdc"
[B5] digest_sdc "simple.sdc"
[B6] enable_cppr
[B7] insert_net simple

Fig. 4. Lineage graph example of five builder operations (marked in cyan). Three parsing tasks (marked in white) are independent on each other and run in parallel.

Algorithm 2: read_celllib(*path*)

```
Input: a cell library file locator path
1 G ← get_lineage_graph();
2 c ← make_celllib_parser(path);
3 parser ← G.add_task([c] () { c.parse(); });
4 reader ← G.add_task([c] () { read_celllib_impl(c); });
5 parser.precede(reader);
6 append_task_to_lineage_graph(G, reader);
```

Algorithm 3: remove_net(*n*)

```
Input: net name n to remove from the timer
1 G ← get_lineage_graph();
2 task ← G.add_task([n] () { remove_net_impl(n); });
3 append_task_to_lineage_graph(G, task);
```

only address the algorithm but also the entire software stack from tool commands to library APIs. We group each timing operation to one of the three categories, *builder*, *action*, and *accessor*. A timing operation can be either a C++ method in the timer class or a command in the shell.

1) Builder (Lineage Graph): A builder operation builds up a timing analysis environment, for example, reading cell libraries and verilog netlists. OpenTimer v2 maintains a lineage graph of builder operations to create a task execution plan (TEP). A TEP starts with no dependency and appends tasks to the lineage graph when users call a builder operation. It records what transformations need to be executed when an action operation is issued. A key advantage of lineage graph is *fine-grained parallelism* and *lazy evaluation*. We can decompose a large builder operation into small blocks and pipeline these tasks with other operations in the lineage graph. Results of builder operations are lazily evaluated when an action is required. The lazy evaluation avoids unnecessary synchronization between subsequent operations.

Fig. 4 shows an example of the lineage graph in OpenTimer v2. The lineage graph 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 appends one or multiple tasks to the end of the lineage (marked in cyan). These operations are not evaluated until an action operation is issued. This example enables fine-grained task parallelism: the process of reading a design file is broken into two subtasks, parsing the file and digesting the data into the timer's in-memory model. Obviously, the parsing part can run in parallel with other independent tasks as long as it precedes its subsequent reader task. We leverage the lineage graph to exploit both intraoperation and interoperation parallelism, and evaluate their results lazily. Another side benefit of the lineage graph is the ease of debugging. We can easily keep track of the timer state by dumping the lineage graph. With advanced setting, we can perform state recovery without time-consuming serialization to an external database.

Algorithms 2 and 3 give an example of two builder operations, `read_celllib` and `remove_net`. We decompose the operation `read_celllib` into two tasks, `parser` and `reader`. Task `parser` is an independent application that scans the syntax of the input cell library and builds a data structure, for instance, parsing tree, abstract syntax tree, and other hierarchical structure. Task `reader` digests the data structure and transforms useful information to the timer's in-memory model. Apparently, `reader` must not run before `parser` completes, and `parser` can run in parallel with other independent tasks. The second operation `remove_net` is at the finest level of our decomposition. We create a single task to process the net modifier. The task body invokes an internal implementation `remove_net_impl` to alter the net data structure of the timer. Encapsulation of other builder operations is similar to these two examples.

2) 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 or the slack value at a pin under a given rise/fall transition and a min/max timing split. 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 both forward propagation (slew, delay, and arrival time) and backward propagation (required arrival time). Fig. 5 shows a task dependency graph created at an action operation to update timing. We can observe that the task graph flows computations naturally with the timing graph. There is no explicit synchronization at each node to coordinate with the loop-based pipeline strategy, but automatic scheduling and distribution of tasks across CPU cores whenever dependencies are met. To enforce forward propagation to precede backward propagation, we only need to add dependencies on the last layer of forward propagation tasks, often at endpoints (zero outgoing edges). In this example, we have `fprop_out` (task at a primary output) and `fprop_f1:D` (task at the data pin of a flip-flop). Notice that the forward propagation tasks are always a subset of backward propagation tasks, because the affected delay may also change the required arrival time.

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

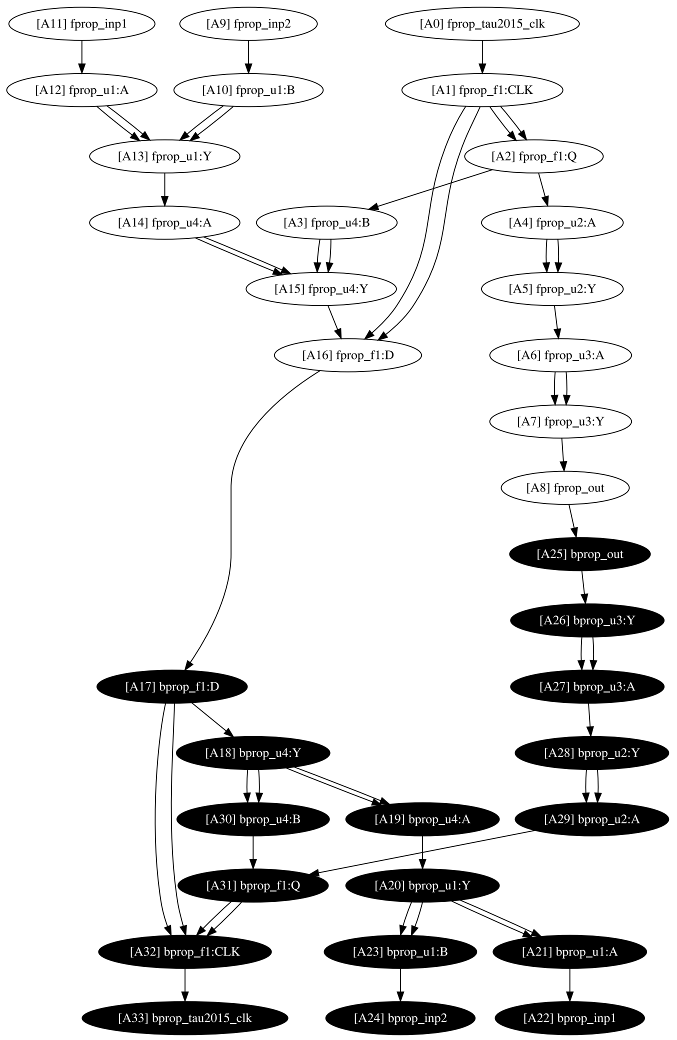


Fig. 5. Task dependency graph to carry out an action operation. The graph consists of forward propagation tasks (white) and backward propagation tasks (black).

constructs a task dependency graph to carry out the timing update. We shall discuss in the later section how OpenTimer v2 maintains the list of propagation candidates for incremental timing. 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 the asynchrony of task execution and improves the CPU utilization for faster runtime.

