Title
OpenTimer Interface for LGraph

Permalink
https://escholarship.org/uc/item/2g48d71z

Author
Ganpati, Rohan Prakash

Publication Date
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPENTIMER INTERFACE FOR LGRAPH

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Rohan Prakash Ganpati

June 2019

The Thesis of
Rohan Prakash Ganpati is approved:

Professor Jose Renau, Chair

Professor Heiner Litz

Professor Scott Beamer

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

OpenTimer Interface for LGraph

by

Rohan Prakash Ganpati

In today's world, the acceleration in taping out ASIC and FPGA chips is majorly limited by the productivity in its development. While there are digital design tools that are exceptionally good, there is a huge gap in the interaction between these tools, resulting in overheads causing delays in the development cycle.

LGraph is an open-source database that represents digital design from any stage of the design flow in a unified format. It supports inputs in several hardware description languages. Tightly integrated with several open-source EDA tools, LGraph aims to provide live results for synthesis and simulation for small changes in the design.

My contribution includes the integration of OpenTimer, an open-source tool that performs timing analysis pre and post place and route. Presently it is possible to receive feedback on timing for a design represented as an lgraph. With further development, our research group looks forward to obtaining live feedback for timing analysis.

Acknowledgments

Sri Gurubyo Namaha. I've been blessed with great teachers and mentors throughout my life who have moulded me for the person I am today.

I'd first want to thank my advisor, Prof. Jose Renau for providing me an opportunity to work on a very interesting and unique research topic. He was always available, understanding and steered me into the right direction to solve problems quicker. His perseverance and way of life has truly inspired me and I hope to follow it.

Sincere thanks to Prof. Heiner Litz and Prof. Scott Beamer at UCSC for providing feedback that helped shape my thesis better. I'd also like to thank Prof. Tsung-Wei Huang at UIUC for answering my questions on OpenTimer.

Humble thanks to my boss, Silas McDermott at Cadence Design Systems, for extending me a full-time position; out of his way; months before my graduation; to let go of my burden with my job search so that I could focus on my thesis.

I largely benefited from working at the MASC research group where my labmates were very helpful and guiding. Particularly, I would like to thank Rafael, Sheng, Akash and Nursultan for helping me with even the most silliest questions I've had.

Heartiest thanks to my former mentor at Solarillion Foundation (SF), Vineeth Vijayaraghavan without whom I'd ever not be where I am. I'd also like to thank my peers from SF who I've had a great time with and learned a lot from.

Finally, I'd like to thank my mother who has sacrificed a lot in her life for me to pursue my goals. Without her support, none of this could have been possible.

Chapter 1

Introduction

> Don't listen to the naysayers.
>
> Arnold Schwarzenegger

Current hardware design techniques lack productivity due to various reasons in the Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Arrays (FPGA) design flow. Some of the top reasons are the time it takes to complete its process and the lack of a unified format to represent processes throughout the design flow.

In most cases, the changes made to the designs are small, yet, it takes a considerable amount of time for any process in the design flow such as simulation, synthesis, place and route, etc. In real world, this means that if a small portion of the code is modified in the front-end, it takes several hours for this modification to reach the back-end and several more for them to fix it. This only gets worse as the project approaches its deadlines.

LiveSynth [9] and SMatch [10] are incremental techniques that reduce the time for synthesis to the extent that it feels like the changes made to the design are reflected instantly. Inspired by these techniques, LGraph [11] was created with a notion to serve as the LLVM for hardware design. It is a unified open-source format that represents digital design from several processes throughout the digital design flow. LGraph aims to provide a live hardware development feel to its community. Any change made at a certain process will be reflected instantaneously to all its dependent processes that it has effects on. This would drastically enhance productivity in the workflow and help alleviate the problems discussed above.

While designing a chip, performance is one of the major parameters that are important for the designers. Static Timing Analysis (STA) is one of the several methods used to verify performance of a design. STA is performed post synthesis and post place and route. STA performed after place and route is more important because it contains parasitic information from the RC networks.

Typically, when a chip is designed, the specifications are first discussed. Then, the RTL design and verification environments are developed and simulated. Later, this design is synthesized and STA is performed to evaluate the performance. Then, this design is placed and routed and STA is performed again. Now, if there is a small change in the design, it would have a severe impact as it has to go through all these processes all over again. These are some of the reasons that cause delay and develop difficult scenarios during the development process. Thus enhancing its productivity is an important area of research.

OpenTimer [4], is an open-source high-performance static timing analysis tool with incremental timing capabilities. It is designed with fast and accurate algorithms to produce results at high speed. OpenTimer provides support for performing STA for a particular design in a possible stage of the design flow, when integrated with LGraph.

However, integrating OpenTimer and LGraph possessed three main challenges. The first challenge was understanding the internals of LGraph. Since LGraph is being actively developed, there is not much documentation for it which makes the development a tedious task. An exhaustive study and debugging of the source code was required to overcome this challenge.

The second challenge was comprehending the internals of OpenTimer. Since the internals of OpenTimer and LGraph are completely different, an analysis of the source code was required to understand its internals and further inspection was performed to analyze its performance when using its C++ API to create circuits on the fly.

The third challenge was integrating OpenTimer with LGraph. Since OpenTimer supports a different input file format as opposed to LGraph's representation, it causes compatibility issues. This was overcome by translating an lgraph to a format that OpenTimer supports. The translation was achieved by using OpenTimer's C++ API where creating 1000 cells takes 0.2s. Since LGraph aims to provide live feedback for small changes to a digital design, this overhead is negligible.

This dissertation discusses the integration of LGraph and OpenTimer. Chapter 2 gives a background on LGraph focussing its internals and interface with an illustrative example. Chapter 3 provides a background on OpenTimer emphasizing its internals and interface with a suitable example. Chapter 4 describes the integration of LGraph and OpenTimer describing its internals and interface with a descriptive example. Finally, Chapter 5 delivers the conclusion of this thesis along with the future work.

Chapter 2

LGraph

> Knowing is not enough, we must apply. Willing is not enough, we must do.
>
> Bruce Lee

This chapter introduces LGraph - a graph optimized for live synthesis and simulation. In order to integrate LGraph and OpenTimer, it is necessary to understand LGraph's internals which are its database organization, structure, node types, iterators and shell interface which is discussed with an example. The database, structure and node types provide a high-level idea on the internals of LGraph while the iterators are used for efficient traversals on an lgraph [1]. The shell interface is used to perform transformations on the lgraph which is discussed later.

---

[1] LGraph refers to the infrastructure and lgraph refers to the unified format that represents digital design

2.1 Introduction

Different phases of the VLSI design flow use different formats to represent a design. The common convention is to use Verilog, Berkeley Logic Interchange Format (BLIF) during logic synthesis, Library Exchange Format (LEF)/ Design Exchange Format (DEF) during physical design, Liberty for timing analysis and Graphic Data System (GDS) for layout.

