LGraph: A multi-language open-source database for VLSI

Rafael Trapani Possignolo, Sheng Hong Wang, Haven Skinner, Jose Renau
Dept. of Computer Engineering,
University of California Santa Cruz.
http://masc.cse.ucsc.edu

ABSTRACT

We present *LGraph*, an open-source database for digital circuits in different phases of the synthesis and physical design flow. *LGraph* is a bi-directional graph and uses memory maps for fast persistence to disk. It is meant to be a convergence point between open-source EDA tools, which will improve the integration of research in different areas. It is already integrated with Yosys, ABC and OpenTimer and includes parsing for LEF/DEF and Liberty formats. It can also read Pyrope, a modern HDL. Extensions will include a placement and a routing tools as well as Chisel integration.

1. INTRODUCTION

Currently, there are many formats to represent a digital design in the different phases of the flow. However, there is no open-source option that can represent the design in different steps of the flow. The alternative is to use different formats throughout the design flow, for instance, Verilog and BLIF during logic synthesis, LEF/DEF or bookshelf during physical design and GDS for layout and parasitics. There have been some attempts to create a more concise framework for a wider set of applications, for instance the OpenDesign Flow Database [6].

Other efforts focused on specific points of the flow. FIRRTL [9] and RTLIL [14] are two open-source formats that target RTL and netlist. Whereas, Rsyn [3] and Ophidian [4] provide an extensible framework for physical design. Also, the formats are usually task specific and not ideal for integration.

Academic EDA research and contests, *e.g.*, ISPD, TAU, and ICCAD, focused on isolated steps of the design flow. While there have been useful advances in various areas of EDA, a need for an integration has been identified [7]. In [7], the author proposed a horizontal benchmark extension methodology, which not simply converts data format between benchmarks, but also reorganizes it by either filling missing parts or simplifying redundancy for different tools.

In the commercial world, OpenAccess (OA),[1] is an "open" format meant to provide interoperability among IC design tools. OA has many legal constraints that limits its usage. The other commercial option is MilkyWay from Synopsys^{TM}; however, since this is a proprietary format from Synopsys, not much can be said about it.

[1] http://projects.si2.org/?page=69.

We present Live Graph (*LGraph*), a graph database, that works as a bridge between different parts of the design flow. It can represent RTL, any netlist, or a placed and routed design. *LGraph* interface with several input languages, like verilog, Liberty, LEF/DEF, and Pyrope [12]. *LGraph* interfaces directly with ABC for technology mapping and synthesis [2], and OpenTimer [5] for static timing analysis. Placement and Routing tools are currently being developed and will work directly in *LGraph*. *LGraph* is based on memory maps for fast persistence – similar to mmap [10] – and is based on the struct of array paradigms to increase memory locality, for instance, when performing static timing analysis, only the timing related tables are brought to cache. *LGraph* is inspired by the *LiveSynth* mindset and aim for incremental synthesis results in a few seconds [11].

Besides synthesis, *LGraph* is actively being developed to support simulation of synthesizable HDLs. There is a focus to have fast code generation, debugging support, and a framework similar to LLVM but focused on the much simpler subset of synthesizable HDLs.

Multiple communities will benefit from *LGraph*. Developers of new HDLs can simply map to *LGraph* and leverage the existing back-end infrastructure. Physical design groups can *LGraph* to provide support for different languages and to evaluate integration with other steps, moving beyond simple benchmarks regularly used for specific steps in physical design. RTL designers will be able to use the integrated open-source flow, which provides a very low entry level barrier, instead of spending time integrating tools from different domains. We see *LGraph* as the LLVM in hardware design since it provides a converging point for both language developers and back-end engineers.

Our results show that *LGraph* is fast being able to traverse netlists with millions of nodes in about $0.01s$ when ordering is not required, and in $0.5s$ traversing from inputs to outputs, which is comparable to the best academic implementation. *LGraph* size is 70% to 90% larger than Verilog, which does not include physical information, and comparable with DEF. One of the main advantages of *LGraph* is to serve as an integration point for open-source projects in different areas of EDA.