3) Accessor (Inspect the Timer): An accessor operation lets users inspect the timer status and dump *static* timing information, for example, dumping the timing graph for visualization purpose 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. Fig. 6 shows an example of using an accessor operation to dump a timing graph. The primary use of accessor operations is inspection. For instance, users can observe the difference before and after the timer evaluates the lineage, or examine the slack value of a pin before and after calling `update_timing`.

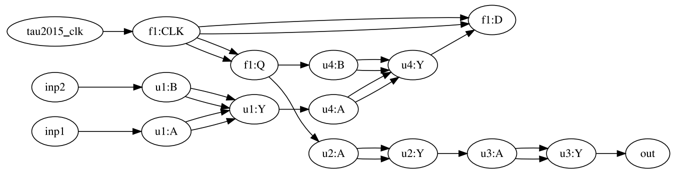


Fig. 6. Inspect the timing graph through accessor `dump_graph`.

TABLE I
OpenTimer v2 Commands and API Concept

API	Type	Function
read_celllib	builder	read a Liberty format cell library
read_verilog	builder	read a gate-level verilog netlist
read_spef	builder	read net parasitics in SPEF format
read_sdc	builder	read a timing constraint file
report_timing	action	report the critical path
report_slack	action	report the slack at a pin
report_at	action	report the arrival time at a pin
report_rat	action	report the required arrival time at a pin
report_slew	action	report the slew at a pin
dump_graph	accessor	dump the timing graph in DOT
dump_lineage	accessor	dump the lineage graph in DOT
dump_timer	accessor	dump the design statistics

4) User-Level Interface Design: It is clear that the proposed API concept enables the timer to exploit high degrees of parallelism both inside and outside a sequence of operations. Once users understand the software architecture of OpenTimer v2, the complexity and effect of each operation become clear and concrete. Table I lists a couple of API examples and shell commands available in OpenTimer v2. Each builder method takes $O(1)$ time, while the time complexity of an action or an accessor operation is algorithm dependent. Another significant benefit from this new software architecture is *thread safety*. On the basis of TEP, all operations are *linearizable* in the order of their calls. The API exposed to users is *thread-safe*.

C. New Task-Based Parallel Incremental Timing Algorithm

We describe how OpenTimer v2 performs incremental timing. In particular, we discuss algorithms for design modifiers, graph-based analysis, and path-based analysis.

1) Design Modifiers: The objective of handling each design modifier is to identify a set of "*frontier pins*" from which the incremental timing update originates. Starting from each frontier pin, we construct a downstream cone for forward propagation. This downstream cone is the maximum region affected by the design modifier. Then, starting from each pin in the downstream cone, we construct an upstream cone for backward propagation. This upstream cone is the maximum region affected by the design modifier. We consider the design modifiers at the *gate level*, *net level*, and *pin level*. Each design modifier is a builder operation that takes constant time in the lineage graph. OpenTimer v2 creates a task for each design modifier on top of an internal implementation that alters the timer state. Function `remove_net_impl` in Algorithm 3 is such an example. Each design modifier has an internal implementation to invoke by the timer when an action call materializes the lineage graph. As a consequence, we discuss the algorithm of each internal implementation with a suffix

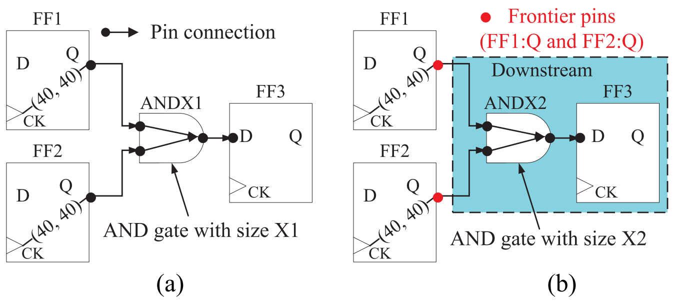


Fig. 7. Repowering a two-input AND gate with another size introduces two frontier pins at its two inputs. (a) Circuit fragment. (b) Repower gate (X1→X2).

Algorithm 4: repower_gate_impl(g, c)

```
Input: an existing gate g, a new cell c
1 remap gate g to new cell c;
2 B ← get_frontier_list();
3 foreach p ∈ g.input_pins do
4     foreach 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
```

impl (Algorithms 4–7). Constructing a builder task for each design modifier is similar to Algorithms 2 and 3.

Gate-level design modifiers include: 1) `insert_gate`; 2) `remove_gate`; and 3) `repower_gate`. Recall in our problem statement, the operation `insert_gate` creates a new *unconnected* 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`. An example of the operation `repower_gate` is shown in Fig. 7. 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 to the fanin pins of the gate's inputs. In this example, the incremental timing propagation is captured by two frontier pins FF1:Q and FF2:Q. Using this fact, our solution to `repower_gate` is presented in Algorithm 4. Algorithm 4 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 frontier list (lines 3–9) for incremental timing update.

Net-level design modifiers include: 1) `insert_net`; 2) `remove_net`; and 3) `read_spef`. Similar to gate-level modifications, the operation `insert_net` creates an empty unconnected net for the design and the operation `remove_net` deletes an empty disconnected net from the design. Due to the isolation, both operations have no impact

Algorithm 5: read_spef_impl(O)

```
Input: data structure O of a parsed parasitics
1 B ← get_frontier_list();
2 foreach net n ∈ O do
3     update the parasitics of net n through O;
4     n.is_rc_up_to_date ← false;
5     $p_r$ ← n.rc_network_root_pin;
6     B.insert($p_r$);
7 end
```

on the current timing profile. The net-level design modifier `read_spef` is the only operation that could affect the timing due to new parasitics. Our solution to `read_spef_impl` is presented in Algorithm 5. Keep in mind the task of `read_spef` is created in the same way as Algorithm 2. Algorithm 5 takes a data structure of a parsed parasitics O from an SPEF generator, provided by the optimization tools, as an input argument. Optimization tools provide SPEF files and iterate each net in O and attaches the net with new parasitics (lines 2 and 3). Whenever a net changes parasitics, the root of its RC network is inserted to the frontier list for incremental timing (lines 4 and 6). In large designs, SPEF files can be very large. Decomposing `read_spef` into two parts, parsing and reading, and pipeline the parsing task with other independent tasks in the lineage graph can gain significant performance benefit.