LGraph [11] is an open-source database for digital design in different phases of the VLSI design flow. LGraph can be thought of as a LLVM for hardware design. It can interface with various Hardware Description Languages (HDLs) such as Verilog, Pyrope and other formats such as LEF/DEF, Liberty. It also interfaces with ABC for logic synthesis and OpenTimer for static timing analysis.

LGraph features a shell interface - lgshell which allows users to use it as an extendable toolset. Additionally, LGraph features an API through which the internal data structure can be manipulated.

2.2 Related Work

Synopsys Milkyway [12] is a library that supports Synopsys' EDA tools from synthesis through place and route until sign-off. Synopsys' tools such as the Design Compiler, PrimeTime can read and write in the Milkyway format but it is a proprietary format and isn't ideal for academic research.

OpenAccess is an open format that supports interoperability among EDA tools.

It has an application programming interface and supports authorization for interoperability between multiple EDA vendors. However, there are some legal restrictions in using it which its limits usage.

FIRRTL [7] is an open source format but it is based on Scala and thus has a steep learning curve for new users. RSyn [2] and Ophidian [3] are open-source formats but they target only physical design. These formats are also task-specific and not the best option for integration.

Yosys [15] is a framework for RTL synthesis which uses RTLIL as an interrepresentable format. Yosys supports synthesizable Verilog as an input and converts it to BLIF and similar formats but LGraph has much smaller read and write times compared with RSYn and Yosys.

2.3 Internals

2.3.1 Database

The LGraph database is based on two concepts - memory maps and struct of array. They are used for fast persistence and to exploit memory locality respectively. The database contains modules and target technologies grouped as graph libraries and tech libraries respectively. The graph library contains the modules which represent the design and the tech library contains the associated technology related to the design. Figure 2.1 represents the LGraph database.

The representation in the database can be used to represent the basics of any

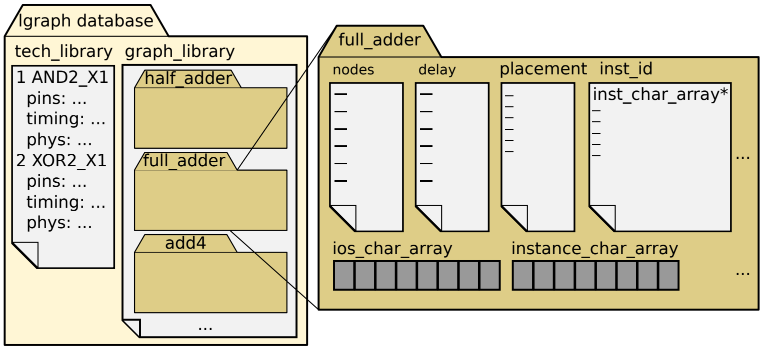


Figure 2.1: LGraph's [11] database organization

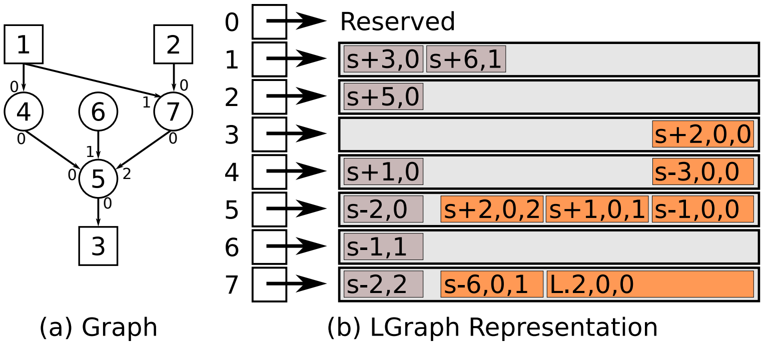
format in the design flow. However, users can modify it to best suit their application while maintaining the rudimentary set of tables that LGraph requires.

2.3.2 Structure

LGraph uses a bidirectional graph as few of the operations require forward traversals and others require backward traversals. The data structure is designed with an intent to satisfy synthesis graph requirements. The size of the nodes are 64 bytes in order to extract the best out of cache locality.

The major elements of LGraph are node, node pin, and edges. The node represents a logic gate, the node pin represents the pin in the logic gate, and the edges connect a pair of node pins. When traversing between a source and destination, the inputs and outputs pertaining to that graph are called as graph IOs.

Figure 2.2 shows a LGraph representation of a regular graph. The boxes/ circles are nodes, the numericals outside them are node pins and the lines connecting them are edges.



(a) Graph (b) LGraph Representation

Figure 2.2: LGraph's internal data structure

2.3.3 Types

The LGraph types are enums that are classified into several categories based on ranges. The first category is allocated for LGraph types, the second category is allocated for subgraph types used for design hierarchy representation, the third category are used for constants, and the last category are used for standard cell technology mapping. Few of the LGraph types are represented in Table 2.1.

2.3.4 Iterators

LGraph may represent a complete hierarchical design, a particular module of that design or even a portion of the design between a specified input and output. Iterating between these points are very crucial and needs to be done efficiently. LGraph supports two types of iterators viz. node iterators and edge iterators to perform these operations.

Table 2.1: Various LGraph node types

NodeType	Functionality
Not_Op	logical negation of inputs
And_Op	logical AND
Or_Op	logical OR
Xor_Op	logical XOR
Join_Op	the {} operator in Verilog
Pick_Op	the [] operator in Verilog
LessThan_Op	LessThan comparator
GreaterThan_Op	GreaterThan comparator
LessEqualThan_Op	LessEqualThan comparator
GreaterEqualThan_Op	GreaterEqualThan comparator
Equals_Op	arithmetic functions equals comparator
Mux_Op	generic multiplexers
ShiftRight_Op	shift input right by a given number of bits
ShiftLeft_Op	shift input left by a given number of bits
GraphIO_Op	keyword input, output and inout in Verilog
SubGraph_Op	instantiation of another module
TechMap_Op	Coarse-Grained elaborated standard cell types
U32Const_Op	constant in Verilog
StrConst_Op	4 state variables in Verilog

2.3.4.1 Node Iterators

The fast iterator is used for unordered but very fast traversal, the forward iterator is used for forward propagation from each input/constant, and the backward iterator is used for backward propagation from each output. Figure 2.3 shows the node iterators in LGraph.

```
for(auto nid : g.fast())     { }
for(auto nid : g.forward())  { }
for(auto nid : g.backward()) { }
```

Figure 2.3: Node iterators in LGraph

2.3.4.2 Edge Iterators

The inp_edges iterator is used to iterate over input edges and the out_edges is used to iterate over output edges. Figure 2.4 shows the edge iterators in LGraph.

```
for(auto& edge : node.inp_edges()) { }
for(auto& edge : node.out_edges()) { }
```

Figure 2.4: Edge iterators in LGraph

2.4 Interface

LGraph features an interactive shell - lgshell to support the Pass and InOu transformations. The lgshell commands can be grouped into passes, InOu's and lgraph operations. Few of these commands are mentioned in Table 2.2.

