# How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library

Tiago Fontana, Renan Netto, Vinicius Livramento, Chrystian Guth,
Sheiny Almeida, Laércio Pilla, José Luís Güntzel
Embedded Computing Lab, Federal University of Santa Catarina, Brazil
{tiago.fontana, renan.netto}@posgrad.ufsc.br

## ABSTRACT

Similarly to game engines, physical design tools must handle huge amounts of data. Although the game industry has been employing modern software development concepts such as data-oriented design, most physical design tools still relies on object-oriented design. Differently from object-oriented design, data-oriented design focuses on how data is organized in memory and can be used to solve typical object-oriented design problems. However, its adoption is not trivial because most software developers are used to think about objects' relationships rather than data organization. The entity-component design pattern can be used as an efficient alternative. It consists in decomposing a problem into a set of entities and their components (properties). This paper discusses the main data-oriented design concepts, how they improve software quality and how they can be used in the context of physical design problems. In order to evaluate this programming model, we implemented an entity-component system using the open-source library Ophidian. Experimental results for two physical design tasks show that data-oriented design is much faster than object-oriented design for problems with good data locality, while been only sightly slower for other kinds of problems.

## 1. INTRODUCTION

Modern game engines must efficiently handle huge amounts of data to render 3D graphics for very-high resolution images, model realistic physical systems, and also process complex artificial intelligence systems. To fulfill such requirements, several concepts and design patterns are applied during the game development to take advantage of modern computer architectures, where the memory represents the main bottleneck. One of the most important concepts employed during the development of game engines is the so-called data-oriented design (DOD). Unlike the traditional object-oriented design (OOD), which focuses on how objects represent problem entities, DOD focuses on how data will be organized in memory. This programming model may reduce the software complexity and aims for a more efficient processing, exploiting the available computer resources such as the memory subsystem and multithreading capabilities.

Traditional OOD makes heavy use of inheritance which tends to create complex class hierarchies, making the software difficult to maintain [10]. Although DOD can alleviate this limitation, the modeling of complex structures and relationships is not as natural as it is in OOD. Therefore, a design pattern called entity-component system is widely adopted in game engines to efficiently handle the creation and destruction of entities, and also to manage their underlying data (properties). The entity-component system can also replace inheritance trees by lightweight relations, like aggregation and composition, to build a more robust and modular software.

Similarly to game engines, electronic design automation (EDA) tools must be able to handle a high volume of data with very tight runtime budgets. Therefore, this work focuses on the discussion and application of these modern software concepts targeting the development of physical design tools. To discuss the related concepts we employ an open-source library for physical design as use case.

Since the early 2000's, there have been some efforts in the physical design community to create academic open-source databases and standardization, such as the works from [2], [3], [8] and [7]. However, few of the current projects are fully open source. For example, the Open Access project from [2] requires approved registration in order to be used, while the open database from [7] provides only binaries, and not the tools' source code. Even the truly open source initiatives, such as the open timer from [6], are mainly constructed using objected-oriented programming concepts, and therefore, might be improved.

In this direction, this work adopts as use case Ophidian, an Open-Source Library for Physical Design Research and Teaching [1], implemented by us, which is available through the collaborative GitHub platform. This library aims to fill the shortage of open source code for basic underlying infrastructure to facilitate research and teaching in the field. In Ophidian, we employed software development concepts borrowed from the game industry to handle large-scale designs. This way, Ophidian is used to explain and discuss how to take benefit from DOD to model circuit cells, pins, interconnections, and their respective properties.

Besides improving the software quality by overcoming the design problems of OOD, experimental results show that the DOD programming model is about 90% faster in a scenario that fully explores data locality, and it is only 6% slower in a


*ISPD '17, March 19-22, 2017, Portland, OR, USA*

DOI: http://dx.doi.org/10.1145/3036669.3038248

scenario with bad data locality. Such results show that this programming model can be efficiently used to solve physical design problems.

The rest of this paper is organized as follows. Section 2 presents the main characteristics and limitations of the OOD and DOD programming models. Section 3 details the entity-component system design pattern. Section 4 presents the experimental results and finally, Section 5 draws the conclusions.

## 2. OBJECT-ORIENTED DESIGN VS. DATA-ORIENTED DESIGN

This section discusses the use of OOD and DOD for software development. First, Section 2.1 shows a few limitations of OOD that make it difficult the development of software. Then Section 2.2 describes how the adoption of DOD may overcome shortcomings limitations, making also possible to improve memory access and program execution time.

