Exploiting Cache Locality to Speedup Register Clustering

Tiago Augusto Fontana, Sheiny Almeida, Renan Netto, Vinicius Livramento, 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
{tiago.fontana, sheiny.fabre}@posgrad.ufsc.br

ABSTRACT

Physical design tools must handle huge amounts of data in order to solve problems for circuits with millions of cells. Traditionally, Electronic Design Automation tools are implemented using Object-Oriented Design. However, using this paradigm may lead to overly complex objects that result in waste of cache memory space. This memory wasting harms cache locality exploration and, consequently, degrades software runtime. This work proposes applying Data-Oriented Design on the register clustering problem. Differently from the traditional Object-Oriented design, the Data-Oriented Design programming model focus on how the data is organized in the memory. As consequence, this programming model may better explore cache spatial locality. In order to evaluate the impact of using the Data-Oriented Design programming model for register clustering, we implemented two software prototypes (a sequential and a parallel implementation) of the K-means clustering algorithm for each programming model. Experimental results showed that the sequential Data-Oriented Design implementation is on average 7.5% faster when compared to the Object-Oriented Design implementation, while its parallel version is 15% faster when compared to the Object-Oriented one.

CCS CONCEPTS

• **Hardware → Physical design (EDA)**; *Software tools for EDA*; • **Software and its engineering →** Software development techniques;

KEYWORDS

Electronic Design Automation, Physical Design, Data-Oriented Design, Register Clustering, Cache locality.

ACM Reference format:
Tiago Augusto Fontana, Sheiny Almeida, Renan Netto, Vinicius Livramento, Chrystian Guth Laércio Pilla and José Luís Güntzel. 2017. Exploiting Cache Locality to Speedup Register Clustering. In *Proceedings of 30th Symposium on Integrated Circuits and Systems Design, Fortaleza, Ceará Brazil, August 2017 (SBCCI 2017),* 7 pages.
DOI: 10.1145/3109984.3110005

SBCCI 2017, Fortaleza, Ceará Brazil
 978-1-4503-5106-5/17/08…$15.00
DOI: 10.1145/3109984.3110005

1 INTRODUCTION AND RELATED WORK

The evolution of Integrated Circuits (ICs) fabrication technology has allowed a drastic and continuous reduction in transistor dimensions, allowing the fabrication of ICs with billions of transistors. To cope with such huge amounts of transistors and with the consequent complexity, the design of contemporary ICs must heavily rely on Electronic Design Automation (EDA) tools [10].

EDA tools have to handle a large volume of data in a short amount of time. For example, problems such as register clustering [14] [18], legalization [1] [17] and timing analysis [6] require a series of operations to be done on millions of circuit cells.

To ensure execution within acceptable runtimes, EDA tools must take full advantage of software optimizations, such as: usage of better data structures, parallelization, and cache locality exploration. If we examine the current EDA tools made available by the academia, such as the works from [3] [7] [8] [9] [12], all of them perform a number of software optimizations, but none of them focus on data organization to explore cache locality.

By properly organizing data, we can improve cache locality by keeping data with common properties close to each other in the memory. For example, the work of [4] evaluated the impact on the execution time of some basic physical design tasks when the programming model called Data-Oriented Design (DOD) is adopted. Differently from the traditional Object-Oriented Design (OOD) paradigm, DOD focuses on data organization rather than on class hierarchy therefore providing explicit control of data organization. They showed that a better data organization can drastically reduce the number of cache misses, which consequently improve software runtime with a great volume of data. However, their work has two main limitations: 1) the experimental data was obtained for the problem of checking if a cell is within the chip boundaries and for interconnection wirelength estimation. Although these are very important tasks that are to be run many several times during physical design, they are very simple in terms of operations and do not represent more challenging tasks. 2) they did not evaluate parallel solutions, although the use of DOD may increase the speedup achieved by such solutions.

This work evaluates the impact of the DOD programming model used by [4] in the context of a more complex physical design task, evaluating both sequential and parallel solutions. As case study, we chose the register clustering problem, since it handles a large amount of data (hundreds of thousands of registers), and it is a very important task to reduce the total power of a circuit. For example, it is the problem tackled by the works of [14] [18].