Pin-level design modifiers include: 1) `disconnect_pin` and 2) `connect_pin`. Pin-level design modifiers are the most essential operations because they directly alter the connectivity in the design. The operation `disconnect_pin` disconnects a pin from its net and the operation `connect_pin` connects a pin to a given net. Both operations modify the structure of the design and have a direct effect on timing. An example of `disconnect_pin` and `connect_pin` is given in Fig. 8. It can be seen from Fig. 8(a) disconnecting the pin I1:o from its net cuts off the connection between 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 FF3:D. Therefore, disconnecting a pin introduces two frontier pins that are the two ends 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 Fig. 8(b), connecting pin I1:o to net n1 produces a new connection from pin I1:o to pin FF3:D. This change has an 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 6 and 7 present our solutions to pin-level modifiers. Notice that a pin is considered either a root of the RC network where we need to remove or insert all possible connections, including any hierarchical connections that cover such a change (lines 4–8 in Algorithm 6 and lines 2–7 in Algorithm 7), or the terminal of the RC network in which case we deal with the only one connection (line 10 in Algorithm 6 and lines 9–11 in Algorithm 7).

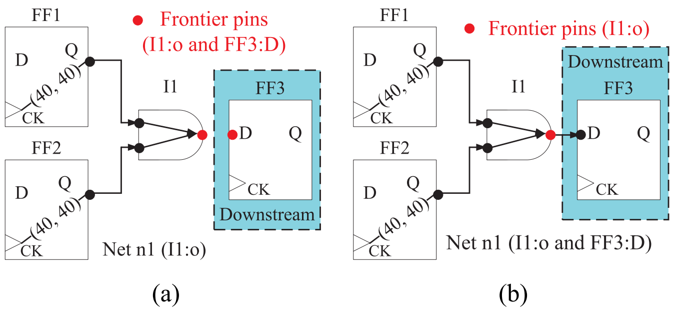


Fig. 8. Design modification by disconnecting/connecting a pin from/to a net. (a) Disconnect pin FF3:D from net n1. (b) Connect pin FF3:D to net n1.

Algorithm 6: disconnect_pin_impl(p)

```
Input: an existing pin p
1  n ← p.net ;
2  p_r ← n.rc_network_root_pin ;
3  B ← get_frontier_list();
4  if p = p_r then
5      foreach 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 hierarchical connections at p;
14 disconnect the pin p from the net n;
```

Algorithm 7: connect_pin_impl(p, n)

```
Input: an existing pin p and an existing net n
1  B ← get_frontier_list();
2  if p.is_rc_network_root_pin = true then
3      foreach p' ∈ n.pinlist do
4          establish the connection from p to p';
5          disconnect all hierarchical connections to p';
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 hierarchical connections to p;
14 connect the pin p to the net n;
```

2) Graph-Based Analysis: At the bottom of every action operation, OpenTimer v2 calls `update_timing` to perform graph-based timing update *starting from frontier pins*. As shown in Algorithm 8, the timer first evaluates the lineage graph if it exists (lines 1–6). After the lineage graph is materialized, the timer enters a state ready for timing update. If users request a full timing update, we insert all pins to the frontier list. We then identify the propagation candidates by expanding the downstream and upstream cones of each frontier pin (lines 10–18), and derive a task dependency graph out of these candidates for graph-based timing update (line 19). Submitting the task dependency graph to a taskflow executor autonomously triggers parallel incremental timing update and returns with up-to-date timing values (line 20).

Algorithm 8: update_timing

```
1  G ← get_lineage_graph();
2  if G.empty() == true then
3      return;
4  end
5  executor.run(G).wait();
6  G.reset();
7  if full_timing == true then
8      insert_all_pins_to_frontiers();
9  end
10 B ← get_frontier_list();
11 C⁺ ← ϕ;
12 C⁻ ← ϕ;
13 foreach p ∈ B do
14     C⁺∪ downstream_cone(p);
15 end
16 foreach p ∈ C⁺ do
17     C⁻∪ upstream_cone(p);
18 end
19 build_propagation_tasks(G, C⁺, C⁻);
20 executor.run(G).wait;
```

Algorithm 9 presents the details of how OpenTimer v2 constructs a task dependency graph to update timing. Each pin is associated with two tasks: 1) forward propagation task `ftask` and 2) backward propagation task `btask`. First, we construct the forward part of the task dependency graph from the forward propagation candidates C^+ (lines 1–8). A typical forward propagation task (line 2) updates the parasitics, slew, delay, arrival time, and timing constraints (setup, hold, and clock) of the associated pin (see Algorithm 10). Notice that the constraint arc is part of the dependency to force clock path to update before the constraint values. Next, we construct the backward part of the task dependency graph from the backward propagation candidates (lines 9–16). A backward propagation task (line 10) updates the quantities that are only available after forward propagation, for instance, required arrival time (see Algorithm 11). Finally, we add dependencies to the last layer of pins to force forward propagation tasks to run before backward propagation tasks (lines 17–21). Our task-based approach is flexible in including other tasks to the graph-based analysis as long as dependency is properly attached. For example, we can create tasks of the path-specific update, pessimism removal credits, and advanced delay modeling and include them to Algorithms 10 and 11. Each task can implement its own pruning strategy as well, such as early termination at pins of unchanged delays.

Using Algorithm 8 as the infrastructure, the value-based timing query, for example, `report_at` (reporting the arrival

```
Algorithm 9: build_prop_tasks($G$, $C^+$, $C^-$)
  Input: a taskflow graph $G$
  Input: forward propagation candidates $C^+$
  Input: backward propagation candidates $C^-$
1 foreach $f \in C^+$ do
2     $f.ftask \leftarrow G$.add_task([](){ fprop($f$); });
3     foreach $p^- \in f.fanin()$ do
4         if $p^- \in C^+$ then
5             $p.ftask$.precede($f.ftask$);
6         end
7     end
8 end
9 foreach $b \in C^-$ do
10    $b.btask \leftarrow G$.add_task([](){ bprop($b$); });
11    foreach $p^- \in b.fanin()$ do
12        if $p^- \in C^-$ then
13            $b.btask$.precede($p.btask$);
14        end
15    end
16 end
17 foreach $b \in C^-$ do
18    if $b.btask.num\_dependents() == 0$ then
19        $b.ftask$.precede($b.btask$);
20    end
21 end
```

```
Algorithm 10: fprop($p$)
  Input: a pin $p$
1 update_parasitics($p.net$);
2 update_slew($p$);
3 update_delay($p$);
4 update_arrival_time($p$);
5 update_timing_constraints($p$);
```

```
Algorithm 11: bprop($p$)
  Input: a pin $p$
1 update_required_arrival_time($p$);
```

```
Algorithm 12: report_at($p$, $s$, $t$)
  Input: an existing pin $p$
  Input: an early/late (i.e., min/max) timing split $s$
  Input: a rise/fall transition $t$
  Output: arrival time at $p$ under $s$ and $t$
1 update_timing();
2 return $p.arrival\_time(s, t)$;
```

time), can be implemented as Algorithm 12. Other action operations can be implemented in the same manner.