### 2.1 Object-oriented design (OOD)

Typically, physical design tools are built using the OOD model, which decomposes a problem into objects. These objects are accessed through a well-defined interface, and their relations are represented through hierarchy, composition and aggregation [4]. The OOD programming model is popular because there is usually a one-to-one mapping between the real world objects and their corresponding objects in the program.

To illustrate the OOD basic concepts, suppose we are developing a physical design library to be used to solve different physical design problems. Then let us assume that a software developer wants to use such library to build a tool for estimating circuit interconnection wirelength. Figure 1 shows an example of a digital circuit containing four nets and eight pins. An interconnection wirelength estimator must get access to the information of which pins belong to each net, as well as to the pins' positions within the circuit layout.

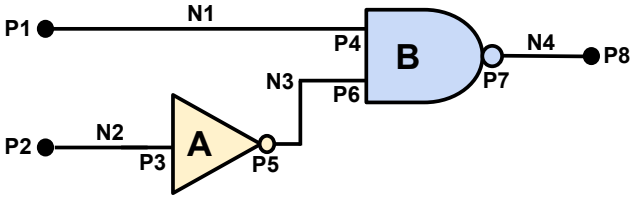


Figure 1: Combinational circuit portion with two logic gates (A and B), four nets (N1 to N4), and eight pins (P1 to P8).

Figure 2 illustrates a possible decomposition for the wirelength estimation problem by using OOD consisting of two modules: *netlist* and *placement*. The *netlist* module employs two classes, *net* and *pin*, to describe the circuit nets and the associated pins. For the *pin* class, this module characterizes only the name of the pin and the net such pin belongs to, without any placement information. The *placement* module, by its turn, describes the pins' positions. The diamond-end arrow between *pin* and *net* classes represents an aggregation relationship, which means that net has a reference to its pins, while a pin has a reference to its owner net. The triangle-end arrow between the two *pin* classes represents a hierarchy relationship, meaning that the *pin* class from the *placement* module extends the attributes from the *pin* class in the *netlist* module.

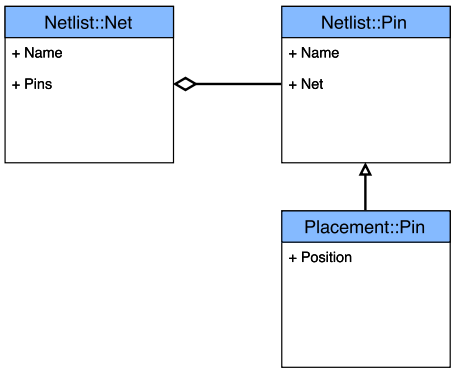


Figure 2: Class diagram to model wirelength estimation problem with OOD approach.

Although it may be easy to decompose a problem using OOD, building software based only on this programming model may lead to an overly complex class hierarchy. This issue is particularly critical in the development of a software library, because it is hard to predict, during the library design, how it will be actually used. For example, suppose that another problem requires pin timing information for the circuit of Figure 1. Then by using OOD, such information could be naturally added by creating a new module called *timing* and a new *pin* class (with pin timing attributes) in this module, which also extends the *pin* class from the *netlist* module. This new decomposition results in the class hierarchy shown in Figure 3.

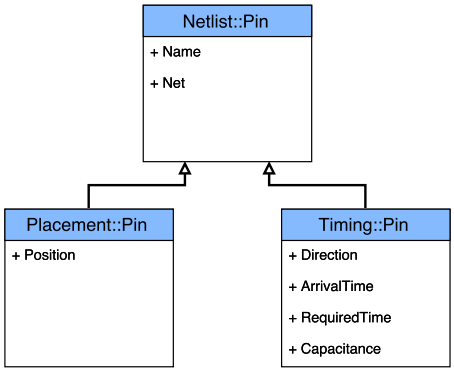


Figure 3: Class diagram to model when pin timing information is added.

Now suppose that another developer wants to use our physical design library to implement a timing-driven placement (ITDP) algorithm. In order to do that, she/he needs a new *pin* class with both placement and timing information. In OOD, this can be accomplished through multiple inheritance, where this new *pin* class extends the *pin* classes from *placement* and *timing* modules. However, multiple inheritance is not supported by all programming languages, and even when supported, it is not recommended because it may lead to design problems [10].