Table 2.2: Commonly used lgshell commands

Commands	Functionality
inou.graphviz	export lgraph to graphviz dot format
inou.yosys.fromlg	write Verilog using yosys from lgraph
inou.yosys.tolg	read Verilog using yosys to lgraph
pass.dce	optimize an lgraph with a dce, gen mapped
pass.sample	counts number of nodes in an lgraph
pass.opentimer	timing analysis on lgraph
lgraph.create	create a new lgraph
lgraph.open	open an lgraph if it exists
lgraph.stats	print the stats from the passed graphs

Figure 2.5 depicts a Verilog file which is fed as an input to LGraph. Transformations can be performed on this file and converted to several different formats.

The Verilog file can be converted to an lgraph by using the first command in

```
module simple_add(
                input [7:0] a,
                input [7:0] b,
                output signed [7:0] h
                );

signed wire [7:0] as = a;
signed wire [7:0] bs = b;

wire [7:0] f = as + bs;
assign h = as + bs - as;

endmodule
```

Figure 2.5: Sample Verilog code for addition

Figure 2.6. This transformation is an InOu and is performed by interfacing with yosys.

```
inou.yosys.tolg files:path/to/file/filename.v
lgraph.open name: filename |> inou.graphviz
```

Figure 2.6: lgshell commands to convert a Verilog file to an lgraph and represent it in DOT format

This lgraph can be converted to a DOT format by using the second command in Figure 2.6. This transformation is also an InOu and is performed by interfacing with Graphviz. Figure 2.7 shows the graphviz representation of the Verilog code in Figure 2.6.

Several passes can be performed on the lgraph converted from the Verilog file. Static timing analysis can be performed on this lgraph or it can be converted to another format such as Pyrope, Chisel etc.

The commands shown in Figure 2.8 convert the lgraph (converted from the Verilog file) back to a netlist. This transformation is also an InOu and is performed by

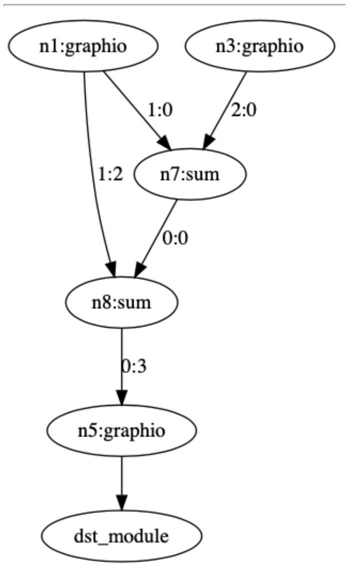


Figure 2.7: Graphviz representation of the simple_add lgraph

```
inou.yosys.tolg files:path/to/file/filename.v
lgraph.open name: filename |> inou.yosys.fromlg
```

Figure 2.8: lgshell commands to convert a Verilog file to an lgraph and the lgraph back to Verilog

interfacing with Yosys. The converted Verilog netlist is represented in Figure 2.9.

```
  module simple_add(a, b, h);
  input [7:0] a;
  input [7:0] b;
  output [7:0] h;
  wire [7:0] lg_0;
  wire [7:0] lg_1;
  assign lg_0 = $signed(a) + $signed(b);
  assign lg_1 = lg_0 - a;
  assign h = lg_1;
  endmodule
```

Figure 2.9: Verilog netlist converted from the simple_add lgraph by LGraph

2.5 Conclusion

In this chapter, the key features of LGraph including its organization, structure, types were discussed. An example using its application programming interface and shell interface were also shown.

Some of the top features of LGraph include its multi-language support and interoperability between the entire VLSI deisgn flow from RTL to layout in a very fast manner.

Future work includes building a custom timing analysis engine and a placement & routing tool for LGraph, and integrating a SAT solver for verifying transformations done with LGraph.

Chapter 3

OpenTimer

> If everything seems under control,
>
> you're not going fast enough.
>
> Mario Andretti

This chapter introduces OpenTimer - an open-source high-performance timing analysis tool for VLSI synthesis. In order to integrate LGraph and OpenTimer, it is important to understand OpenTimer's internals which are its design philosophy, tool configuration, C++ API and shell interface which is discussed with an example. The design philosophy provides a high-level description on the software architecture, the tool configuration talks about the file formats supported, the C++ API and shell interface are two ways through which OpenTimer can be used to query timing information.

3.1 Introduction

Static Timing Analysis (STA) is an essential process in the Electronic Design Automation (EDA) flow. Given a clock frequency, STA is used to simulate the expected timing of a design and check for possible timing violations.

OpenTimer [4] is an open-source high-performance timing analysis tool for VLSI systems. Some of the its key features include parallel incremental timing, Common Path Pessimism Removal (CPPR), block-based and path-based timing analysis. The tool accepts industry standard input file formats.

OpenTimer features a user-friendly Application Programming Interface (API) and an interactive shell to query timing realted information. The tool can be integrated to other projects and supports multiple ways for the integration.

3.2 Related Work

Synopsys PrimeTime [13] is an industry leading sign-off solution for timing. Its major features are core static timing analysis and multi-scenario analysis. Cadence Tempus [1] is another industry leading sign-off solution for the same. Some of its major features are integration with other tools in its flow and with the cloud. Both these tools are proprietary and their subprocesses are obscured. Therefore, they aren't ideal for open-source research.

iitRACE [8] is an academic incremental timing analysis tool with clock pessimism removal. Its primary focus is to be memory efficient. iTimerC [6] is another academic incremental timing with CPPR analysis. Both these tools are good but OpenTimer outperforms them in terms of accuracy and speed.

OpenSTA [5] is a static timing analysis tool from parallax software that recently went open source. It is a relatively newer release and certainly a tool to be on the active lookout for.

3.3 Internals

3.3.1 Overview

OpenTimer is divided into three phases as shown in Figure 3.1. In the first phase, it performs a parallel read of the input files. In the second phase, it performs parallel timing analysis based on the input files. In the final phase, timing information can be queried from the shell or the API.

