# Ophidian: an Open-Source Library for Physical Design Research and Teaching

Renan Netto, Tiago Augusto Fontana, Sheiny Fabre, Bernardo Ferrari,
Vinicius Livramento, Thiago Barbato, João Souto, Chrystian Guth, Laércio Pilla and José Luís Güntzel
Embedded Computing Lab (ECL) — Dept. of Computer Science and Statistics
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
{renan.netto, tiago.fontana, sheiny.fabre}@posgrad.ufsc.br

## ABSTRACT

There is a lack of open source physical design automation software infrastructure that could be used by researchers to try new algorithms with as little effort as possible, and by students to easily understand and experiment classical algorithms. Such infrastructure should be easy to use and highly modular to allow programmers to develop new algorithms without being required to understand details of the underlying library code. However, such modularity should come without compromising the software performance. This paper presents Ophidian: an open-source library for physical design research and teaching. Ophidian aims to provide a basic infrastructure to develop algorithms for different physical design automation steps, while providing simple implementations of classical algorithms in order to help students understand the physical design flow. In addition, Ophidian makes use of modern software engineering design patterns to allow the development of programs with short runtimes while providing high code modularity. To highlight the benefits of using Ophidian, we implemented a software prototype that executes the A* algorithm for each interconnection segment of a circuit. The experimental results showed that the prototype requires 30% shorter runtime when compared to a traditional Object-Oriented implementation.

## 1 INTRODUCTION

Developing and prototyping new physical design automation algorithms, or even reproducing an existing one is an arduous work. The programmer must not only develop the algorithm itself, but also implement the necessary parsers and data structures to handle circuit data. Therefore, it is important that students and researchers working on physical design automation could have access to open-source code to quickly and easily build the surrounding infrastructure, thus dedicating most of their time and effort to achieve higher contributions in solving the target problem itself.

There are already some initiatives towards open-source tools for different physical design steps, such as timing [8], placement [3] and routing [4, 11]. However, all those tools are devoted to specific steps of the design flow and their integration in a complete flow does not seem evident. As consequence, while it is possible to use them for research, their applicability for teaching purposes is quite restricted. The reason is that those tools implement very efficient, but sophisticated algorithms and techniques that are difficult for novice students to understand. Indeed, for didactic purposes classical algorithms are simpler and thus easier to understand.

On the other hand, Rsyn is an open-source library that aims to help the research on physical design algorithms [6]. Rsyn provides a basic infrastructure to develop algorithms for different physical design automation steps, and supports the benchmarks of recent contests, such as the ICCAD 2015 [10] and ISPD 2018 CAD Contests [12]. While Rsyn provides a generic infrastructure to develop physical design algorithms, it is focused on research and it is still under development.

The development of an open-source library imposes some restrictions that may not be necessary when developing stand alone tools and/or algorithms. Since a library is designed to be used by many different people, it is essential that it is highly modular, and can be easily extended [13]. In addition, the library core functions must have good performance, so that their runtime does not impact on the whole application runtime.

Therefore, in this paper we present Ophidian: an Open-Source Library for Physical Design Research and Teaching [7]. Ophidian is based on modern software engineering design patterns and open-source tools in order to provide a modular and easy to use library for physical design research and teaching. The features of Ophidian include:

- Support to industry file formats, such as DEF, LEF and Verilog.
- Support to benchmarks of recent contests, such as ICCAD 2015 and 2017 CAD Contests, as well as ISPD 2018 CAD Contest.
- Usage of modern software engineering design patterns, such as Data-Oriented Design (DOD) and Entity-Component System. Section 3 presents an overview of such design patterns.
- Usage of data structures that provide higher cache locality for the program. As consequence, applications developed using Ophidian may achieve shorter runtime, as presented in Section 4.
- Usage of unit testing and continuous integration [5], to improve maintainability and ensure that it builds and runs on modern operating systems.
- Ophidian is fully open-source, with the source code available on Gitlab [1].

The remaining sections of this paper are organized as follows: Section 2 presents an overview of the Ophidian library, showing an example of how to implement an application using Ophidian.