Without resorting to multiple inheritance, the solution consists on creating a new *pin* class that extends the one

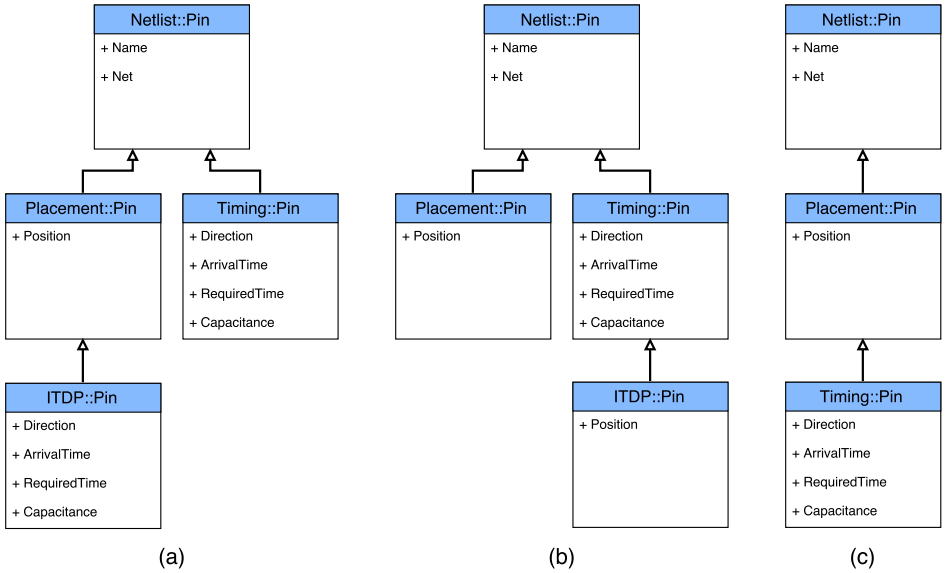


Figure 4: Possible class hierarchy to support information of *Timing* and *Placement* for a timing-driven placement algorithm following OOD approach.

from either the *placement* or the *timing* module, and repeating the code from the other class (that was not extended). Figures 4 (a) and (b) show these two solutions. Anyway, there is no clean manner of reusing placement and timing information without replicating code. The only remaining option is to push everything up in the *pin* class from the *timing* module by making it to extend the one from the *placement* module. This solution is illustrated in Figure 4 (c). However, it is not always necessary to have placement information in the *timing* module. For instance, a static timing analysis tool might not need placement information during early design steps. Therefore, adopting the latter solution would lead to memory waste, since unnecessary information would be stored.

## 2.2 Data-oriented design (DOD)

DOD may be used to overcome the previously mentioned limitations of OOD. This concept was first introduced in the work by Sharp [11], whose objective was to improve processing time and memory efficiency. Differently from OOD, the DOD programming model represents the problem objects, such as nets and pins, by indexes that are used to access their properties. The properties, by their turn, are contiguously stored in memory as data arrays. Figure 5 shows how we can model the same circuit from Figure 1 by following the DOD approach. The first two arrays describe the circuit's nets and pins as indexes. The remaining arrays describe their properties (equivalent to the attributes in OOD), which are accessed using the nets' and pins' indexes.

By doing so, the properties are not tied to objects, making it possible to add only the properties that are necessary for a given algorithm. For example, suppose an algorithm requires only placement information (as in global placement algorithms). In this case, the timing information is not necessary and the algorithm can include only the netlist and placement modules, resulting in the model shown in Figure 6 (a). On the other hand, if an algorithm needs only timing information (for example, for a static timing analysis

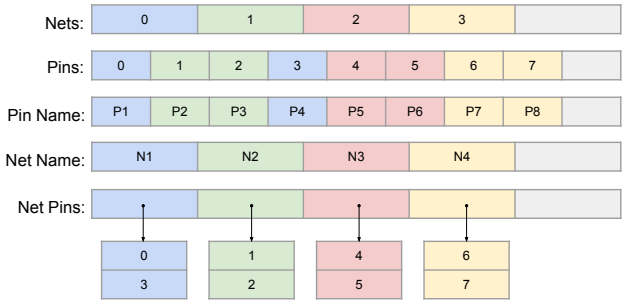


Figure 5: Modeling a combinational circuit with DOD approach. The lines represent arrays to describe nets and pins properties.