Therefore, the main contributions of this work are the evaluation of the DOD programming model in a more complex physical design

problem (register clustering), and the evaluation of the speedup achieved by parallel solutions when employing such programming model. To analyze the impact of cache locality we implemented two different versions of the K-means clustering algorithm, a classic method for register clustering. The only difference between implementations are how data is organized in memory (traditional OOD vs. DOD).

The experimental evaluation was performed using the ICCAD 2015 ITDP Contest infrastructure [11]. Experimental results for sequential implementations of the K-means algorithm show average cache misses reduction of 24% when using the DOD programming model. As consequence, the DOD approach executes on average 9.6% faster than the OOD approach. On the other hand, when analyzing parallel implementations of the same algorithm, the use of DOD resulted in 20% less cache misses, on average, and 15% faster implementations.

The remaining of this paper is organized as follows. Section 2 formulates the register clustering problem and describes the K-means algorithm. Section 3 presents the DOD programming model and how it can be applied to the register clustering problem. Section 4 brings the experimental evaluation and finally, Section 5 draws the conclusions.

2 REGISTER CLUSTERING PROBLEM

The clock distribution of high performance circuits requires a hierarchical topology composed of global, regional and local clock networks [5]. The global and regional networks are responsible for delivering the clock signal from its source to the local clock network using different topologies such as mesh, tree or hybrid [19]. The local clock distribution is often implemented as buffered trees of registers. Each buffered tree is constructed by clustering groups of closely placed registers, wherein each cluster is driven by a local clock buffer. The process of grouping closely placed registers is called **register clustering**, and can be formalized as follows:

Given a set of register locations $\mathcal{R} = \{r_1, r_2, ..., r_n\}$ where each $r_i \in \mathbb{R}^2$, the register clustering problem can be defined as: partitioning the set $\mathcal{R}$ into a predefined number k of clusters (subsets) $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_k\}$ such that the sum of intra-cluster wirelength is minimized, as stated through Equations (1) to (4). Each $\gamma_i \in \Gamma$ represents the set of elements that belong to a cluster i. Equation (1) presents the objective function defined under the L_1-norm (Manhattan distance), where $c_i \in \mathbb{R}^2$ represents a given cluster center. Generally, the cluster's center is computed as the center of gravity of the registers belonging to the cluster as $c_i = \sum_{r_j \in \gamma_i} r_j / |\gamma_i|$, where $|\gamma_i|$ represents the cardinality of γ_i. The set of constraints defined by Equations (2) to (4) simply defines a partition of $\mathcal{R}$, i.e. every register belongs to a single cluster and all registers are clustered.

$$\textbf{Min} \quad : \quad \sum_{i=1}^{k} \sum_{r_j \in \gamma_i} \|c_i - r_j\|_2^2 \tag{1}$$

$$\textbf{S.t.} \quad : \quad \gamma_i \in \Gamma \implies \gamma_i \neq \emptyset \tag{2}$$

$$: \quad \gamma_i, \gamma_j \in \Gamma \ \wedge \ i \neq j \implies \gamma_i \cap \gamma_j = \emptyset \tag{3}$$

$$: \quad \mathcal{R} = \bigcup_{\gamma_i \in \Gamma} \gamma_i \tag{4}$$

In this work we solved the register clustering problem using K-means [16]. K-means is a popular clustering algorithm, used not only to solve register clustering, but also in different areas, such as machine learning and data mining. Algorithm 1 describes the steps of the K-means algorithm, while Figure 1 illustrates a simple example of its execution.

The algorithm receives as input a set $\mathcal{R}$ of register locations and a set $\mathcal{C}$ of k initial cluster centers. There are different ways to determine the number of clusters. In this work we defined a maximum number (50) of elements per clusters and used this number to determine the number of clusters ($k = \lceil \frac{|\mathcal{R}|}{50} \rceil$). The initial centers positions were randomly determined using a uniform distribution. The execution time of the algorithm is propositional to the number of clusters. Therefore, a good choice of cluster number has great impact on the execution time of the algorithm. The algorithm's output is a set of clusters and their new centers.