3) Path-Based Analysis: Path-based analysis is an important step in the timing analysis flow. Among various techniques, a core routine, often referred to as `report_timing`, requests a set of critical paths and generates timing reports. We developed the critical path generation algorithm based on our prior work in [24]. To the best of our knowledge, this is by far the fastest algorithm in the literature. The algorithm consists of two complementary data structures: 1) *suffix tree* and 2) *prefix tree*. Each path is transformed to 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 with respect to a suffix tree. Each edge in the prefix tree represents a *unique* path deviated from the suffix tree. 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 Fig. 9. Due to the space limit, we refer readers to [24] for a detailed discussion of the algorithm.

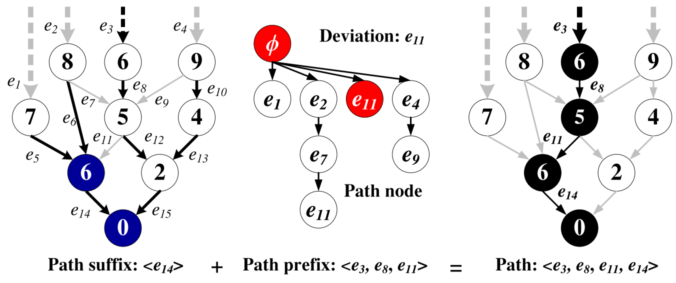


Fig. 9. OpenTimer v2 applies implicit path representation based on a suffix tree and a prefix tree data structures per query to perform path-based analysis [24].

```
Algorithm 13: report_timing
  Input: Endpoints, path count $k$
  Output: top-$k$ critical paths across all endpoints
1 update_timing();
2 $E \leftarrow$ update_endpoints($k$);
3 PathHeap $heap$;
4 $taskflow$.transform_reduce($E$.begin(), $E$.end(), $heap$,
5     merge_two_heaps($heap1$, $heap2$, $k$),
6     merge_heap_endpoint($heap$, $endpoint$, $k$),
7     endpoint_to_heap($endpoint$, $k$)
8 );
9 $executor$.run($taskflow$).wait()
10 return $heap$.extract_paths();
```

Algorithm 13 presents a skeleton of `report_timing` that outputs the top-k critical paths across all endpoints. The data structure `PathHeap` encapsulates both suffix tree and prefix tree for generating critical paths at one endpoint [24]. We developed a method `transform_reduce` in Cpp-Taskflow to perform the parallel reduction. The method transforms each endpoint to a path heap of k critical paths and then iteratively merges two heaps into a new heap of size k until one left. In general, the algorithm can handle more arguments such as outputting the top-k critical paths only for the min (hold) or max (setup) session. Our parallelization framework is very general and is extensible to many path-specific updates. For example,

TABLE II
SOFTWARE COST OF OPENTIMER V1 AND V2

Tool	Parallelism	LOC	Effort	Sched	Dev	Cost
v1	OpenMP	9,123	2.04	0.70	2.90	$275,287
v2	Cpp-Taskflow	4,482	0.97	0.53	1.83	$130,523

TABLE III
ACCURACY COMPARISON BETWEEN OPENTIMER V1 AND V2 ON TAU15 CONTEST BENCHMARKS [3]

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%

we incorporate common path pessimism removal (CPPR) into path generation tasks.

IV. EXPERIMENTAL RESULTS

OpenTimer v2 is implemented in C++17 on a 40-core 3.2-GHz 64-b Linux machine with 128-GB memory. We used G++ 8.0 with `-std=c++17` to compile the source. Our compiler supports POSIX threads (pthread) and OpenMP through the ubiquitous gcc tool chain. Experiments are undertaken on the TAU15 contest benchmarks with a golden reference generated by IBM Einstimer under static mode [3]. We used the same benchmarks to evaluate the correlation between OpenTimer v2 and a commercial tool OpenSTA by Parallax Software Inc. [25]. To precisely measure the effect of parallelism, we lock each thread to one CPU core by using both: 1) library-specific API to restrict the number of spawned threads and 2) OS-level utilities (`taskset`) to affine the running process to the same number of CPU cores. All data are the average of ten runs.

A. Software Cost

We discuss the software cost between OpenTimer v1 and OpenTimer v2. OpenTimer v1 adopted OpenMP 4.5 tasking clauses to implement the pipeline-based incremental timing [20]. OpenTimer v2 leveraged Cpp-Taskflow to implement the task-based incremental timing [22]. The new software architecture and API concept allow the core of OpenTimer v2 to remain compact and modular. Developers can quickly use our timer in a standalone analysis environment or derive a timing-driven application. Table II compares the software costs between OpenTimer v1 and v2. *LOC* denotes the source lines of code, *Effort* estimates the development effort, *Sched* shows the largest estimated schedule of a component, *Dev* estimates the developer count, and *Cost* estimates the cost to develop. All quantities are reported by the popular Linux tool `SLOCCount` under the constructive cost model (COCOMO) organic mode [26]. The organic mode emulates a small research team of 2–3 people.

OpenTimer v2 reduces almost 50% of the coding complexity from v1 (9K versus 4K LOC) even with enhanced capability and new features. We attribute this to our new parallel task-based incremental timing framework and the software architecture of OpenTimer v2. A large amount of parallel code that was used to maintain the pipeline data structure in OpenTimer v1 is now replaced by only a few lines of Cpp-Taskflow code. Pushing parallel task processing to Cpp-Taskflow keeps the core of OpenTimer v2 small and efficient. It also minimizes the rate of change required by the timer, which makes it easier to keep OpenTimer v2 scalable and robust. Our measurement may be subjective, but it highlights the potential development effort involved in building derived work on top of OpenTimer v2.