performed before placement), it can include only the netlist and timing modules, resulting in the model in Figure 6 (b). Finally, for the implementation of an incremental timing-driven placement tool, the algorithm should include all the three modules (netlist, placement and timing), resulting in the model in Figure 6 (c). Notice that such programming model enables developers to employ only the piece of data required for their algorithms, thus avoiding memory waste. In addition, by storing the arrays of properties contiguously in memory, a more efficient use of the memory hierarchy is enabled.

Although DOD can simplify the class hierarchy, its concept is not trivial to adopt, since most software developers are used to think about objects' relationships rather than data organization. In order to efficiently use this programming model, it is necessary to manage the arrays of properties to ensure they remain contiguous as new data is added and removed. A design pattern called **entity-component system** can be used to handle such problem [10], as will be presented in Section 3.

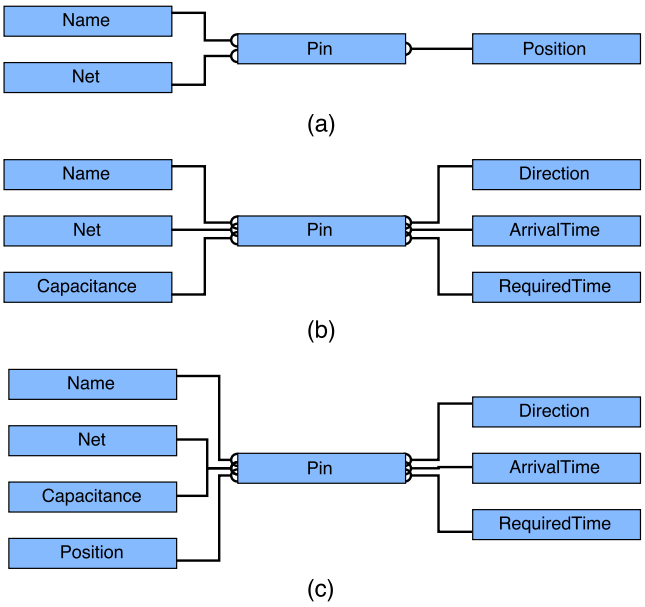


Figure 6: Properties of a pin using the DOD approach: pins have only placement information (a), pins have only timing information (b), pins have both information (c).

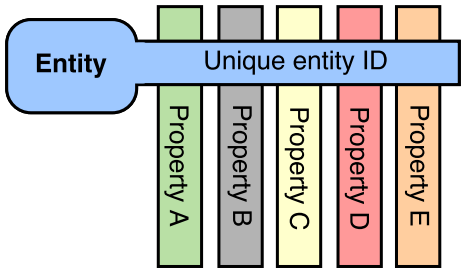


Figure 7: Example of entity with five properties. The entity id is used to access the information of all properties.

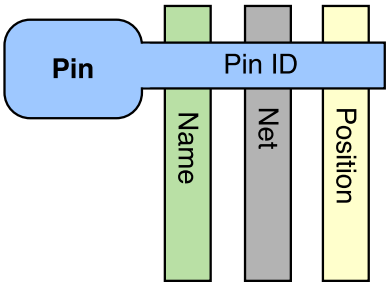


Figure 8: Possible representation of a pin using the entity-component system design pattern.

# 3. THE ENTITY-COMPONENT SYSTEM DESIGN PATTERN

This section introduces the concepts of the entity-component system design pattern. First, Section 3.1 presents the concepts of entity and component. Then Section 3.2 shows how the entity system manages these concepts. Finally, Section 3.3 explains how to implement relationships between entities using this design pattern.

## 3.1 Entities and components

The entity-component system design pattern consists on decomposing a problem into a set of entities and their components [10]. The entities are analogous to the objects in OOD, while the components correspond to their properties (or attributes). Hereafter, we refer to components as properties to adhere to the context of this work. Differently from objects, entities are not complex structures, but only unique identifiers (ids). Each property, by its turn, is represented using an array of data. The entity id is used to access its properties in those arrays. For instance, Figure 7 illustrates an entity with five properties, where each property corresponds to an array, and the entity id is used to access all arrays for a given index. Similarly to the properties, each one of these entities is stored in a contiguous array.