2. LGRAPH

LGraph is a graph representation optimized to represent netlists during different phases of the synthesis and

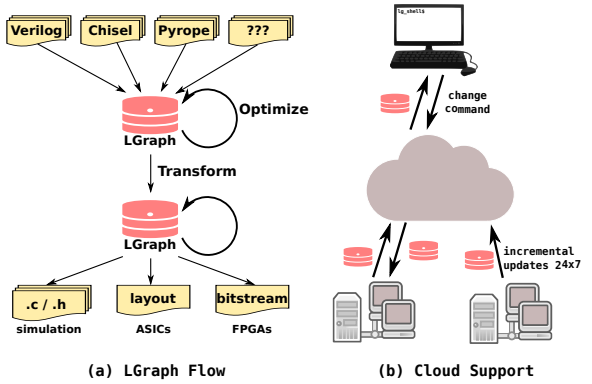

Figure 1: The overall flow starts with RTL generating a *LGraph* that can be synthesized, placed, routed, timed, so forth. *LGraph* also offers support for the cloud, automatically passing databases to servers and collecting the resulting *LGraph*.

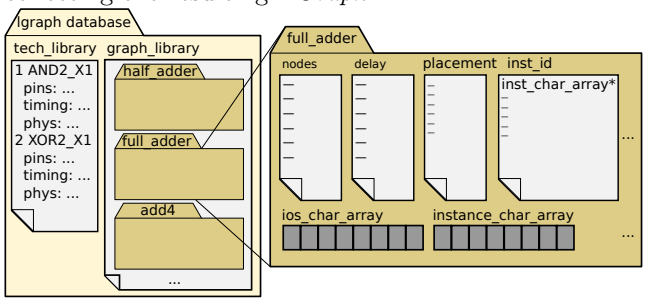

Figure 2: The database is a collection of *LGraphs* that represent modules and a technology file with standard cell information. Each *LGraph* is a collection of tables with information from different steps of the flow.

physical implementation. It is implemented in C++14 and exposed to users as an API to manipulate the data structure or as an extendable toolset with a terminal.

2.1 Overview and Expected Usage

LGraph is intended to be used from RTL to layout, as steps are being performed, the database is modified, either in-place or through the generation of a new instance (Figure 1). *LGraph* holds information both on the design itself and on the standard cell library.

The regular flow consists of generating an *LGraph* from an RTL description, such as Verilog. Transformation passes, like dead code elimination and common subexpression elimination, can be performed directly on *LGraph* to optimize the design. To leverage the large existing open-source EDA code base, *LGraph* can also interface with other tools. Transformations can be applied in parallel leveraging the servers on the cloud. Our framework allows for easily extending or replacing any of the existing algorithms or tools so that researchers can leverage the existing infrastructure for other steps of the flow. Also, the integrated approach eases the implementation of co-optimization that spans separate design steps.

2.2 LGraph Database organization

As a memory map, the *LGraph* database gets automatically persisted to disk when the program exits. Conceptually, the database contains a target technology (for mapped designs) and a set of modules (*LGraphs*) that represent the design itself. Information corresponding to the standard cell library is not duplicated in the graph nodes; instead, each node has a type that points to a specific cell in the library.

A *LGraph* represents a single module and consists of a collection of tables, indexed by node ID. The overview of the database organization is provided in Figure 2 (not all tables represented). To prevent adding arbitrarily sized strings in the tables, strings are stored in a separate char array that provides a unique ID. Note the case of instance names ("inst_name"). Names are stored in the char array "instance_name_char_array", while the table "inst_name" stores pointers to specific strings in the array.

Users can define custom tables as they identify the need for new applications. However, *LGraph* provides a set of standard tables that should be used to keep compatibility across users. Those tables cover the basic functionalities of VLSI netlists and are enough to represent a netlist at any point during the design time.