Initially, all clusters are empty (line 2, Figure 1 (a)). Then, the algorithm solves the clustering problem by iteratively executing two main steps: **assignment step** and **update step**. During the assignment step (lines 3 to 6), each register is assigned to the cluster with the closest center (Figure 1 (b)). After assigning all registers to some cluster, the update step (lines 7 to 9) relocates all cluster centers to be the center of mass of its elements (Figure 1 (c)). These two steps are executed until convergence or by a fixed number of iterations. A possible final solution of K-means execution can be viewed in Figure 1 (d).

The K-means algorithm performs operations in every register of a circuit. The total number of registers in a modern IC is on average 10% to 15% of the total numbers of cells (a modern IC typically contains millions of cells). By this reason, the K-means algorithm operates on a huge amount of data. By improving the data access we can reduce the algorithm runtime, as it will be presented in Section 3.

Algorithm 1: K-means

Input : Set $\mathcal{R}$ of register locations, set $\mathcal{C}$ of k cluster centers
Output: Set of clusters Γ and their new cluster centers $\mathcal{C}$

```
1  do
2      γ_i ← ∅, i = 1 ... k;
3      foreach r_i ∈ R do
4          c_j ← arg min_{c_j ∈ C} {||c_j − r_i||_2^2};
5          γ_j ← γ_j ∪ {r_i};
6      end
7      foreach γ_i ∈ Γ do
8          c_i ← Σ_{r_j ∈ γ_i} r_j / |γ_i|;
9      end
10 while clusters centers do not converge;
```

3 THE PROPOSED DATA ORIENTED IMPLEMENTATION FOR REGISTER CLUSTERING

This section focuses on the discussion on how the DOD programming model can improve the runtime of the register clustering

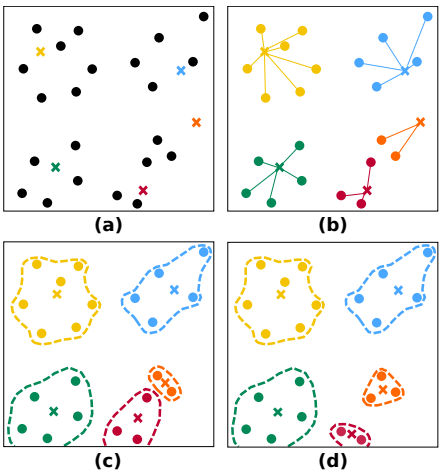

Figure 1: Execution example for K-means algorithm: black dots represent registers, crosses represent cluster centers and dashed lines delimit cluster regions. In (a), randomly generated initial cluster centers are shown. In (b), each register is assign to the closest cluster. In (c) the cluster centers are updated to a center of mass, and (d) represents a possible final solution after some iterations.

problem. Section 3.1 explains DOD and its data organization. Section 3.2 shows how to apply DOD concepts in the register clustering problem.

3.1 Data-Oriented Design (DOD)

The memory architecture of modern computers is typically hierarchical as shown in Figure 2. Observe that the lower levels in the hierarchy (e.g. hard disk and main memory) have higher capacity but also higher latency. On the other hand, the higher levels (e.g. cache memory and CPU registers) are fast but with small capacity. When a program needs to access the memory, first it tries accessing the data in the higher levels. When the data is not found in the cache levels we say that a **cache miss** occurs. When a cache miss occurs, the data is accessed from the main memory and then saved in the cache, so that it can be accessed from it later on. Whenever this data is necessary again, it will be available in the cache, and its search in the cache levels will result in **cache hit**.

The **cache spatial locality** is an important property of the memory hierarchy systems. This property states that the likelihood of accessing a resource from the memory is higher if a nearby resource was just referenced [15]. The cache system explores this property by storing the data in **cache blocks**. Whenever data is accessed from the memory, it is saved in the cache together with other nearby data. This way, if the nearby data is accessed afterwards, it will result in a cache hit.

