OpenTimer 2.0: A High-performance Timing Analysis Tool for VLSI Systems

Tsung-Wei Huang*, Chun-Xun Lin*, and Martin Wong*
*Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—Since the first release in 2015, OpenTimer has gained much attention from both industry and academia. OpenTimer is an award-winning tool in the ACM TAU Timing Analysis Contests (2014 through 2016) and the golden timer of many related CAD contests that encourage researchers and students to contribute their ideas to EDA community. These efforts also help energize academic research and developments in EDA systems, especially on the open-source front. After three year developments, we have announced OpenTimer 2.0 in 2018 – a major release under MIT license. We rewrote the entire OpenTimer codebase using modern C++17 and developed a new incremental timing engine using our parallel programming library Cpp-Taskflow. Compared to the previous generation, version 2 largely improved functionalities, standard input format supports, multi-threading, and runtime performance. OpenTimer is selected as part of the open-source silicon compiler project IDEA, funded by DARPA. We are committed to offer multi-year supports hopefully moving toward sign-off capability. OpenTimer is available at GitHub https://github.com/OpenTimer/OpenTimer.**

I. Motivation

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

II. The Version 1

In 2015, we release OpenTimer version 1. The previous generation was called UI-Timer. UI-Timer was the first place winner in the 2014 ACM TAU Timing Analysis Contest on Common Path Pessimism Removal. During the contest, we developed a new path-based timing analysis algorithm using constant time and space to represent a critical path. The results were far faster from all participants by more than an order of magnitude. Later on, we decided to release the source code of the tool and renamed it to OpenTimer. Since then, we have kept working on the core development and won several awards from the ACM TAU Timing Analysis Contest in 2015 through 2016. At the same time, we collaborated with both industry experts and academic scholars to organize timing analysis contests using OpenTimer as the golden timer. This included ACM TAU Timing Analysis Contest in 2016 through 2019, and 2015 IEEE/ACM ICCAD CAD Contest on Timing-driven Incremental Detailed Placement.

III. The Version 2

The results of our research and developments made us acquire the funding from DARPA under the category of IDEA project to support our continuing effort. in 2018 May, we made a new major release called OpenTimer 2.0 and switched the license from GPL to MIT. We rewrote the code base using modern C++17 and leveraged the parallel tasking model of our another open-source project called Cpp-Taskflow to redesign the core incremental timing engine. The main difference between v1 and v2 is the parallelization framework in the incremental timing, where v1 used OpenMP and v2 used Cpp-Taskflow.

A. Software Cost

TABLE I: Software Costs of OpenTimer v1 and v2

Tool	Task Model	SLOC	Effort	Sched	Dev	Cost
v1	OpenMP 4.5	9,123	2.04	0.70	2.90	$275,287
v2	Cpp-Taskflow	4,482	0.97	0.53	1.83	$130,523

Effort: development effort estimate, person-years (COCOMO model)
Sched: largest estimated schedule of a component (COCOMO model)
Dev: estimated average number of developers (efforts / schedule)
Cost: total estimated cost to develop (average salary = $56,286/year).

Table I measures the software costs between the two versions of the OpenTimer core using the Linux tool SLOCCount. The cost estimate is based on the constructive cost model (COCOMO) under the organic mode – *small teams with good experience working on a research-driven environment*. Compared with OpenMP tasking, Cpp-Taskflow offers a better programmability to describe graph workloads. In OpenTimer v2, a large amount of exhaustive OpenMP dependency clauses that were used to carry out dynamic tasking are now replaced with only a few lines of flexible Cpp-Taskflow code. We attribute this to the library programmability, which has the potential to affect the software cost in various aspects such as code complexity, development effort, and delivery costs (see Table I). While our measurement may not be perfect, it provides a valuable insight into the software cost caused

by the library programming model in a large-scale application made up of multi-year research effort.

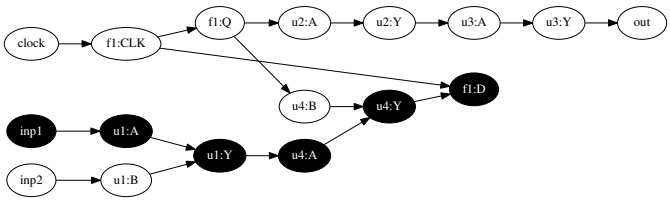

Fig. 1: An example task dependency graph of a single timing update.