2.3 Graph Representation

Internally, *LGraph* uses a bi-directional adjacency list representation [13], more efficient to represent sparse graphs. The main drawbacks of an adjacency list are the difficulty of getting reverse edges. Thus, the decision for a bi-directional adjacency list graph is still more memory efficient for VLSI netlist than an adjacency matrix and allow both forward and backward traversals efficiently.

Each netlist gate can be represented by multiple graph nodes.[2] Each gate is represented in *LGraph* by an unique node ID. Node IDs are sequentially assigned, at node creation, starting from 1. Each port in a gate is identified by a port ID, PID for short. An overview of the *LGraph* representation with the adjacency list and edges is shown in Figure 3. The graph in the left of the figure is represented by the *LGraph* in the right.

For cache locality efficiency, nodes are preallocated 64 bytes to be aligned with a L1 cache line. If more space is needed – *e.g.*, due to a large number of output edges – extra space is allocated. For compactness, most edges use relative indexes. Since most edges will point to relatively close indexes, only a small number of bits is needed to represent them.

2.4 LGraph Types

In *LGraph*, a type is represented as a 64bit unsigned integer in the form of an enumeration, divided into ranges. Each range serves a specific category of node type. The first range of few dozens is reserved for "native" or primitive *LGraph* types. Those represent control flow, unmapped logic and arithmetic operators, and wire operators. Then a range of 2^{32} can hold subgraph types, which are used to represent the design hierarchy. The third and fourth ranges are used for constants, 32 bit (third) and arbitrarily sized (stored as char array IDs in the fourth range). The last range is used for

[2]For the sake of clarity, throughout this document, *node* refers to a graph node, *gate* refers to a netlist gate. In *LGraph* a gate can be represented by multiple nodes.

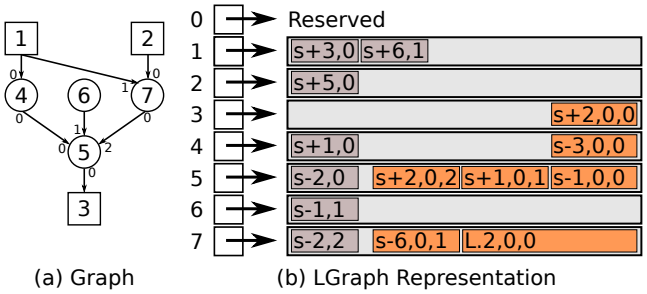

Figure 3: *LGraph* uses an adjacency list to efficiently represent sparse graphs. The graph (left) is represented as the *LGraph* (right). In the left, the little numbers outside of nodes represent port IDs, omitted for gates with a single port ID. Short edges contain relative indexes ($\pm delta$), and long edges have absolute indexes.

standard cell technology mapping, where each cell is also assigned a unique ID.

One particularity in the types representation is the use of single port per function in commutative operations, *e.g.*, an *and* gate contains a single input port, since the order of the operation does not matter. Also, it is possible to connect any number of operators in each port. Thus it is possible, for instance, to create a 3 input *and* gate using the same operator as one would use for a 2 input *and* gate. Arithmetic operators like *plus*, *mul*, *lt*, *gt*, and others also have specific ports for signed and unsigned inputs. In the specific case of *plus*, there are also ports for *minus* operation. The overall goal is to reduce code complexity when performing transformation, since several transformations are insensitive to whether a number is signed or not and can handle fewer cases with the reduced number of operators.

LGraph defines special operators for wire manipulation. *Join* operators are used to concatenate wires and *Pick* operators are used to select ranges within a multi-bit wire. For *Pick*, we leverage the information on the bit width of the node and rely on a constant that tells the offset from which we start the range.

2.5 Routing Representation

For routing, *LGraph* uses a shape based representation. Overall, each pair of nodes can be connected by one out of a few basic shapes. To generate more complex routing shapes, extra nodes can be inserted. Each basic shape connects a set of nodes, from a source to at least one sink, whose placements (x,y, and layer) are known. We anticipate that extra basic routing shapes will be added in the future, but currently, *LGraph* supports straights, L and T shape. There is a trade-off between extra shapes and extra nodes that requires further study to decide on extra shapes. Metal layers and vias are based on metal layer information from the nodes.