In order to decompose the wirelength estimation problem mentioned in Section 2 using the entity-component system pattern, the circuit nets and pins could be modeled as entities as follows. For each net, the properties would be the net name and its pins, while for each pin, the properties would be the pin name, its position and its net. Figure 8 illustrates a possible pin representation using the entity-component system design pattern.

## 3.2 The Entity system

An entity system is necessary to properly access entities and their properties. The entity system is responsible for creating and destroying entities, as well as managing the arrays of properties when changing the number of entities. For example, if an entity is created or destroyed, the entity system must efficiently ensure that all arrays remain contiguous in memory.

Besides managing the creation and destruction of entities, the entity system must provide an interface to access entities and properties with constant time complexity. In order to present the entity system algorithms, we are going to use the notation described in Table 1. Observe that $E_S$ and $P$ are arrays, and therefore, their values can be accessed through the corresponding indexes. For example, $E_S(i)$ represents the entity in the $i^{th}$ position of the array of entities.

Table 1: Main notation.

| Symbol | Meaning |
|---|---|
| $S$ | An entity system |
| $e$ | entity $e$ |
| $id_e$ | $id$ of entity $e$ |
| $E_S$ | array of entities belonging to an entity system $S$ |
| $E_S(i)$ | entity stored in the $i^{th}$ position of the array $E$ |
| $\mathcal{P}_S$ | set of properties associated to an entity system $S$ |
| $P$ | array of property $P$ |
| $P(j)$ | value stored in the $j^{th}$ position of a property array $P$ |

Algorithm 1 describes how the entity system creates new entities. When a new entity $e$ is created (line 1), the entity system assigns to $e$ an identifier $id_e$ equivalent to the size of the entities array $E_S$ (line 2). Then $e$ is inserted in the end of the entities array $E_S$, by calling the $PUSH\_BACK$ function (line 3). Finally, for each property associated to the entity system $S$, a default value is added for the new

entity $e$ (lines 4-6). Notice that the identifier of $e$ ($id_e$) is used as an index to access the arrays of properties.

**Algorithm 1:** ENTITY_CREATE

```
   Input  : Entity system S
   Output: New entity e
1  e ← new entity;
2  id_e ← |E_S|;
3  PUSH_BACK(E_S, e);
4  foreach P ∈ P_S do
5  |  P(id_e) ← default value;
6  end
7  return e;
```

Algorithm 2 presents the entity destruction steps. Given an entity $e$ to be destroyed, the entity system could simply remove it from the entities array. However, removing an element from the middle of a contiguous array has a time complexity of $\mathcal{O}(n)$, with $n$ being the number of entities. Another option would be to assign the entity $e$ as invalid, instead of removing it from the array. Nonetheless, this approach would leave holes in the entities array, which could hinder the entity system's performance.

Instead of removing the entity $e$ from the array or assigning it as invalid, the entity system replaces it with the last entity $e'$ from the entities array (lines 1-4). Then the last element of this array is removed by calling the $POP\_BACK$ function (line 5). This way, the entities array remains contiguous in memory and the destroy operation is performed in $\mathcal{O}(1)$. After replacing $e$ by $e'$, the entity system does the same for the arrays of properties, replacing the values associated to entity $e$ by the ones from entity $e'$ (lines 6-9).

**Algorithm 2:** ENTITY_DESTROY

```
   Input  : Entity system S, and entity e to destroy
   Output: Entity system S without e
1  n ← |E_S|;
2  e' ← E_S(n − 1);
3  E_S(id_e) ← e';
4  id_e' ← id_e;
5  POP_BACK(E_S);
6  foreach P ∈ P_S do
7  |  P(id_e) ← P(n − 1);
8  |  POP_BACK(P);
9  end
10 return S;
```

## 3.3 Relationships between entities

Instead of heavily using hierarchical relationships (like OOD), the entity-component system design pattern represents relations between entities using mainly composition and aggregation. A composition represents an ownership relationship. An aggregation, by its turn, is simply an association between different entities, without ownership [5]. For example, a circuit cell is composed of pins, which means that when the cell is destroyed all its pins must be destroyed too. On the other hand, the relationship between a net and its pins is simply an aggregation. As consequence, if a net is destroyed, the relationship is also destroyed, but all pins remain alive in the entity system.