B. Parallel Incremental Timing

In terms of algorithms, OpenTimer v1 relied on a bucket-list data structure to model the task dependency and performed parallel timing propagations in a pipeline fashion. We found it very difficult to go beyond this paradigm with OpenMP due to unpredictable graph structures during incremental timing update. With Cpp-Taskflow, we are able to break this bottleneck. Cpp-Taskflow's graph description language allows us to model both static and dynamic task dependencies regardless of graph structures. The task dependency graph works seamlessly with the timing graph, allowing computations to flow asynchronously rather than level by level. Figure 1 shows an example task dependency graph (critical timing path on black) of a single timing update on a sample circuit.

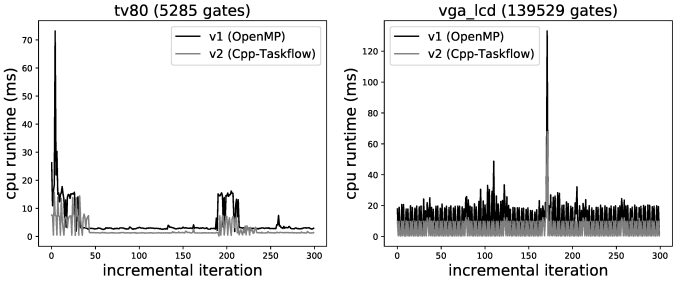

Fig. 2: Runtime comparisons of the incremental timing between OpenTimer v1 (OpenMP) and v2 (Cpp-Taskflow). The average runtime of v2 is about 2× faster than v1.

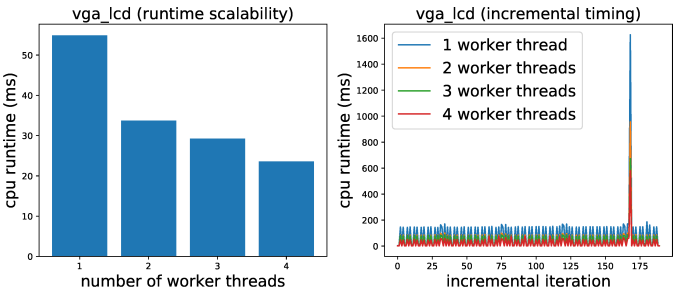

Fig. 3: Scalability of OpenTimer v2 and detailed profiles of the incremental timing across different number of threads.

Figure 2 demonstrates the performance comparison of the incremental timing between OpenTimer v1 and v2. For fair purpose, we disabled all new features in v2 and considered only the basic graph-based update to make both perform the same timing computations. We evaluated the runtime versus the incremental iterations on two real circuit designs tv80 (5.3K gates and 5.3K nets) and vga_lcd (139.5K gates and 139.6K nets) with 45nm NanGate cell library. As shown in Figure 2, v2 is consistently faster than v1 (2.14× on tv80 and 2.19× on vga_lcd). The profiler attributes 64% of the speed-up to the removal of levelization and 36% to the task-based timing updates. Next we demonstrate the scalability of v2. In this experiment we enabled all new features to perform complete timing analysis including path-based analysis on vga_lcd. Figure 3 shows the runtime scalability versus different core (thread) counts, and detailed profiles in incremental timing. The speed-up rate at four threads is about 2.33×. We also obtain a consistent speed-up at each incremental timing iteration. Depending on the circuit structure, the speed-up curve varies a lot. In a proprietary design, we achieved 3.7× faster with four threads.

C. Application Programming Interface

At programming level, OpenTimer v2 has incorporated many new changes. To enable efficient incremental timing, the API is categorized to three groups:

- **Builder**. 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 every time user calls a builder operation. It records what transformations need to be executed after an action has been called.
- **Action**. A TEP is materialized and executed when the timer is requested to perform an action operation. Each action operation triggers timing update from the earliest task to the one that produces the result of the action call. Internally, OpenTimer creates task dependency graph to update timing in parallel, including forward (slew, arrival time) and backward (required arrival time) propagations.
- **Accessor**. The accessor operations let you inspect the timer status and dump timing information. All accessor operations are declared as constant methods in the timer class. Calling them promises not to alter any internal members. For example, you can dump the timing graph into a dot format and use tools like GraphViz for visualization.

We also incorporated many new features and functionalities such as OpenTimer shell to enable interactive timing analysis and continuous integration framework to build up the contribution flow. Please refer to our GitHub at https://github.com/OpenTimer/OpenTimer for more details.