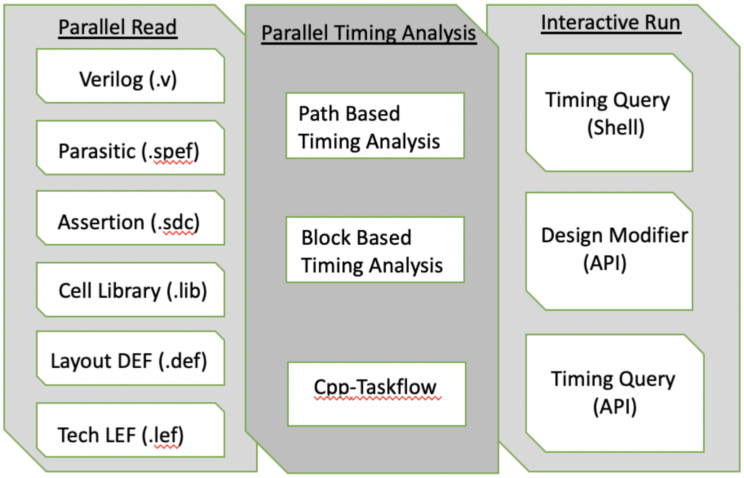


Figure 3.1: Program flow of OpenTimer

Since optimization transforms have the capacity to change designs, it could potentially affect timing information. To avoid this problem, the pipeline task scheduler used in OpenTimer 1.0 was upgraded to a parallel incremental timing engine using Cpp-Taskflow in OpenTimer 2.0 which was based on C++17.

The improvement of performance from OpenTimer 1.0 to OpenTimer 2.0 is majorly based on replacing OpenMP based parallelization using Cpp-Taskflow. Cpp-Taskflow is an efficient parallel programming library for complex task dependencies.

3.3.2 Input Files

OpenTimer complies with industry-standard format for input files which include a Verilog netlist, liberty, Standard Parasitic Exchange Format (SPEF), and a Synopsys Design Constraint (SDC) file.

3.3.2.1 Liberty (.lib) file

OpenTimer requires two liberty files that contains the cells and their associated timing information such as delay, capacitance etc. One of them would depict the early characteristics and the other would depict the late. In either case, these cells have to be ones that are available to the design.

3.3.2.2 Verilog Netlist (.v) file

OpenTimer requires a Verilog netlist file that contains gate level description of the circuit design. These cells should correspond to a cell from the liberty file. At the moment, OpenTimer does not support hierarchy in the design but is in active development to accomodate it.

3.3.2.3 Standard Parasitic Exchange Format (.spef) file

OpenTimer reuires a SPEF file that contains the parasitics of a group of nets in the form of a RC network. This includes internal nodes and wire resistances between them.

3.3.2.4 Synopsys Design Constraint (.sdc) file

OpenTimer requires a SDC file that contains timing conditions of a design in a tcl-based format. This includes the design intent as well as constraints for further processes in the design flow. At the moment, OpenTimer supports a limited number of commands but is in active development to accommodate more.

3.3.3 API Categories

OpenTimer's design philosophy reveals its parallel incremental timing features. They are distinguished into three groups viz. builder, action, and accessor based on its performance and usability as mentioned in Table 3.1.

Table 3.1: OpenTimer's API Categories

Type	Description
Builder	create lazy tasks to build an analysis framework
Action	carry out builder operations to update the timing
Accessors	inspect the timer without changing any internal data structures

- **Builder:** When a set of builder operations are called, OpenTimer creates a task execution plan (TEP) by adding these operations to a lineage graph. The graph takes care of maintaining its status based on action operations and is performed with the help of Cpp-Taskflow by creating dependency graphs. Some of the common builder operations include read_celllib, insert_gate, set_slew etc.

- **Action:** When an action operation is performed, the TEP is materialized and executed. The dependency graph updates timing including forward and backward propogations in parallel. After the action call is completed, the lineage graph is updated with timing. Some of the common action operations include update_timing, report_timing, report_slack etc.
- **Accessor:** Accessor operations provide the ability to output timing related information. This can also allow the user to check on the status of the timer. These operations can be graphically visualized using tools like GraphViz. Some of the common accessor operations include dump_timer, dump_slack, dump_net_load etc.

3.4 Interface

3.4.1 Built-In Shell

OpenTimer features an interactive shell for timing analysis. These shell commands are grouped to a builder, action or accessor operation. Some of the most common shell commands under builder include reading input files and connectivity based operations; under action include updating and reporting timing related information; under accessor include dumping timing related information on the shell.

The sample circuit consists of five cells - NAND1 gate that has 2 input signals

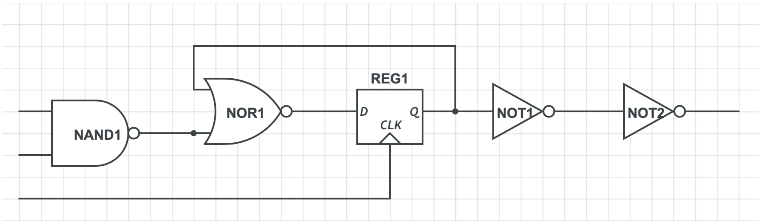


Figure 3.2: Sample circuit to illustrate the usage of the OpenTimer shell

connecting to NOR1 gate. The NOR1 gate connects to a DFF and receives input back from the DFF. The DFF is connected to a NOT1 gate which is in turn connected to a NOT2 gate.

The configuration file depicted in Figure 3.3 inputs a sample liberty file that contains the timing related information of the cells used in the design, a sample Verilog file that describes the circuit as shown in Figure 3.2, and a sample SDC file that contains more timing information about the specific design constraints. It enables common path pessimism removal. Upon completion of these operations, it dumps the taskflow graph, reports the most critical path and dumps the current timing graph.

```
read_celllib sample.lib
read_Verilog sample.v
read_sdc     sample.sdc
cppr -enable
dump_taskflow
report_timing num_paths 1
dump_graph
```

Figure 3.3: Configuration file to illustrate the usage of the OpenTimer shell

The configuration file for OpenTimer represented in Figure 3.3 when executed in the OpenTimer shell, generates an output, depicted in Figure 3.4. This output file reports the timing for the most critical path in the design. It lists the start point, end point, delays, time, direction and type for each instances. It also mentions the slack for the design.