It is possible to further explore the cache spatial locality by changing how the data is organized in a given software. For example, by contiguously saving data in the memory it is possible to speedup the memory access of an algorithm that operates in a big set of data (e.g. a register clustering algorithm that performs operations on all circuit registers). As consequence, the software performance is

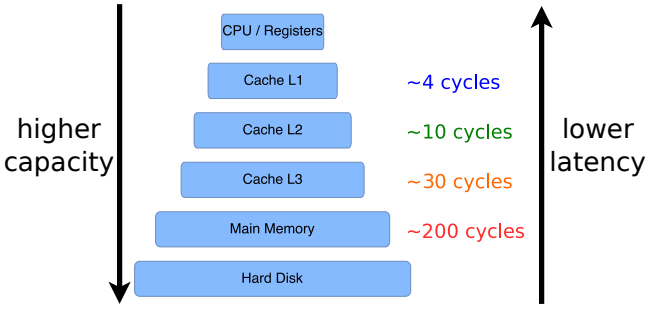

Figure 2: Hierarchy of memory present in modern computers. The hierarchy is compose by hard disk, main memory, three levels of cache and CPU registers. Adapted from [15].

improved. This data organization is not always efficiently done by the traditional OOD paradigm. Note that, in OOD paradigm, when objects are fetch to cache some usefully data are fetched to cache together. Therefore, cache is wasted with data that is unlikely to be used in OOD paradigm.

An alternative for the OOD is the DOD programming model, which focuses on how data is organized in the memory. Figure 3 shows an example on how to model data using this programming model. While OOD typically uses complex objects and their attributes to represent data, DOD represents it using entities and properties. Each entity is simply an index used to access its properties, while each property is stored as a separate array. For example, in Figure 3, the array entities stores all the entity indexes, while the other arrays store their properties, which can be accessed using the indexes from the entities array. By organizing the data this way, when an algorithm needs only one property (e.g. Property A), it does not need to bring needless data to the cache, making a better use of the cache spatial locality. In the next section we will present how this concept can be applied to the register clustering problem.

Entities:	0	1	2	3	
Property A:	A	B	C	D	
Property B:	(1, 2)	(2, 1)	(3, 2)	(5, 5)	
Property C:	False	True	True	True	

Figure 3: Modeling entities and their properties with DOD approach.

3.2 DOD Improving Register Clustering

In order to confront DOD and OOD we first show how the register clustering problem can be modeled by using the traditional OOD. Figure 4 shows a possible class diagram to model the register clustering problem. In this Figure, there are two classes: *Register* and *Cluster*. The *Register* class has a position attribute, necessary for the register clustering problem, but also the register name, which could be necessary in other parts of the software. The *Cluster* class, by its turn, has as attributes its centers and the registers assigned to it (stored in an array).

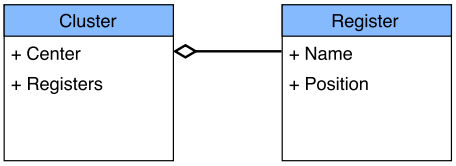

Figure 4: Class diagram to model register clustering problem using OOD. The diamond-end arrow between *Cluster* and *Register* classes represents an aggregation relationship.

Note that, the OOD keeps all attributes inside their objects what gathers easy access to all of its attributes. However, some algorithms need only to access a subset of the object data, and keeping them all together inside the object may lead to wasted cache space. For example, Figure 5 shows in the left-hand side a possible implementation of the *Register* class presented in Figure 4, and a code that iterates through all registers accessing their positions. The right-hand side shows the cache content when running this code, assuming a cache block of 128 bytes. Observe that, since the *Register* object contains a *name* attribute which is not used by the code, 50% of the cache capacity is wasted with unused data.

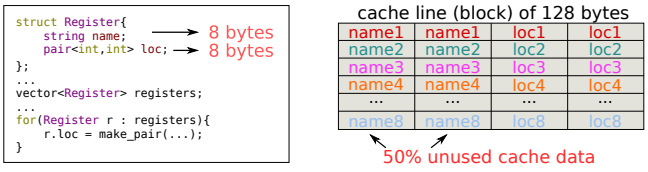

Figure 5: The impact of object modeling with OOD on cache usage. The left-hand side represents a possible implementation of the *Register* class presented in Figure 4. Right-hand side shows the cache content when running this code.

As mentioned in the previous Section, it is possible to improve the data organization using the DOD programming model. Figure 6 shows how to model the register clustering problem using this programming model. Observe that this modeling contains the same data as the OOD one presented in Figure 4. The only difference is that now the data is organized using the entities and properties approach from Figure 3. The clusters and registers are represented by entities, each one stored in an independent contiguous array, as well as their properties (cluster centers, cluster registers, register names and register positions).