Such relationships can be added to the entity system implementation (Algorithms 1 and 2) as special properties. This way, when the property is removed from the array of properties (line 8 from Algorithm 2), it automatically destroys the relationship. In addition, if the property is a composition, it also destroys the related entities.

# 4. EXPERIMENTAL RESULTS

In order to evaluate the impact of using either OOD or DOD on software performance, thus establishing a comparison between those programming models, we have implemented software prototypes for solving two physical design problems. Then those prototypes were run using the experimental infrastructure presented in Section 4.1. Then Section 4.2 describes the evaluated physical design problems, whereas Sections 4.3 and 4.4 present the experimental results for each problem.

## 4.1 Experimental infrastructure

We generated experimental results for the 8 circuits available from the ICCAD 2015 Contest (problem C: Incremental Timing-Driven Placement) [9]. These circuits were derived from industrial designs having from 768k to 1.93M cells. We performed all experiments in a Linux workstation whose architecture is depicted in Figure 9. This machine has an Intel® Core® i5-4460 CPU running at 3.20 GHz, 32GB RAM (4 × 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.

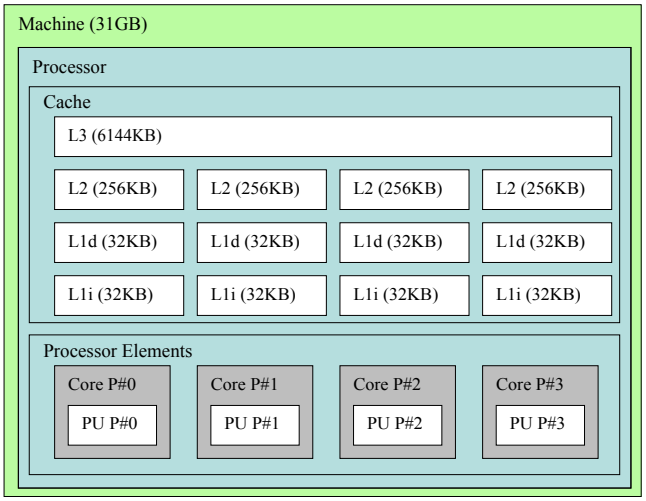


Figure 9: Architecture of workstation.

## 4.2 Evaluated physical design problems

We implemented the entity-component system design pattern described in Section 3 in the Ophidian library [1]. Ophidian is an open-source library developed by us for physical design teaching and research. It is composed of independent modules and implements the DOD concepts presented

in Section 2. In addition, Ophidian supports industrial formats, such as Verilog, DEF, LEF and Liberty, so that it can be used by other research groups.

Then we used Ophidian to implement software prototypes for the following physical design problems:

- Problem A: verifying if all circuit cells' positions lie within the circuit's physical boundaries. In this scenario only one property is necessary for each entity (cell position), so it fully explores the data locality provided by DOD.

- Problem B: estimating the interconnection wirelength for all circuit nets. This scenario accesses different properties of different entities. Therefore, it cannot efficiently explore the data locality provided by DOD, since the properties of the pins in a single net may not be contiguous in memory, unless the property array is previously sorted.

For each problem, two versions of prototypes were developed: one with the DOD model (using the entity-component system design pattern) and another with the OOD model. Therefore, in the next sections we compare the runtime and the number of cache misses for each programming model.

The experiments are available in the Ophidian GitHub repository [1], so that they can be easily reproduced[1].

## 4.3 Experimental results for Problem A

Figure 10 presents the average runtime results (y axis) for each circuit (x axis) in the problem A scenario. For all figures in this section, blue bars identify the DOD results whereas red bars denote the OOD results. The runtimes of the DOD version were between $4ms$ and $12ms$, whereas the runtimes of OOD were between $74ms$ and $186ms$. Therefore, for problem A, DOD is on average 94% ($16\times$) faster than OOD. The shorter runtime achieved by DOD is due to the fact that problem A uses only one entity-component system and only one property. This is the best scenario for DOD because it fully benefits from the cache hierarchy due to its spatial locality. Exploiting data locality reduces the time to access the main memory, which represents the main bottleneck.