The configuration file can be fed to the shell or each of the commands can be individually fed in to the shell. An alternate is to use the API which is demonstrated in the next subsection.

```
Startpoint     : inp1
Endpoint       : f1:D
Analysis type  : min
------------------------------------------------------------------
       Type        Delay          Time    Dir   Description
------------------------------------------------------------------
       port        0.000         0.000   fall   inp1
        pin        0.000         0.000   fall   u1:A  (NAND2X1)
        pin        2.786         2.786   rise   u1:Y  (NAND2X1)
        pin        0.000         2.786   rise   u4:A  (NOR2X1)
        pin        0.181         2.967   fall   u4:Y  (NOR2X1)
        pin        0.000         2.967   fall   f1:D  (DFFNEGX1)
    arrival                      2.967          data arrival time

related pin       25.000        25.000   fall   f1:CLK (DFFNEGX1)
 constraint        1.518        26.518          library hold_falling
   required                     26.518          data required time
------------------------------------------------------------------
      slack                    -23.551          VIOLATED
```

Figure 3.4: Critical path reported by OpenTimer for the discussed sample circuit and configuration file

3.4.2 C++ API

OpenTimer features a number of methods for timing analysis. Similar to the shell, they are grouped to a builder, action, or accessor operation. Some of the most common API methods under builder include insertion, deletion of a net, gate, connection, disconnection of a pin; under action include updating and reporting timing

Table 3.2: Commonly used OpenTimer API methods

Type	Form	Description
insert_gate	builder	inserts a gate (instance) to the design
insert_net	builder	inserts an empty net to the design
connect_pin	builder	connects a pin to a net
report_at	action	reports the arrival time at a pin
report_slew	action	reports the transition time at a pin
report_rat	action	reports the required arrival time at a pin
dump_graph	accessor	dumps the timing graph to an output stream
dump_taskflow	accessor	dumps the lineage graph to an output stream
dump_timer	accessor	dumps the statistics of the design

information; under accessor include dumping timing realted information. Table 3.2 shows few commonly used OpenTimer API methods.

The code snippet shown in Figure 3.5 is an example usage of OpenTimer's C++ API. It deletes NOT1 gate in Figure 3.2 and replaces it with an AND gate. The timer class under namespace ot is the entry point for the API in which the core methods for timing analysis are present.

The cell library, Verilog netlist and its SDC file are first read. Then, the required gates and nets are inserted in the design. Later, appropriate pin connections and disconnections are performed. Finally, the critical path is displayed.

3.5 Conclusion

In this chapter, the key features of OpenTimer, including its design philosophy, tool configuration were discussed. An example usage of its application programming interface and shell interface are also shown.

Some of the top features of OpenTimer include its non-conventional parallel

```
ot::Timer timer;

timer.read_celllib(”sample_stdcells.lib”, ot::MIN)
     .read_celllib(”sample_stdcells.lib”, ot::MAX)
     .read_Verilog(”sample.v”)
     .read_sdc(”sample.sdc”);

timer.insert_gate(”SAMPLE_GATE_1”,”AND_X1”)
     .insert_net(”SAMPLE_NET_1 ”)
     .disconnect_pin(”INV_X1”)
     .connect_pin(”SAMPLE_GATE_1”, ”SAMPLE_NET_1”)
     .connect_pin(”AND_X1”, ”SAMPLE_NET_1 ”)
     .connect_pin(”SAMPLE_GATE_1”, ”DFF_X1”);

auto critical_path = timer.report_timing(1);

std::cout << critical_path << '\n';
```

Figure 3.5: Sample C++ code to illustrate OpenTimer’s API method

incremental timing analysis engine with Cpp-Taskflow and Common Path Pessimism Removal.

Results from the TAU competition prove that OpenTimer is the fastest open-source timing analysis tool compared to its peer academic timing analysis tools that have similar features [4].

Future work includes expanding the SPEF parser to include more features in parasitic extraction, expanding the Verilog parser to include hierarchical designs, and later supporting behavioural Verilog.

Chapter 4

Integration

> Risk is the price you pay for opportunity.
>
> Tom Selleck

This chapter focusses on the integration between LGraph and OpenTimer. To understand the transformation of the internals from a format that is represented by LGraph to a format that is supported by OpenTimer, the integration process is discussed with an example using the lgshell.

4.1 Introduction

LGraph's vision is to support interoperability in the VLSI design flow from RTL to layout. Hugely motivated by LiveSynth [9], LGraph targets to achieve it by producing results in a few seconds.

OpenTimer is one of the several open-source EDA tools integrated with LGraph that performs STA. This integration is effective as it is an one-stop tool that can be used to perform STA post synthesis and post place and route. It is also envisioned to provide live STA feedback when a design is written in an HDL supported by LGraph.

A major challenge in this integration was that the internals of OpenTimer and LGraph are completely different. This was overcome by the OpenTimer pass that performed translation for effective communication between both softwares.

The OpenTimer pass traverses lgraphs, constructs equivalent circuits on the fly, computes timing information and annotates it to the lgraph. With LGraph's goal of providing blazing-fast results and its requirement in performing timing analysis, OpenTimer proved to be compatible for integration while benchmarking it.

Table 4.1: Benchmark test for a Ripple Carry Adder using OpenTimer's API method

Circuit	Cells	Time
Ripple Carry Adder using Full Adder cells	1,000	00.278s
	10,000	01.499s
	100,000	15.981s

A Ripple Carry Adder was constructed (on-the fly) using OpenTimer's API methods with 1,000, 10,000, and 100,000 full adder cells. Table 4.1 shows the speed of performing this task using OpenTimer's API methods.

The integration process shown in Figure 4.1 is described as follows:

- The input files required for OpenTimer are read in one or more options via LGraph.
- Then the lgraph [1] present in the database is targeted, traversed and an equivalent circuit is built using OpenTimer's builder API functions.

---

[1] This lgraph could be in any stage of the design flow.

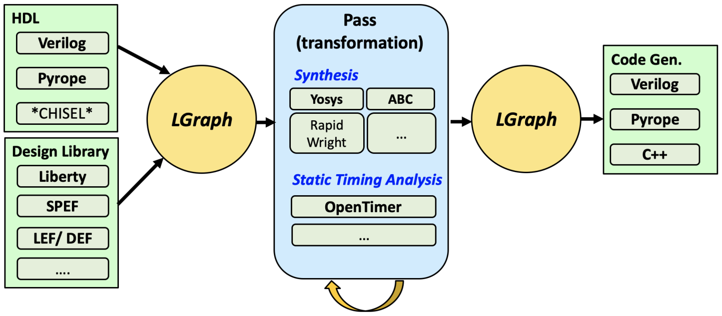


Figure 4.1: Integration flow between LGraph and OpenTimer including other Passes and InOu's

- Later, the timing information is computed using this circuit based on the user's option.
- Finally the delay is annotated back to the lgraph.

4.2 Internals

4.2.1 Reading Files

OpenTimer requires two liberty files for associating cells to a cell library's timing information, a SDC file for tcl-based input describing its timing conditions and a SPEF file that contains parasitic information of the nets in a RC network. It also requires a Verilog netlist but that is provided as an lgraph which is discussed later in this chapter.