B. Performance

We study the performance of OpenTimer v2 and its improvement over OpenTimer v1 on four fronts: 1) accuracy on TAU15 benchmarks; 2) incremental timing runtime; 3) scalability with CPU cores; and 4) overhead of task parallelism.

1) Accuracy: Table III compares the accuracy between OpenTimer v1 and v2 on a set of TAU15 contest benchmarks [3]. These benchmarks are where OpenTimer v1 failed to achieve full accuracy due to a compromise we made for speed. In Algorithm 1, updating the level list (line 5) is in fact very time consuming. We need to run a topological traversal over the downstream cone and upstream cone of frontier pins to restore a correct level order. About 25% runtime is taken by this process. Therefore, OpenTimer v1 adopted a simple heuristic to prune to level update at a threshold, at the cost of accuracy loss. The tradeoff is likely inevitable if we stick with the loop-based pipeline decomposition strategy. However, OpenTimer v2 modeled the timing propagation in a task dependency graph that is directly associated with the timing graph. We need no extra effort to maintain additional level dependencies imposed by the pipeline data structure. Neither is any accuracy–speed tradeoff. The new accuracy values are able to match the golden reference completely.

2) Incremental Timing: The TAU15 contest benchmarks have fewer than ten incremental timing iterations, making it hard to study the performance. Therefore, we modified these circuits to incorporate 1000 incremental timing iterations. Each incremental timing iteration consists of a sequence of builder operations and one action operation at the end. The action operation triggers incremental timing update from the earliest builder to the first action call that queries the timing result. For example, a valid incremental timing iteration may be one gate-sizing operation `repower_gate` followed by a value-based timing query `report_at`. In practice, an optimization engine may invoke millions of iterations in a timing-driven loop.

Fig. 10 compares the incremental timing performance between OpenTimer v1 and v2 on two combinational circuits (c6288 and c7552) and six sequential circuits (ac97_ctrl, aes_core, wb_dma, tv80, des_perf, and vga_lcd). We draw the comparison in two lines, *per-iteration* runtime and *cumulative* runtime. Both quantities are important for optimization

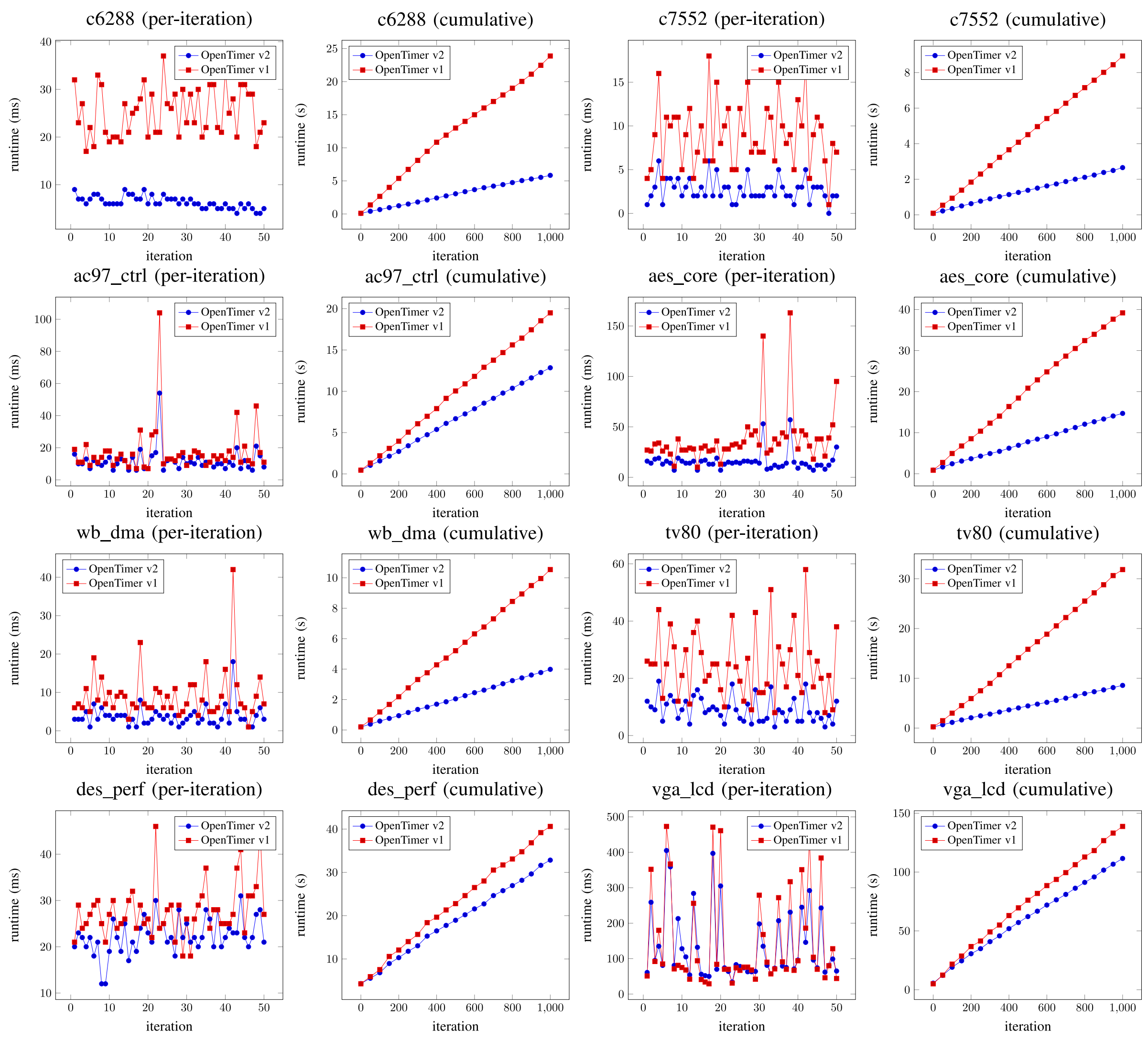


Fig. 10. Comparison of incremental timing performance between OpenTimer v1 and v2 on combinational circuits c6288 and c7552, and sequential circuits ac97_ctrl, aes_core, wb_dma, tv80, des_perf, and vga_lcd.

tools to understand the efficiency of using a timer in a local scope and a global span. As a whole, OpenTimer v2 outperforms v1 in almost all scenarios. In combinational circuits c6288 and c7552, v2 is consistently faster than v1. The maximum speedup in a single iteration is 6.83× and 4.5× at c6288 and c7552, and the respective average speedup is 3.71× and 2.75×. For sequential circuits, the runtime difference between v1 and v2 is also clear. The largest margin we observed is at tv80, where v2 reached the goal in 8.57 s and v1 required 31.84 s (3.47× slower). Unlike other circuits, tv80 has many long data paths. This results in a very long pipeline in v1's parallel decomposition strategy. The cost to incrementally update the level list of long length ends up very high. On the other hand, vga_lcd is structured in an opposite way. It has many flip-flops, and most data paths are short in the pipeline. Shorter pipeline often produces more nodes in a single layer from the timing graph perspective. Therefore, the performance margin between v1 and v2 is close in this scenario. Still, v2 is faster. The maximum and average speedup values are 1.86× and 1.11×, respectively. Another interesting finding in vga_lcd is the first iteration, where full timing took place to update the entire timing graph. v2 is a bit slower due to the fact we discussed above. However, the difference is negligible.