Section 3 shows how Ophidian makes use of modern software engineering design patterns. Section 4 shows an use case of a physical design application to illustrate how the usage of modern design patterns may improve the performance of a program, while Section 5 draws the conclusions and presents the next development steps of Ophidian.

## 2 LIBRARY OVERVIEW

Figure 1 shows an overview of the Ophidian library. The library is divided into modules, where each module encapsulates different information from the circuit. For example, the netlist module encapsulates the cells, pins and nets information. Given the netlist class, the user can know which pins belong to each cell, and which nets are connected to those pins. The placement module, in its turn, encapsulates the information on cell geometries and their locations. This way, developers can use only the modules that are necessary for the application they are building. However, instantiating several classes (one for each module) may also be a nuisance to the developer. To circumvent this difficulty, Ophidian encapsulates all those classes in a single Design class, which acts like a shell to all the other classes. This way, developers only need to instantiate a single class (Design), and can make use of the necessary modules.

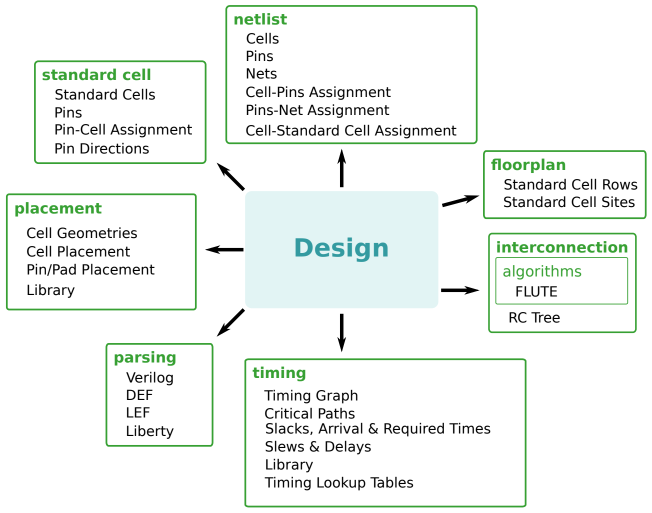

**Figure 1: Overview of the current modules of Ophidian.**

The Design class should be able to handle different benchmarks while keeping this transparent to the developer. Ophidian does this through the factory design pattern [16]. This design pattern consists on a generic class (i.e. Design class) which can be built through different ways. Then, a factory class/module provides a set of functions to build this generic class. In the case of Ophidian, the library provides functions to build the Design given different benchmarks. For example, the developer can build a Design from .DEF, .LEF and .Verilog files (as it was the case of the ICCAD 2015 Contest), or only from .DEF and .LEF files (as in the case of the ICCAD 2017 Contest). In the end, the developer has access to the Design object without being required to know how to build it.

Finally, there are two ways of using Ophidian: developers can use it only as an external library to build their own algorithms, or they can change Ophidian code themselves, since the library is fully open-source. For those that wish to use it only as an external library, Ophidian provides a project template in a separate repository [2]. This project template shows how to use the library inside another application, which is useful for teaching. For instance, in a physical design automation course students would be able to develop a single algorithm without being required to understand the library implementation: they only need understand how to use the library interface.

## 3 IMPLEMENTATION OVERVIEW

In order to provide the modularity described in Section 2 without compromising the program performance, we make use of the data-oriented design (DOD) programming model. Unlike object-oriented design (OOD), which focuses on how to represent the problem with objects, DOD focuses on how data is organized in memory. To illustrate the difference between those two programming models, suppose we want to build a program to solve a given physical design problem which requires modeling nets and pins. Figure 2 shows a possible class hierarchy using OOD for such purpose. In this hierarchy, the program will have a set of Net objects, where each object has a name and a reference to all its pins. Each Pin is also an object by itself, with a name and a reference to the net it belongs to. If it is necessary to know the pin position as well, there is a specialization of the Pin class that includes a position attribute.