To illustrate the superior cache locality of DOD, Figure 11 shows the number of cache misses (y axis) resulted by solving problem A using both programming models. The number of cache misses includes both instruction and data caches, and all three levels of cache available in the workstation. The number of cache misses for DOD (from $0.5M$ to $1.4M$) was, on average, one-tenth of those achieved by OOD (from $5.4M$ to $13.7M$). These results explain the much lower runtime of DOD for problem A.

## 4.4 Experimental results for Problem B

Figure 12 presents the average runtime results for problem B. The runtimes of DOD were between $3497ms$ and $10300ms$, while the runtimes of OOD were between $3306ms$ and $9654ms$. Differently from problem A, in order to solve problem B, the program employing the DOD programming

[1]The sources are available in the *ISPD2017* branch of the GitHub repository (https://github.com/eclufsc/ophidian/blob/ISPD2017/test/ispd/main.cpp).

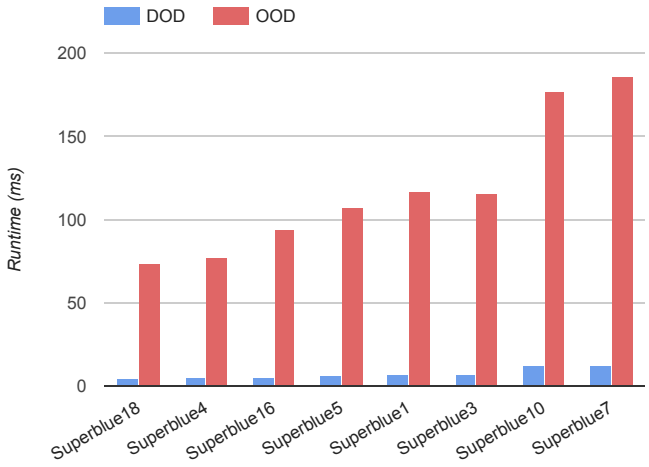


Figure 10: Runtime results for the two prototypes of problem A.

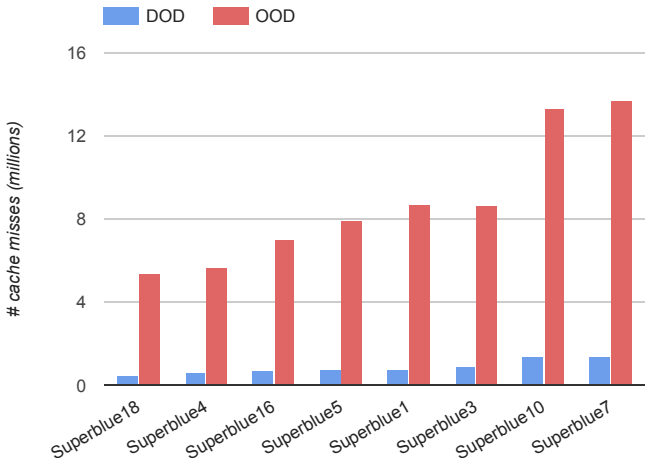


Figure 11: Cache miss results for the two prototypes of problem A.

model needs to access different entity-component systems (nets and pins) and multiple properties. In addition, the pins belonging to each net may not be contiguous in memory, which affects performance. As consequence, for problem B, OOD is on average 6% faster than DOD.

In order to verify the impact of the worse data locality of problem B, Figure 13 shows the number of cache misses resulted from each programming model. Observe that the number of cache misses of DOD (from $253M$ to $819M$) was, on average, 10% greater than those obtained by OOD (from A. The higher number of cache misses from OOD resulted in the lower performance for problem B. However, DOD still provides software engineering advantages by overcoming the limitations of OOD presented in Section 2.1. In addition, the performance difference in problem B (when DOD is slower) is much lower than that in problem A (when DOD is faster).

# 5. CONCLUSIONS

Inspired software development concepts employed by game industry, the DOD programming model (along with the entity-component design pattern) can be used to overcome limita-

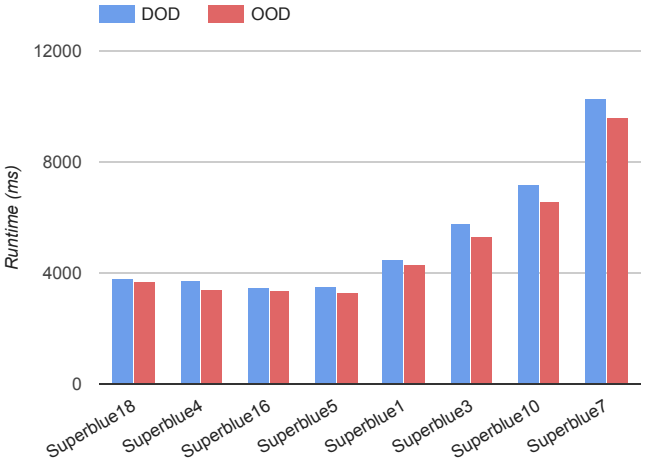


Figure 12: Runtime results for the two prototypes of problem B.