3) Scalability: We measure the runtime scalability with increasing number of cores on four large circuits, vga_lcd (139.5K gates), leon2 (1.6M gates), leon3mp (1.2M gates), and netcard (1.5M gates). We randomly generated 1000 incremental timing iterations for each circuit and measured the elapsed runtime of `update_timing` to complete all iterations at 1, 8, 16, 24, 32, and 40 cores. Fig. 11 draws the result. We observed OpenTimer v2 scales higher than OpenTimer v1 with increasing cores. Both saturate at about 8–12 cores. 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, that produces a large amount of data for higher parallelism. Another interesting aspect is the speedup number versus core count. Using 8 cores, we speed up vga_lcd by about 3.6×, which is away from the idea factor of

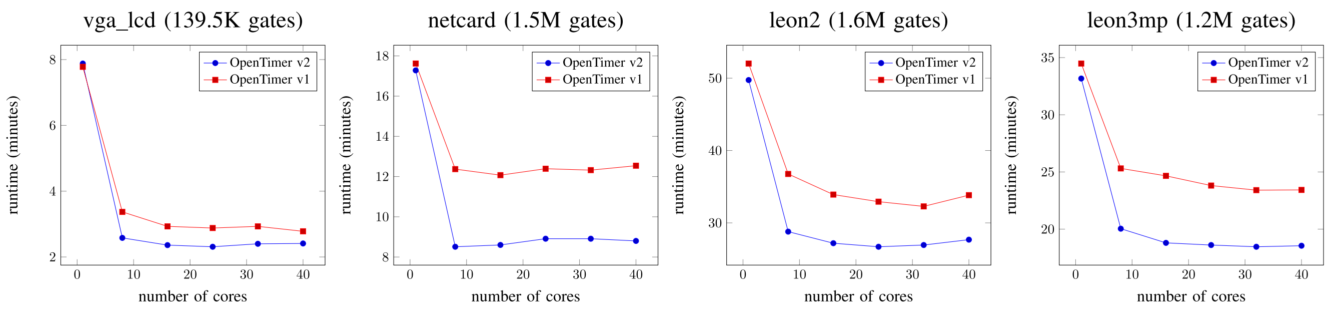


Fig. 11. Runtime scalability with increasing number of CPU cores on four large circuits, vga_lcd, netcard, leon2, and leon3mp.

8×. It is less likely to have a linearly strong scalability because the STA workload is graph oriented. Computational patterns are irregular and dependent on each other. The structures of the STA graph and design transforms restrict the maximum parallelism we have at a time. However, this is unpredictable for the STA engine itself.

Regardless of the core count used, the task-based strategy leads to much better runtime scalability over the loop-based pipeline. The maximum parallelism in the loop-based strategy is dominated by the number of independent nodes in a single layer. It is very difficult to maximize this quantity with available thread resources in an incremental environment. In contrast, the task-based strategy lets computation flow naturally with the timing graph structure. The scheduler autonomously optimizes the parallelism with dynamically generated tasks. As a consequence, the runtime difference between two strategies becomes remarkable as we increase the numbers of cores and tasks. For example, it took 12.54 min for OpenTimer v1 to finish whereas v2 reached the goal in only 8.80 min (30% faster). At about 8–16 cores where both tools reach the saturation point, the gap between OpenTimer v1 and v2 remains pronounced. We can clearly see OpenTimer v2 is more efficient than v1 at each core number.

4) Overhead of Task Parallelism: Task parallelism is advantageous in flowing computation along the timing graph, but creating a task graph has some overhead. Fig. 12 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 levellist and the parallel execution of tasks across all levels takes 71%. In fact, updating the levellist is one major bottleneck v1. It is very costly to maintain the data structure along with incremental timing because it requires dynamic changes at each iteration. The time spent on updating the levellist may overwhelm the benefit of loop parallelism. In terms of memory overhead, v2 bumps up about 10% of maximum resident set size (RSS) due to the management of task graphs (see Table IV).

C. Comparison With Parallax Software OpenSTA

We study the timing correlation between OpenTimer v2 and the commercial tool, OpenSTA, by Parallax Software

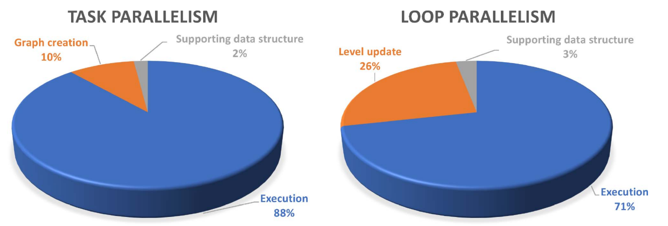


Fig. 12. Runtime profiling for task parallelism in OpenTimer v2 and loop parallelism in v1.