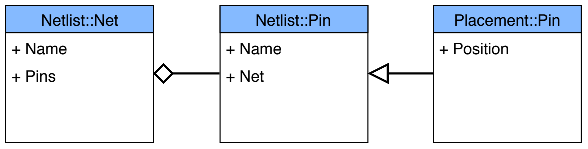

**Figure 2: Class diagram representing a physical design problem with OOD approach. The diamond-end arrow between *pin* and *net* classes represents an aggregation relationship between two classes. The triangle-end arrow between the two *pin* classes represents a specialization relationship.**

Notice that, while this class hierarchy is very simple, real world applications may require significantly more complex class hierarchies, which hampers the development of efficient programs. One way of avoiding overly complex class hierarchies is making use of the DOD programming model, as depicted by Figure 3. In this programming model the Net objects are replaced by an array of Net indices. Those indices are then used to access the attributes of the nets, which are also stored in independent arrays. For example, in Figure 3 there is one array for the nets, and two arrays for their attributes (names and pins). The same applies for the Pins. Therefore, if the developer needs to add another attribute to a given class she/he need not create a new class specialization. It is only necessary to add an extra array with this attribute, thus avoiding complex hierarchies. In addition to the higher modularity, the DOD programming model also improves the program cache locality, since under in this model data is stored in a contiguous array. A more detailed discussion on how the DOD programming model exploits cache locality to reduce memory access and speed up programs are found in [7].

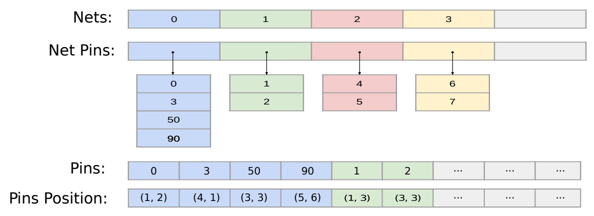

**Figure 3: Representation of a physical design problem with the DOD approach. The lines represent arrays to describe nets and pins properties.**

However, using DOD programming model may not be trivial for the developer. That is why we use the entity-component system to provide an easy-to-use DOD infrastructure inside Ophidian. This design pattern consists in decomposing a problem into sets of entities and components (also called properties). The entities are similar to the objects in OOD, except that each entity is simply a unique identifier. Then, this identifier is used to access all its properties, which are stored in separate arrays.

The mere availability of entities and properties does not simplify the development process itself. Therefore, Ophidian provides an entity system which is capable of creating and removing entities, as well as retrieving their properties. All those operations have constant time complexity and hence do not have a large impact on the program performance. As consequence, the developer can use those entities in a similar way that she/he would do it in an OOD program, except that the entities would be stored in a DOD fashion.

## 4 EVALUATION ON A USE CASE

The DOD programming model is one of the most relevant features of Ophidian. In order to evaluate its impact on program performance, we evaluated the library on the implementation of the A* algorithm [14] for global routing [9], which is a typical graph algorithm. Since a circuit may be modeled as a set of cells, pins and interconnections, many physical design problems may be solved using graph algorithms. Therefore, the results obtained on this case study may be similar on other physical design problems. In order to find the global routing of each circuit interconnection, each interconnect belonging to the circuit was decomposed into a steiner tree. Then, the A* algorithm was used to map each steiner tree segment to a set of G-cells of the circuit.

We generated experimental results for the 8 circuits available from the ICCAD 2015 Contest (problem C: Incremental Timing-Driven Placement) [10]. These circuits were derived from industrial designs having from 768k to 1.93M cells. We performed all experiments in a Linux workstation with an Intel® Core® i5-4460 CPU running at 3.20 GHz, 32GB RAM ($4 \times$ 8GB DDR3 at 1600MHz), and three levels of cache. All results presented in this section represent the average of 30 executions to ensure a small confidence interval.

Before presenting the results, it is important to explain the differences between the OOD and DOD implementations of the A* algorithm. Since we are using this algorithm for global routing, there is a need to model nets and pins in both programs. For that purpose, we used the same structure presented in Section 3 for modeling nets and pins. In addition, since this is a graph algorithm, we also need to model the graph itself. Figures 4 and 5 show how we modeled the graph using OOD and DOD, respectively. When using OOD, each node is represented by an object, which contains four attributes: *CameFrom*, *G_score*, *F_score* and *Edges*. All those attributes are used by the A* algorithm to identify the shortest path between two nodes in the graph. Each edge is also an object, which contains references to the source and target nodes, as well as a capacity and weight, used by the A* algorithm. On the other hand, when using DOD the nodes and edges are represented only by their indices, as in Figure 5. Their attributes are the same as in the OOD version, but they are all stored in separate arrays, which are indexed using the node and edge indices.