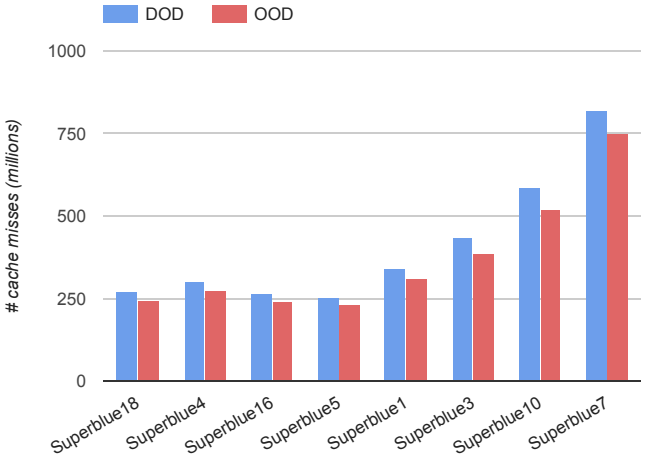


Figure 13: Cache miss results for the two prototypes of problem B.

tions of OOD in software implementation. This work discussed DOD concepts in the context of physical design problems, and presented an entity-component system implementation for such problems. The entity-component system design pattern was implemented in an open-source physical library called Ophidian.

The experimental results showed that the DOD programming model can reduce the program runtime by 94% on average, in a scenario that fully exploits data locality. These results are consequence of a lower number of cache misses provided by this programming model (90% reduction). On the other hand, in a scenario with worse data locality, using the OOD programming model resulted, on average, in a 6% faster solution. However, this faster runtime achieved by OOD in the second scenario does not compensate for its much slower performance in the first scenario. In addition, employing DOD solves some typical design limitations resulted from OOD, thus resulting in software quality improvements.

As future work, we intend to investigate the limitations of the DOD programming model in scenarios with worse cache locality. A possible technique to improve the data locality, therefore making possible to improve the performance of DOD, consists in sorting the arrays of properties in order to group values that should be accessed in sequence. In addition, we intend to investigate the parallelism potential of DOD. For problems with good data locality, the DOD programming model improves the memory access and, consequently, may increase the speedup of a parallel solution.

## 6. ACKNOWLEDGMENTS

This work was partially supported by Brazilian agencies CAPES, through M.Sc. grants, and CNPq, through Project Universal (457174/2014-5) and PQ grant 310341/2015-9.

## 7. REFERENCES

[1] Embedded Computing Lab, Federal University of Santa Catarina, “Ophidian: an open source library for physical design research and teaching”. https://github.com/eclufsc/ophidian.

[2] Silicon integration initiative, “open access”. http://www.si2.org/openaccess/.

[3] University of michigan, “umich physical design tools”. http://vlsicad.eecs.umich.edu/BK/PDtools/.

[4] G. Booch. *Object oriented analysis & design with application*. Pearson Education India, 2006.

[5] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[6] T.-W. Huang and M. D. Wong. Opentimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 895–902. IEEE Press, 2015.

[7] J. Jung, I. H.-R. Jiang, G.-J. Nam, V. N. Kravets, L. Behjat, and Y.-L. Li. Opendesign flow database: the infrastructure for VLSI design and design automation research. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 42. ACM, 2016.

[8] A. B. Kahng, H. Lee, and J. Li. Horizontal benchmark extension for improved assessment of physical cad research. In *Proceedings of the 24th edition of the great lakes symposium on VLSI*, pages 27–32. ACM, 2014.

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

[10] R. Nystrom. *Game programming patterns*. Genever Benning, 2014.

[11] J. A. Sharp. Data oriented program design. *ACM SIGPLAN Notices*, 15(9):44–57, 1980.