TABLE IV
MEMORY OVERHEAD OF OPENTIMER V2

Circuit	#Gates	#Nets	#Ops	v1	v2
b19	255.3K	255.3K	5641.5K	4.6 GB	4.9 GB
cordic	45.4K	45.4K	1607.6K	1.3 GB	1.37 GB
des_perf	138.9K	139.1K	4326.7K	3.0 GB	3.3 GB
edit_dist	147.6K	150.2K	3368.3K	3.1 GB	3.4 GB
fft	38.2K	39.2K	1751.7K	1.2 GB	1.29 GB
netcard	1496.0K	1497.8K	11594.6K	20.8 GB	23.7 GB
softusb_navre	6.9K	7.0K	427.8K	0.5 GB	0.54 GB
tip_master	37.7K	38.5K	1300.4K	0.8 GB	0.82 GB

TABLE V
COMPARISON BETWEEN OPENTIMER V2 AND THE COMMERCIAL TOOL OPENSTA [25] ON GENERATING THE TOP-1000 TIMING CRITICAL PATHS OF EACH DESIGN

Circuit	#Gates	#Nets	ρ_{slack}	ρ_path
c1355	180	221	0.99	100%
c1908	222	255	0.99	100%
c2670	344	501	0.99	100%
c3540	691	741	0.99	100%
c5315	918	1096	0.98	100%
c6288	1677	1699	0.95	97%
c7552	1147	1353	0.99	100%
usb_phy	923	928	0.95	98%
tv80	5285	5317	0.93	97%
des_perf	105K	106K	0.97	99%
vga_lcd	139K	139K	0.99	100%

ρ_{slack}: correlation coefficient in slack values
ρ_{path}: percentage of matched paths

Inc. [19], [25]. OpenSTA is an open-source gate-level static timing verifier that has been used by many design houses. Over the past decades, OpenSTA has conducted thousands of correlations with industrial sign-off timers. While it may not be fair to compare an academic tool with a commercial one due to many confidential infrastructures, we are particularly interested in establishing timing correlation between

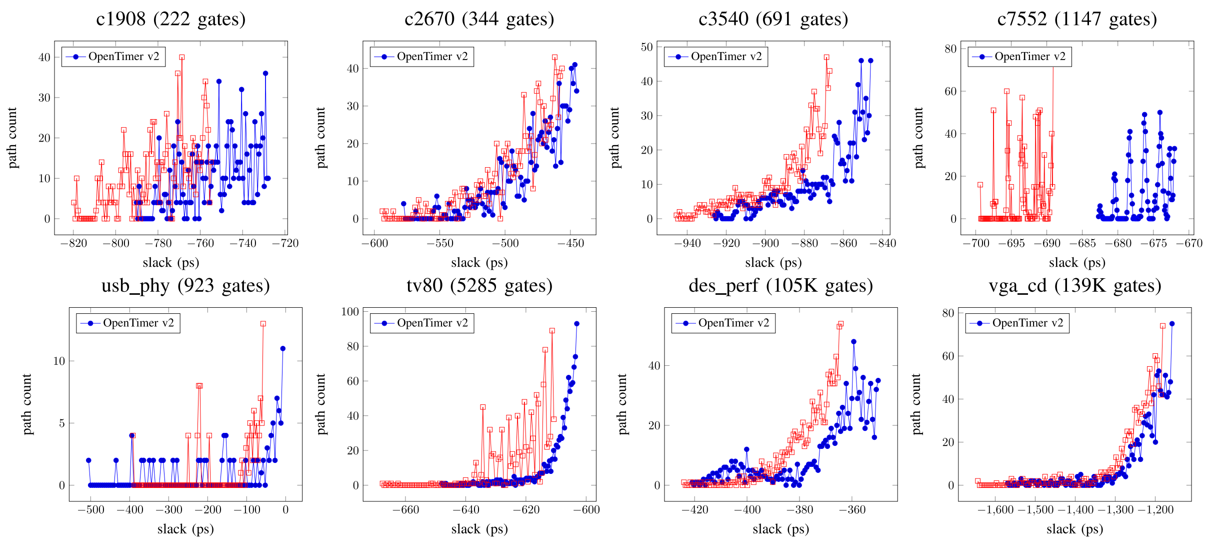


Fig. 13. Comparison of the top-1000 path slack distribution between OpenTimer v2 and OpenSTA on combinational circuits c1908, c2670, c3540, and c7552, and sequential circuits usb_phy, tv80, des_perf, and vga_lcd.

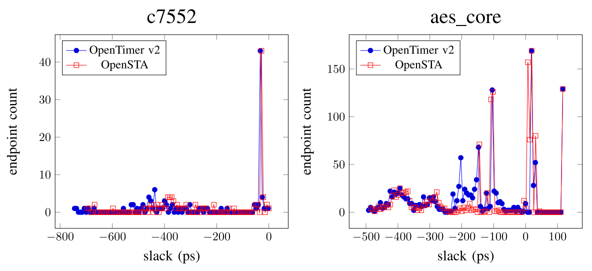


Fig. 14. Comparison of all violation endpoints between OpenTimer v2 and OpenSTA.

OpenTimer v2 and commercial tools. Perfect timing correlation between tools leads to faster timing closure, and thus reduction in cycle time.

The overall correlation data are shown in Table V. We selected seven combinational circuits and four sequential circuits from TAU15 benchmarks and used OpenTimer v2 and OpenSTA to report the top-1000 critical paths in a full timing run. We compare the accuracy using the correlation coefficient ρ between the two slack vectors extracted from the path reports. The two timing reports are positively correlated to each other with ρ ranging from 0.93 to 0.99. We observed a strong linear relationship between two reports. Fig. 13 visualizes the slack distributions between OpenTimer v2 and OpenSTA on four combinational circuits and four sequential circuits. The main reason for different slack values between two tools is the modeling of net delay and effective capacitance in RC networks. For path-based comparison, both tools produced almost identical results. For example, circuits vga_lcd and c7552 both matched 100% paths. We observed consistent correlation results also in violation endpoints. Fig. 14 compares the graph-based slack distribution across all violation endpoints on two circuits, c7552 and aes_core. The correlation coefficients are 0.99 for both circuits. Results are consistent in all other benchmarks with ρ in the range of 0.97 to 0.99 for violation endpoints.

V. Conclusion

In this article, we presented OpenTimer v2—a new parallel incremental timing analysis engine. Compared with the previous generation, we have rewritten the codebase from the ground up using modern C++17 and advanced parallel computing techniques to facilitate the design of parallel incremental timing. Our parallel decomposition strategy has delivered new performance scalability and programming productivity that were previously out of reach. For one particular example, OpenTimer v2 achieved up to 5.33× maximum speedup in one incremental timing iteration over v1. We have also shown a strong linear correlation to an industrial static timing verifier that has approached sign-off quality. OpenTimer v2 is open source, and we are committed to free sharing of our technical innovation to advance timing research.