The process for reading files is described in Algorithm 1. There are 3 inputs that are required from the user - a liberty file, a SDC file and a SPEF file. The liberty can be two individual files that contain early and late characteristics or a single file

Algorithm 1 read_file

```
procedure READ_FILE(.lib, .lib_max, .lib_min, .SDC, .SPEF)
    if len(.lib_max)==0 && len(.lib_min)==0 then
        timer.read_celllib(.lib);
    else
        timer.read_celllib(.lib_max,ot::MAX);
        timer.read_celllib(.lib_min,ot::MIN);
    end if
    parse.sdc(.SDC);
    timer.read_spef(.SPEF);
end procedure
```

which contains both.

The liberty and SPEF files are read through OpenTimer's API methods read_celllib and read_spef respectively. The SDC files are passed to another method which parses it.

4.2.2 SDC Parser

OpenTimer uses a SDC parser from Synopsys's TAP-in tools [14] which makes it partially unportable. Since it is anyways limited to a set of commands such as create_clock, set_input_delay, set_output_delay, set_input_transition, and set_load, it is reasonable to have a custom SDC parser. Algorithm 2 describes the procedure for parsing a SDC file.

The .SDC file is input from the user and is parsed line by line to check for tcl commands supported by OpenTimer. When such commands are found, the equivalent OpenTimer API methods are used to translate and execute them.

The create_clock method creates a clock given its name and period. The set_at,

Algorithm 2 parse_SDC

```
procedure PARSE_SDC(.SDC)
    while line gets next line in the file do
        if line contains create_clock then
            name ← clock_name
            period ← clock_period
            timer.create_clock(name, period);
        else if line contains option then
            option ← set input delay or set input transition or set output delay
            name ← input name or output name
            min_max ← MIN or MAX
            rise_fall ← RISE or FALL
            delay ← input delay or output delay
            timer.option(name, ot::MIN/MAX, ot:RISE/FALL, delay);
        end if
    end while
end procedure
```

set_rat and set_slew methods set the input delay, output delay, and input transition delay respectively. Apart from the name of the input or output, these commands require extra information about its delay to indicate if it is a min/max and rise/fall.

4.2.3 Building Circuit

Once the input files required for OpenTimer are loaded, an equivalent circuit is constructed from a lgraph. The steps required to perform this task is described in Algorithm 3.

The graph is traversed using a node iterator and the cell name and instance names are stored. These are used to create cells mapped to its cell library. The insert_gate method is used to perform this operation.

Next, the edge iterator is used to iterate the output edges of each nodes and

Algorithm 3 build_circuit

```
procedure BUILD_CIRCUIT(LGraph *g)
    for const auto &nid : g→ forward() do
        cell_name ← name_of_cell
        instance_name ← name_of_cell_instance
        timer.insert_gate(cell_name, instance_name);
        for const auto &edge : node.out_edges() do
            net_name ← name_of_net
            nodepin_name ← name_of_nodepin
            timer.insert_net(net_name);
            if edge is graph input then
                timer.insert_primary_input(net_name);
            else if edge is graph output then
                timer.insert_primary_output(net_name);
            end if
            timer.connect_pin(cell_name:nodepin_name, net_name);
        end for
    end for
end procedure
```

the net names and the node pin names are stored. The insert_net method is used to create these wires. While traversing, the primary inputs and outputs are separately stored and the insert_primary_input, insert_primary_output methods are used to create primary inputs and outputs.

Finally, the connect_pin method is used to connect the node pins of the corresponding node to the net connected to it.

4.2.4 Computing Timing

Upon building the circuit, timing information can be queried by the user as described in Algorithm 4. Inputs from the user are typically optional but include printing the critical path or the circuit related information or dumping the graph information.

The timer is first updated to maintain the latest information about the circuit. This is done by using the timer.update_timing() from OpenTimer's API.

Algorithm 4 compute_timing

```
procedure COMPUTE_TIMING(user_input, num)
    timer.update_timing();
    if user_input is report_timing then
        path equals timer.report_timing(num);
        for size_t i=0; i<path.size(); ++i do
            print path[i];
        end for
    else if user_input is dump_graph then
        timer.dump_graph();
    else
        path equals timer.report_timing(1);
        print path[0];
    end if
end procedure
```

By default, the method prints the most critical path. Otherwise, it prints the top n critical paths input by the user. In both these cases, timer.report_timing is the API method used from OpenTimer for performing the action.

In general, the critical path is calculated between the primary inputs and the primary outputs. However, critical path between two given points can be calculated when these points are mentioned as the source and destination when the pass is executed. The default option would be to consider the graph's input and output nodes or in Verilog convention, the input and output of the top module, as the primary inputs and outputs.

The timing graph can be dumped for debugging, which is a very useful feature. This can be invoked by the user and in turn uses timer.dump_graph API method to execute this task.

4.2.5 Annotating Delay

Once the circuit is built, the timing information is updated and annotated back to the nodes in LGraph as described in Algorithm 5. This requires no user inputs and does not vary based on user inputs required for computing timing information. This traverses OpenTimer's internal data structure and updates the delay fields in LGraph's database.

Algorithm 5 annotate_delay

```
procedure ANNOTATE_DELAYS(LGraph *g)
    timer.update_timing();
    for const auto &nid : g→ forward() do
        cell_name ← name_of_cell
        instance_name ← name_of_cell_instance
        for const auto &edge : node.out_edges() do
            net_name ← name_of_net
            nodepin_name ← name_of_nodepin
            delay ← max(delay(nodepin_name));
        end for
    end for
end procedure
```

Similar to computing the timing information, while annotating delay back to LGraph, the timer is first updated to maintain the latest information about the circuit. This is done by using the timer.update_timing() from OpenTimer's API.

The graph is traversed using a node iterator and the cell name and instance names are collected. For every cell, the edge iterator is used to iterate the output edges of each nodes and the max of all delays from each edge is stored in the LGraph database as shown in Figure 4.2.

The database represents the basic information required for any format in the

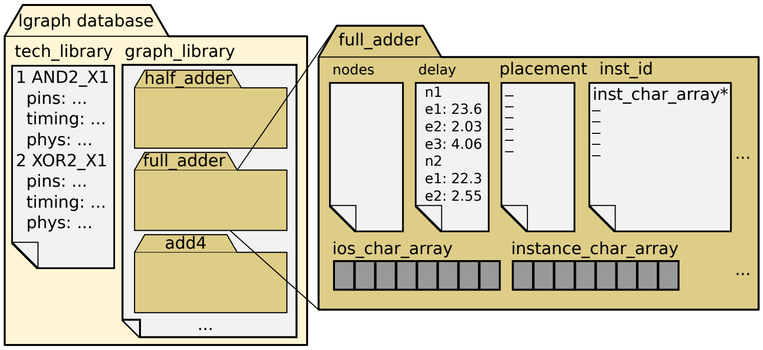


Figure 4.2: LGraph's database organization after computing timing information and storing them in its nodes.

design flow. This pass would modify the database and update it with timing related information such as the delay for each cell. This information can now be used by other timing dependent passes such as place and route etc.