By keeping the data stored in independent contiguous arrays, an algorithm that needs to access only a subset of it does not waste cache space with unused data. For example, Figure 7 shows a code example that performs the same operation as in Figure 5, but following the DOD programming model. Observe that now the properties are stored in independent arrays, so that the loop that accesses register locations does not need to waste cache memory space with the register names. As consequence, the cache is filled only with useful data.

As an extra benefit, the DOD can provide better data organization that can be particularly exploited by parallel implementations to achieve much higher speedups. For example, Algorithm 2 presents

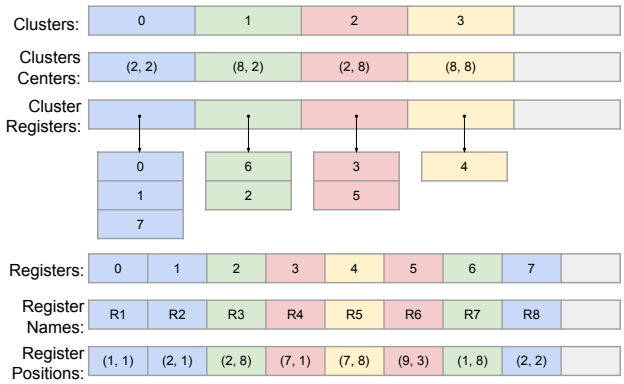

Figure 6: Modeling register cluster problem with DOD approach. The lines represent arrays to describe cluster and registers attributes.

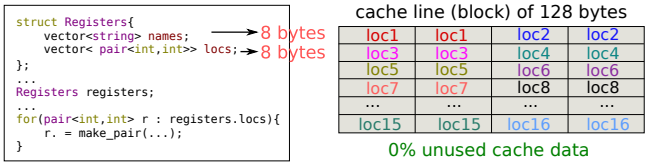

Figure 7: The impact of object modeling with DOD on cache usage. The left-hand side represents a possible implementation of the *Register* class presented in Figure 6. Right-hand side shows the cache content when running this code.

a possible parallel version of the Algorithm 1. Observe that the loops that assign each register to a cluster (lines 3 to 6) and update the cluster centers (lines 11 to 13) were parallelized since there is no data dependency between the register locations. However, the step of adding each register to the elements set for the closest cluster can not be performed in parallel (line 5 of Algorithm 1), since more than one register can be assigned to the same cluster. This would result in a race condition.

A possible solution to solve this race condition would be to add a semaphore or mutex when a register is added to a cluster, but this would reduce the performance of the parallel algorithm. To solve this issue, we proposed another variable, called α, to store temporarily the best cluster of each register (line 5). After the assignment step, the algorithm sequentially iterates over the registers adding each register to the previously found cluster (lines 7 to 10).

In the next section, we evaluate the impact of the data organization in the number of cache misses and, consequently, runtime. We also compare the performance improvements caused by parallelization when we use OOD and DOD programming models.

4 EXPERIMENTAL RESULTS

In order to evaluate the impact of data organization in the number of cache misses and runtime, thus establishing a comparison between OOD and DOD programming models, we implemented four software prototypes to solve the register clustering problem.

Algorithm 2: Parallel K-means

Input : Set $\mathcal{R}$ of register locations, set of $\mathcal{C}$ cluster centers
Output: Set of clusters Γ and their new cluster centers $\mathcal{C}$

do
  $\gamma_i \leftarrow \emptyset,\ i = 1 \ldots k$;
  parallel **foreach** $r_i \in \mathcal{R}$ **do**
    $c_j \leftarrow \arg\min_{c_j \in \mathcal{C}} \{\|c_j - r_i\|_2^2\}$;
    $\alpha_i \leftarrow \gamma_j$;
  end
  foreach $r_i \in \mathcal{R}$ **do**
    $\gamma_j \leftarrow \alpha_i$;
    $\gamma_j \leftarrow \gamma_j \cup \{r_i\}$;
  end
  parallel **foreach** $\gamma_i \in \Gamma$ **do**
    $c_i \leftarrow \sum_{r_j \in \gamma_i} r_j / |\gamma_i|$;
  end
while *clusters centers not converge*;

The first two are sequential implementations using each programming model (following the steps described by Algorithm 1). The other two are parallel implementations using Algorithm 2 and each programming model. We used the OpenMP parallel programming interface [13] to parallelize Algorithm 2. All versions were implemented in C++ and the source code is available online [2].