Future work focuses on new GPU algorithms to speed up STA, in particular, full timing analysis. One possible direction is to leverage data-oriented design techniques for CPU–GPU collaborative computing. Prior research has demonstrated significant performance improvement in physical design and simulation [27]–[29]. A key benefit of the task-based approach is its extensibility. We can encapsulate the data-parallel GPU kernel in a task and couple it together with other dependent CPU tasks. Also, we plan to develop a conditional tasking interface in our Cpp-Taskflow project to support early termination of tasks for implementing efficient pruning techniques during the timing propagation [15]. Additionally, we are interested in incorporating model-order-reduction strategies into our task to speed up incremental delay calculation [30].

Acknowledgment

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

References

[1] T.-W. Huang *et al.*, "OpenTimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 895–902.

[2] J. Bhasker *et al.*, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Boston, MA, USA: Springer, 2009.

[3] J. Hu *et al.*, "TAU 2015 contest on incremental timing analysis," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 895–902.

[4] M.-C. Kim *et al.*, "ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite," in *Proc. IEEE/ACM ICCAD*, Nov. 2015, pp. 921–926.

[5] *WOSET*. Accessed: 2018. [Online]. Available: https://github.com/woset-workshop/woset-workshop.github.io

[6] *Qflow*. Accessed: 2018. [Online]. Available: http://opencircuitdesign.com/qflow/

[7] *VSDflow*. Accessed: 2018. [Online]. Available: https://www.vlsisystemdesign.com/vsdlibrary/

[8] *CloudV*. Accessed: 2018. [Online]. Available: https://cloudv.io/

[9] I. H.-R. Jiang *et al.*, "OpenDesign flow database: The infrastructure for VLSI design and design automation research," in *Proc. IEEE/ACM ICCAD*, 2016, pp. 1–6.

[10] J. Jung *et al.*, "DATC RDF: An academic flow from logic synthesis to detailed routing," in *Proc. IEEE/ACM ICCAD*, 2018, pp. 1–4.

[11] *LGraph*. Accessed: 2018. [Online]. Available: https://github.com/masc-ucsc/lgraph

[12] *Ophidian*. Accessed: 2018. [Online]. Available: https://gitlab.com/eclufsc/ophidian

[13] *OpenTimer*. Accessed: 2018. [Online]. Available: https://github.com/OpenTimer/OpenTimer

[14] *DARPA IDEA Program*. Accessed: 2018. [Online]. Available: https://www.darpa.mil/news-events/2019-05-31

[15] P. Lee *et al.*, "iTimerC 2.0: Fast incremental timing and CPPR analysis," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 890–894.

[16] C. Peddawad *et al.*, "iitRACE: A memory efficient engine for fast incremental timing analysis and clock pessimism removal," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 903–909.

[17] P.-Y. Lee *et al.*, "FastPass: Fast timing path search for generalized timing exception handling," in *Proc. IEEE/ACM ASPDAC*, 2018, pp. 172–177.

[18] K. E. Murray *et al.*, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *Proc. IEEE FPT*, 2018, pp. 110–117.

[19] *OpenSTA*. Accessed: 2018. [Online]. Available: https://github.com/abk-openroad/OpenSTA

[20] *OpenM P 4.5*. Accessed: 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[21] *Intel TBB*. Accessed: 2018. [Online]. Available: https://github.com/01org/tbb

[22] T.-W. Huang *et al.*, "CPP-Taskflow: Fast task-based parallel programming using modern C++," in *Proc. IEEE IPDPS*, 2019, pp. 1–28.

[23] *CPP-Taskflow*. Accessed: 2018. [Online]. Available: https://github.com/cpp-taskflow/cpp-taskflow

[24] 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.

[25] *Parallax Software*. Accessed: 2018. [Online]. Available: http://www.parallaxsw.com/

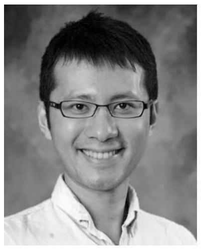
[26] *SLOC Count*. Accessed: 2018. [Online]. Available: https://dwheeler.com/sloccount/

[27] T. Fontana *et al.*, "How game engines can inspire EDA tools development: A use case for an open-source physical design library," in *Proc. ACM ISPD*, 2017, pp. 25–31.

[28] R. Netto *et al.*, "Ophidian: An open-source library for physical design research and teaching," in *Proc. WOSET*, 2018, pp. 1–4.

[29] D. Liechy, "Object-oriented/data-oriented design of a direct simulation Monte Carlo algorithm," in *Proc. AIAA*, 2014, pp. 1–11.

[30] H. Levy *et al.*, "A rank-one update method for efficient processing of interconnect parasitics in timing analysis," in *Proc. IEEE/ACM DAC*, 2000, pp. 75–78.

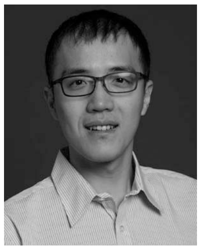


Tsung-Wei Huang received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University, Tainan, Taiwan, in 2010 and 2011, respectively, and the Ph.D. degree from the Electrical and Computer Engineering (ECE) Department, University of Illinois at Urbana–Champaign, Urbana, IL, USA, in 2017.

He is currently an Assistant Professor with the ECE Department, University of Utah, Salt Lake City, UT, USA. He has been building software systems from the ground up with a specific focus on parallel processing and timing analysis.

Dr. Huang was the recipient of the 2019 ACM/SIGDA Outstanding Ph.D. Dissertation Award for his contributions to parallel processing and timing analysis.

Guannan Guo received the B.S. degree from the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL, USA, in 2017, where he is currently pursuing the Ph.D. degree with research focus on circuit timing analysis, distributed systems, and scheduling algorithms.



Chun-Xun Lin received the B.S. degree in electrical engineering from National Cheng Kung University, Tainan, Taiwan, in 2009, and the M.S. degree in electronics engineering from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, in 2011. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL, USA.

His current research interests include parallel processing and physical design.

Martin D. F. Wong (Fellow, IEEE) received the B.S. degree in mathematics from the University of Toronto, Toronto, ON, Canada, in 1979, the M.S. degree in mathematics from the University of Illinois at Urbana–Champaign (UIUC), Urbana, IL, USA, in 1981, and the Ph.D. degree in computer Science from UIUC in 1987.

From 1987 to 2002, he was a Faculty Member of computer science with the University of Texas at Austin, Austin, TX, USA. He returned to UIUC in 2002, where he was the Executive Associate Dean with the College of Engineering from 2012 to 2018 and the Edward C. Jordan Professor of electrical and computer engineering. He is currently the Dean of the Faculty of Engineering, Chinese University of Hong Kong, Hong Kong.

Dr. Wong is a Fellow of ACM.