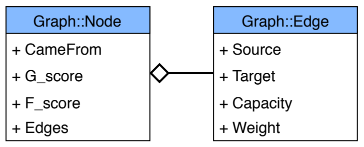

**Figure 4: Class diagram for graph representation using OOD programming model.**

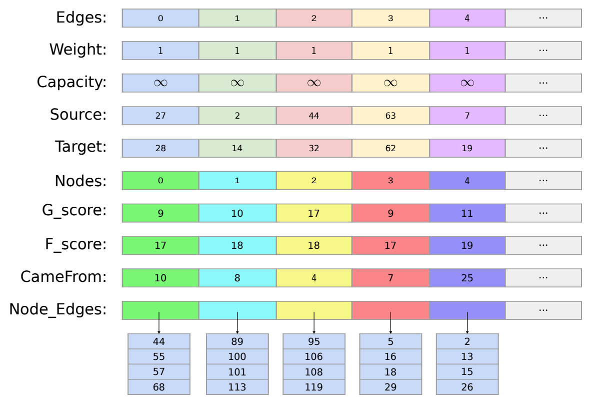

**Figure 5: Graph representation using DOD programming model.**

Figure 6 shows the number of cache misses (y axis) of each implementation of the A* algorithm for each circuit (x axis, sorted in ascending order of number of cells). We measured the number of cache misses of all levels of cache (both instructions and data caches) using the PAPI tool [15]. The OOD implementation resulted in 1.33 billion of cache misses, on average, while the DOD implementation resulted in 0.62 billion of cache misses, on average (almost 53% less cache misses than OOD). This reduction on number of cache misses relies largely on the DOD implementation of the graph data structure, since the graph is traversed multiple times in order to route all the circuit nets. In addition, observe that the circuit with highest number of cache misses was *superblue*10, which is not the circuit with highest number of cells. This happens because *superblue*10 is the circuit with the highest area and consequently, more nodes in the graph.

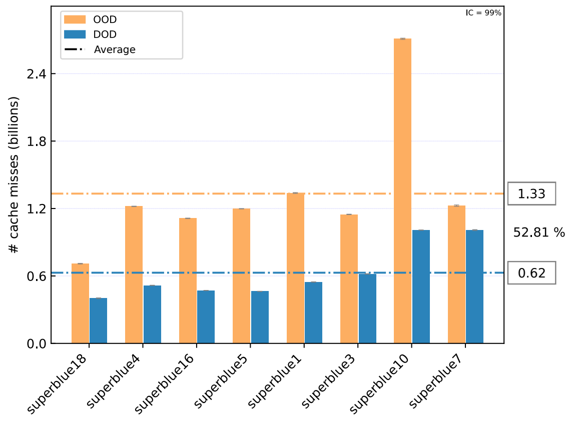

**Figure 6: Number of cache misses (y axis) of A* for each circuit evaluated (x axis). The orange bars represent cache misses for OOD implementation, the blue bars show cache misses for DOD implementation.**

The lower number of cache misses achieved by the DOD implementation directly impacts on its runtime as well. Figure 7 shows the runtime results of each implementation. As it can be seen, the OOD implementation took on average 20s to route all net segments, while the DOD required only 14s on average to do so (30% faster than OOD). Such results show that the lower number of cache misses achieved by using DOD successfully reduces the runtime of the application. This improvement, combined with the entity-component system presented in Section 3, allows Ophidian to provide a modular and fast library to develop physical design applications.