4.3 Interface

The lgshell can be used to query timing related information using the OpenTimer pass supported by LGraph. This interface is explained using an example where a Verilog netlist is converted to an lgraph, represented in a DOT format and is queried using the lgshell for various timing information.

Figure 4.3 shows a Verilog netlist that contains 3 primary inputs, 1 primary output, 5 cells, and 8 nets. These cells are instantiated from a cell library and the same cell library should be provided to LGraph and OpenTimer for proper synchronization.

The Verilog netlist is converted to a lgraph using the Yosys pass. A cell library is provided while doing the same and it is done with a full techmap option to preserve

```
module sample (
inp1,
inp2,
tau2015_clk,
out
);

// Start PIs
input inp1;
input inp2;
input tau2015_clk;

// Start POs
output out;

// Start wires
wire n1;
wire n2;
wire n3;
wire n4;
wire inp1;
wire inp2;
wire tau2015_clk;
wire out;

// Start cells
NAND2X1 u1 ( .A(inp1), .B(inp2), .Y(n1) );
DFFNEGX1 f1 ( .D(n2), .CLK(tau2015_clk), .Q(n3) );
INVX1 u2 ( .A(n3), .Y(n4) );
INVX2 u3 ( .A(n4), .Y(out) );
NOR2X1 u4 ( .A(n1), .B(n3), .Y(n2) );

endmodule
```

Figure 4.3: Sample Verilog netlist to illustrate an example of the OpenTimer integration with LGraph

the cell names and its instance names in the lgraph for performing timing analysis. Figure 4.4 shows the commands to perform this operation.

The tech-mapped version of the DOT format of the Verilog netlist described in Figure 4.3 is depicted in Figure 4.5. There are different types of nodes such as graphio, blackbox, strconst.

```
lgraph> inou.yosys.tolg  files:sample.v techmap:full  lib:sample.lib
lgraph> lgraph.open name: sample |> inou.graphviz
```

Figure 4.4: lgshell commands to convert the example Verilog netlist to an lgraph and represent it in DOT format

Lgraph preserves the tech-mapped cells in terms of a blackbox that retains information such as the name of the cell, name of the instance, name of the nodepin, name of the net connected to the nodepin.

The graphio's are of two types input graphio and output graphio. These describe the primary input and the primary output of the circuit which are the entry and exit points for timing analysis.

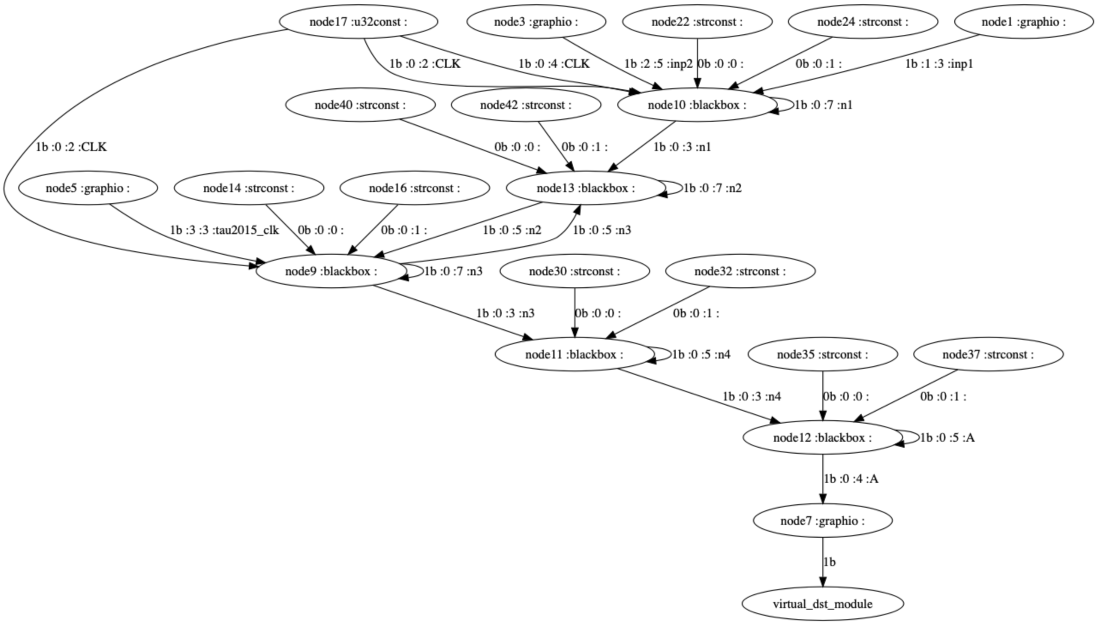


Figure 4.5: Graphviz representation of the lgraph converted from the sample Verilog netlist

The default option while using the OpenTimer pass when provided with a lgraph, a liberty file and a SDC file is to report the critical path. Figure 4.6 describes the commands required to convert a Verilog netlist to a tech-mapped lgraph and later make the OpenTimer pass report the critical path.

```
inou.yosys.tolg files:sample.v techmap:full lib:sample.lib
lgraph.open name: sample |> pass.opentimer lib:sample.lib sdc:sample.sdc

Startpoint    : inp1
Endpoint      : f1:D
Analysis type : min
-------------------------------------------------------
       Type      Delay       Time   Dir  Description
-------------------------------------------------------
       port      0.000      0.000  fall  inp1
        pin      0.000      0.000  fall  u1:A (NAND2X1)
        pin      2.786      2.786  rise  u1:Y (NAND2X1)
        pin      0.000      2.786  rise  u4:A (NOR2X1)
        pin      0.181      2.967  fall  u4:Y (NOR2X1)
        pin      0.000      2.967  fall  f1:D (DFFNEGX1)
    arrival                 2.967        data arrival time

related pin     25.000     25.000  fall  f1:CLK (DFFNEGX1)
 constraint      1.518     26.518        library hold_falling
   required                26.518        data required time
-------------------------------------------------------
      slack               -23.551        VIOLATED
```

Figure 4.6: Critical path reported by LGraph for the sample Verilog netlist

Another option while using the OpenTimer pass is to report circuit information. Figure 4.7 describes the commands required to convert a Verilog netlist to a tech-mapped lgraph and later make the OpenTimer pass display important circuit information to the user.