2.6 Open Projects

The *LGraph* code base is in active development and available on GitHub[3]. However, there are still some open ideas to improve adoption by the community.

[3] `https://github.org/mascucsc/lgraph`.

Some ongoing work in our group includes the implementation of a placement and routing tool for ASIC. The implementation of a custom timer is also on-going and will run on *LGraph* instead of requiring to first export to OpenTimer. There is also an ongoing effort to integrate a SAT solver into *LGraph* to facilitate synthesis transformations and verification in *LGraph* itself.

Support for multiple emerging languages is being added, for instance there is currently work in integrating Pyrope [12], which is leveraging *LGraph* as an API to perform control flow graph analysis and data flow graph generation and optimization. Chisel [1] support is also among the planned extensions for *LGraph*.

For simulation, the team has prototypes using C++ targets and LLVM [8]. The main advantage would be the generation of simulation binaries. The code and infrastructure generated allow to read from several languages and generate a single simulation as long as the HDLs are fully synthesizable.

3. EVALUATION

3.1 Setup

We compared *LGraph* with Yosys (version 0.7+483) [14] and RSyn (commit 02d79e4) [3] for reading and writing from disk and traversal time. For sizes, we compare *LGraph* with Verilog, RTLIL (Yosys representation) and DEF. We used the ICCAD15 SuperBlue benchmarks.[4] The benchmarks range in size from 770k to 1.9M gates.

We implemented two simple algorithms: 1) histogram of cell types and 2) find maximum combinational depth. Although basic operations, those closely mimic area estimation and timing estimation. Calculating the histogram of cell types can be done in any order, and thus allow for a faster traversal of the netlist. Finding the maximum combinational depth is done in topological order, *i.e.*, from inputs to outputs. All experiments were run on a Intel(R) Xeon(R) E3-1275 v3 core @ 3.50GHz with 32GB of memory, running Archlinux. Tools were compiled with clang v6.0.0.

3.2 Results

Figure 4 shows the runtime for both the histogram and the combinational depth algorithms. Since the passes are overall much faster than the read and write times, we performed histogram 100 times and depth 20 times. We note that *LGraph* has much smaller read and write times due to the memory mapped, which avoids the need to parse netlist files. Yosys has a slower read time since it has a parser capable of reading the full specification of Verilog. RSyn has a more basic Verilog parser meant for netlists and that is not general for any Verilog, and thus is faster. RSyn can also read DEF netlists, which is not available in Yosys. In general, *LGraph* is able to load benchmarks in about $1-2s$, RSyn in about $30-40s$ and Yosys in about $130-200s$.

The traversals have comparable runtime between RSyn and *LGraph*. An unordered traversal takes about $0.01s$ with either of the tools, in Yosys the traversal time

[4] `http://cad-contest.el.cycu.edu.tw/problem_C/default.html`

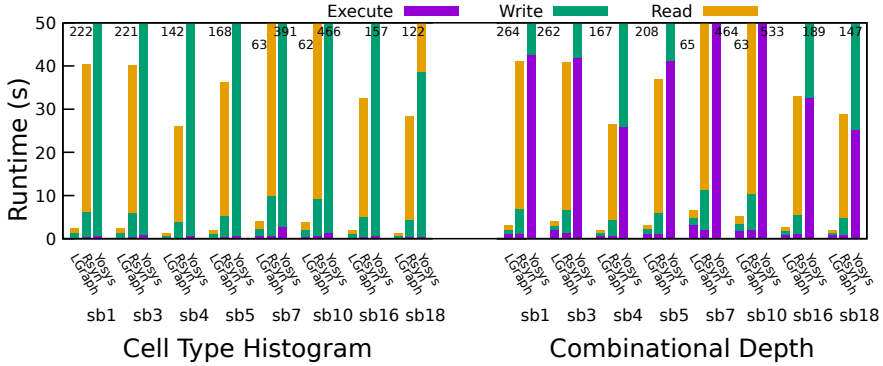

Figure 4: *LGraph* has a much smaller read and write overhead, since it uses memory maps to prevent parsing. Traversing a netlist in *LGraph* is as fast as RSyn and in faster than Yosys, in particular for ordered traversal.