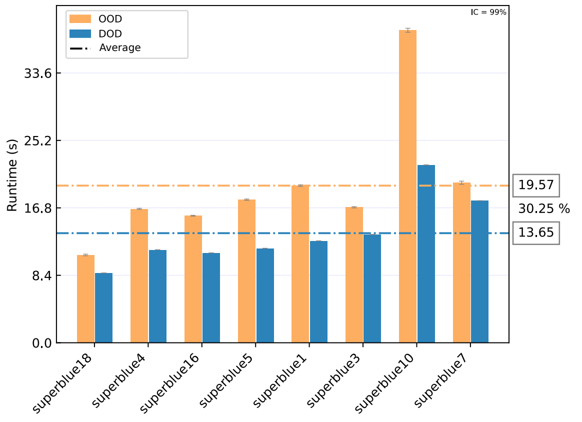

**Figure 7: Runtime (y axis) in seconds of A* for each circuit evaluated (x axis).**

## 5 CONCLUSION AND NEXT STEPS

We presented Ophidian, an open-source physical design library for research and teaching. The library aims to provide a modular interface to develop both state-of-the-art and classical physical design algorithms, through separate modules for each physical design step and support to benchmarks of recent physical design competitions.

In addition, Ophidian provides modularity while exploring the inherent cache locality of some physical design problems, through the use of DOD and the entity-component system design pattern. As consequence, programs developed using Ophidian can achieve a lower number of cache misses which helps improve performance.

Currently, Ophidian contains modules for handling standard cells, floorplan, netlist, placement and interconnections. As next steps, we intend to support other physical design steps, such as clock tree synthesis and routing, including traditional algorithms for each physical design step. This way, it is easier for researchers to develop algorithms for a single step, while they can rely on Ophidian infrastructure to implement the remaining steps. In addition, we intend to provide a script interface, so that the library can be easily used along other physical design tools.

## REFERENCES

[1] Embedded Computing Lab, Federal University of Santa Catarina, “Ophidian: an Open Source Library for Physical Design Research and Teaching”. https://gitlab.com/eclufsc/eda/ophidian. (2018).

[2] Embedded Computing Lab, Federal University of Santa Catarina, “Ophidian project template”. https://gitlab.com/eclufsc/eda/ophidian_project_template. (2018).

[3] Chung-Kuan Cheng, Andrew B Kahng, Ilgweon Kang, and Lutong Wang. RePlAce: Advancing Solution Quality and Routability Validation in Global Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[4] Ke-Ren Dai, Wen-Hao Liu, and Yih-Lang Li. NCTU-GR: Efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-D global routing. *IEEE Transactions on very large scale integration (VLSI) systems* 20, 3 (2012), 459–472.

[5] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk.* Pearson Education.

[6] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. 2017. Rsyn: An Extensible Physical Synthesis Framework. In *Proceedings of the 2017 ACM on International Symposium on Physical Design (ISPD ’17)*. ACM, New York, NY, USA, 33–40.

[7] Tiago Fontana, Renan Netto, Vinicius Livramento, Chrystian Guth, Sheiny Almeida, Laércio Pilla, and José Luís Güntzel. 2017. 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*. ACM, 25–31.

[8] Tsung-Wei Huang and Martin DF Wong. 2015. Opentimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 895–902.

[9] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. 2011. *VLSI physical design: from graph partitioning to timing closure.* Springer Science & Business Media.

[10] M. Kim, J. Hu, J. Li, and N. Viswanathan. 2015. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In *ICCAD*. 921–926.

[11] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing. *IEEE Transactions on computer-aided design of integrated circuits and systems* 32, 5 (2013), 709–722.

[12] Stefanus Mantik, Gracieli Posser, Wing-Kai Chow, Yixiao Ding, and Wen-Hao Liu. 2018. ISPD 2018 Initial Detailed Routing Contest and Benchmarks. In *Proceedings of the 2018 International Symposium on Physical Design*. ACM, 140–143.

[13] Tim O’Reilly. Lessons from open-source software development. *Commun. ACM* 42, 4 (1999), 32–37.

[14] Stuart J Russell and Peter Norvig. 2009. *Artificial intelligence: a modern approach.* Pearson Education.

[15] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[16] Pree Wolfgang. Design patterns for object-oriented software development. *Reading Mass* 15 (1994).