Figure 4.8 describes the commands required to display the timing graph in a DOT format. Tools such as Graphviz can be used to view this DOT file as mentioned

```
lgraph> inou.yosys.tolg files:sample.v techmap:full liberty:sample.lib
process_module \sample
lgraph.open name:sample |> pass.opentimer lib:osu018_stdcells.lib
Number of gates 5
Number of primary inputs 3
Number of primary outputs 1
Number of pins 17
Number of nets 8
```

Figure 4.7: Circuit information reported by LGraph for the sample Verilog netlist

in Figure 4.9. This is a very convenient feature as it allows the user to understand the critical path in a much more intuitive manner.

```
inou.yosys.tolg files:sample.v techmap:full lib:sample.lib
lgraph.open name: sample |> pass.opentimer lib:sample.lib
```

Figure 4.8: lgshell commands to represent the timing graph of the sample Verilog netlist in DOT format

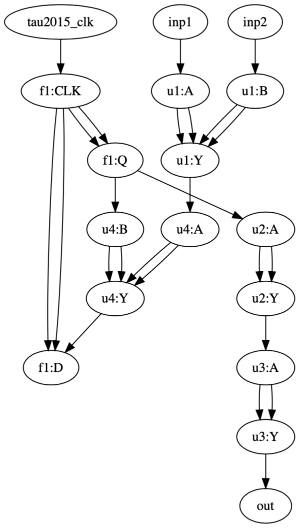


Figure 4.9: Graphviz representation of the sample Verilog netlist's timing graph

4.4 Conclusion

In this chapter, the key features of the integration were discussed along with the algorithms used in its implementation. An example of the integration is illustrated using the lgshell interface.

This integration required no modification of the OpenTimer source code, and it only added around 500 lines of code to the LGraph code base. However, this was achieved in a separate pass without modifying the LGraph's source code. The newly added pass also has test files in OpenTimer's pass directory to inspect the functionality when the pass's source code is modified to adapt future implementation.

Some of the features of the integration include its ability to read files required for OpenTimer, parse SDC files, build equivalent circuits from lgraphs, compute timing information to find its critical path and annotate cell delays on the lgraph based on it. This is also achieved with negligible overhead in performance.

Future work includes integrating other open source timing analysis tools such as OpenSTA and creating lgtiming - a pass that would allow the user to select the required timing tool to compute timing information.

Chapter 5

Conclusion and Future Work

> When you reach the end of what you
> should know, you will be at the
> beginning of what you should sense.
>
> Kahlil Gibran

In this thesis, the internals and interface of LGraph was initially explained with the help of its lgshell. It was shown how LGraph functions as a LLVM for hardware designs.

Next, OpenTimer was introduced and its internals and interface was explained with the help of its ot-shell. It was shown why OpenTimer was chosen as the preferred tool for STA integration with LGraph.

Finally, the integration of OpenTimer and LGraph was described and its internals and interface were explained using an example. Some of the ideas that evolved over this phase were the development of a custom STA tool tailor made for LGraph's purposes.

OpenTimer is actively maintained and its future inclusions such as hierarchical support for Verilog files must be maintained by the pass. There are also other alternatives for timing analysis such as OpenSTA. These tools should be integrated into LGraph. A unified pass should be made to accommodate these tools under a single umbrella called lgtiming.

All the timing analysis tools have common input file format. A pass can be created to load the input files to save them in LGraph's database. This storage information can be saved as json formats and fed as inputs to these timing analysis tools.

When the SDC file is parsed, it is very important to differentiate the regular inputs from the clock. This might be as trivial as recognizing it from the name of the clock or as complicated as differentiating between the clock and the clock enable in the case of a clock gated circuit. It is interesting to have a pass, *mark the clocks* that performs this task.

This creates a platform to improve productivity in performing timing analysis and when aligned with LGraph's future goals, leads to new venues such as obtaining live timing feedback as and when the RTL is written.

Bibliography

[1] Cadence Design Systems Inc. Tempus timing signoff solution. `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/silicon-signoff/tempus-timing-signoff-solution.html`. Online; accessed on 26 April 2019.

[2] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. Rsyn: An extensible physical synthesis framework. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, pages 33–40, New York, NY, USA, 2017. ACM.

[3] Tiago Fontana, Renan Netto, Vinicius Livramento, Chrystian Guth, Sheiny Almeida, Laércio Pilla, and José Luís Güntzel. How game engines can inspire eda tools development: A use case for an open-source physical design library. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, pages 25–31, New York, NY, USA, 2017. ACM.

[4] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *Computer-Aided Design, Proceedings of the IEEE/ACM Interna-*

tional Conference on, ICCAD'15, pages 895–902, Piscataway, NJ, USA, Nov. 2015. IEEE Press.

[5] James Cherry, William Scott. Opensta. `https://github.com/abk-openroad/OpenSTA`. Online; accessed on 26 April 2019.

[6] Pei-Yu Lee, Iris H. R. Jiang, Cheng R. Li, Wei-Lun L. Chiu, and Yu-Ming Yang. iTimerC 2.0: Fast incremental timing and CPPR analysis. In *Computer-Aided Design, Proceedings of the IEEE/ACM International Conference on*, ICCAD'15, pages 890–894, Piscataway, NJ, USA, Nov. 2015. IEEE Press.

[7] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[8] C. Peddawad, A. Goel, , and N. Chandrachoodan. iitrace: A memory efficient engine for fast incremental timing analysis and clock pessimism removal. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 903–909, Nov 2015.

[9] Rafael T. Possignolo and Jose Renau. LiveSynth: towards an interactive synthesis flow. Poster at the 28th HotChips: A Symposium on High Performance Chips. Available at `https://users.soe.ucsc.edu/~rafaeltp/files/livesynth-hotchips2016.pdf`.

[10] Rafael T. Possignolo and Jose Renau. SMatch: Structural matching for fast resyn-

thesis in fpgas. In *Design Automation Conference, Proceedings of the 56th*, DAC'19, New York, NY, USA, Jun. 2019. ACM.

[11] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau. Lgraph: A multilanguage open-source database. In *Open-Source EDA Technology, Proceedings of the First Workshop on*, WOSET'18, Oct. 2018.

[12] Richard Goering. Synopsys milkyway: Interoperability step. `https://www.eetimes.com/document.asp?doc_id=1201777#`. Online; accessed on 26 April 2019.

[13] Synopsys, Inc. Primetime static timing analysis. `https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html`. Online; accessed on 26 April 2019.

[14] Synopsys, Inc. Technology access program (tap-in). `https://www.synopsys.com/community/interoperability-programs/tap-in.html`. Online; accessed on 13 May 2019.

[15] Clifford Wolf. Yosys open SYnthesis suite. `http://www.clifford.at/yosys/`, 2016. Online; accessed on 8 November 2018.