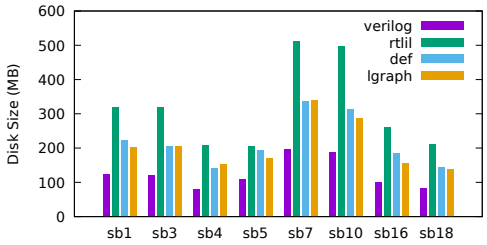

Figure 5: *LGraph* size in disk is compatible with that of DEF files. Verilog was the most efficient representation among the ones tested.

is about 50% slower, but still around $0.015s$. For ordered operations (*i.e.*, from input to output), *LGraph* and RSyn are able to traverse a large netlist in about $0.5s$, whereas a traversal in Yosys takes about $5 - 10s$ depending on the benchmark.

We also looked into the size of each representation. For that comparison, we only looked into the physical benchmarks. Verilog netlists are the smallest of the representations considered, ranging from 80 to 198MB for the benchmarks tested. RTLIL, the internal representation of Yosys was the largest representation, ranging from 205 to 500MB. *LGraph* was mostly equivalent to DEF files, both ranging from about 140 to 340MB. A summary of the sizes for the benchmarks tested is provided in Figure 5. However, Verilog and RTLIL do not include physical information.

4. CONCLUSIONS

We present *LGraph*, an open-source database for digital circuits that can represent netlists in different phases of the design flow from RTL to layout. *LGraph* is intended as a convergence point for efforts from different groups and communities, from HDL and compilers research to physical design.

Our results show that *LGraph* can traverse netlists with millions of nodes in about $0.01s$ when ordering is not required, and in about $0.5s$ traversing from inputs to outputs. This is comparable to the best academic implementation. *LGraph* is comparable in size to DEF, and about 70% to 90% larger than Verilog, which does not include physical information. Further work includes integration with place and route tools, other HDLs and extensions to support academic contests.

Acknowledgments

We like to thank all the students who have contributed to this project. This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF. This work was supported by UCSC Center for Research in Open Source Software.

5. REFERENCES

[1] J. Bachrach, H. Vo, et al, "Chisel: constructing hardware in a scala embedded language," in *DAC '12*.

[2] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV'10*.

[3] G. Flach, M. Fogaça, et al, "Rsyn: An extensible physical synthesis framework," in *ISPD '17*.

[4] T. Fontana, R. Netto, et al, "How game engines can inspire eda tools development: A use case for an open-source physical design library," in *ISPD '17*.

[5] T. W. Huang and M. D. F. Wong, "Opentimer: A high-performance timing analysis tool," in *ICCAD '15*.

[6] J. Jung, I. H. R. Jiang, et al, "Opendesign flow database: The infrastructure for vlsi design and design automation research," in *ICCAD '16*.

[7] A. B. Kahng, H. Lee, and J. Li, "Horizontal benchmark extension for improved assessment of physical cad research," in *GLSVLSI '14*.

[8] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04*.

[9] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016.

[10] Z. Lin, M. Kahng, K. M. Sabrin, et al, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *Big Data '14*.

[11] R. T. Possignolo and J. Renau, "LiveSynth: Towards an interactive synthesis flow," in *DAC'17*.

[12] H. Skinner, R. Possignolo, and J. Renau, "Liam: An actor based programming model for hdls," in *MEMOCODE'17*.

[13] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[14] C. Wolf, "Yosys open synthesis suite," "http://www.clifford.at/yosys/", 2016.