The rest of this section is organized as follows: Section 4.1 presents the experimental infrastructure. Section 4.2 reports the experimental results for sequential implementations of K-means algorithm. Next, Section 4.3 shows the experimental results for the parallel implementations.

4.1 Experimental Infrastructure

We generated experimental results for the 8 circuits available from the ICCAD 2015 Contest (problem C: Incremental Timing-Driven Placement) [11]. 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 (DDR3 at 1600MHz). The workstation processor has four cores and three levels of cache. The first two levels of cache (L1 and L2) are private for each processor and have a size of 64KB and 256KB, respectively. The third level of cache (L3) is shared among all processor and has a size of 6144KB. All results presented in this section represent the average of 30 executions to ensure a small confidence interval.

4.2 Experimental Results for the Sequential Implementation

Figure 8 presents the number of total cache misses (y axis) resulting from the sequential implementations running on each circuit (x axis). For all figures in this section, blue bars identify the DOD results whereas red bars denote the OOD results. 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 17M to 173M) was, on average, 24% smaller than that achieved by OOD (from 29M to 214M). This result indicates that, as expected, the cache spatial locality is better explored using DOD.

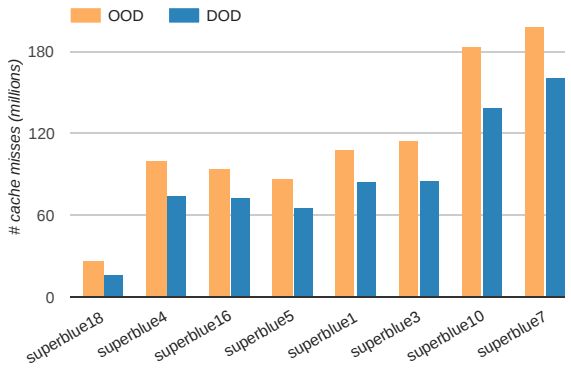

Figure 8: Cache miss results for the sequential implementation of register clustering problem.

The smaller number of cache misses for DOD should also reduce the algorithm runtime. To verify that, Figure 9 shows the average runtime results (y axis) for each circuit (x axis). The runtimes of the DOD version were between 724ms and 2364ms, whereas the runtimes of OOD were between 804ms and 2526ms. On average, the DOD programming model is 7.5% faster than OOD. Therefore, these results confirm that by exploiting cache locality it is possible to speed up register clustering algorithms.

Note that the reduction in the misses does not improve runtime in the same proportion. This is due to the fact that, by reducing cache misses, we only reduce the runtime of instructions that access memory. Nevertheless, these instructions are only a subset of the software instructions.

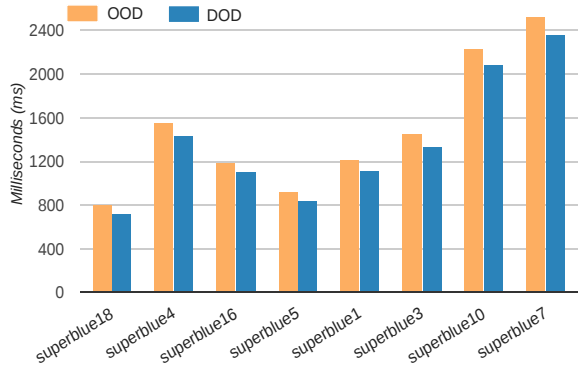

Figure 9: Runtime results for the sequential implementation of register clustering problem.

4.3 Experimental Results for the Parallel Implementation

Figure 10 presents the number of total cache misses (y axis) in the parallel implementation. Note that the overall number of cache

misses has reduced since the parallel implementation uses the L1 and L2 caches of all cores, increasing the available amount of memory. The number of cache misses for DOD (from 7M to 131M) was, on average, 20% less than achieved by OOD (from 17M to 159M). This result demonstrates that, as expected, data organization led to a smaller number of cache misses even for the parallel implementation. Once again, the parallel version using DOD programming model obtained the lowest average number of cache misses for all versions.

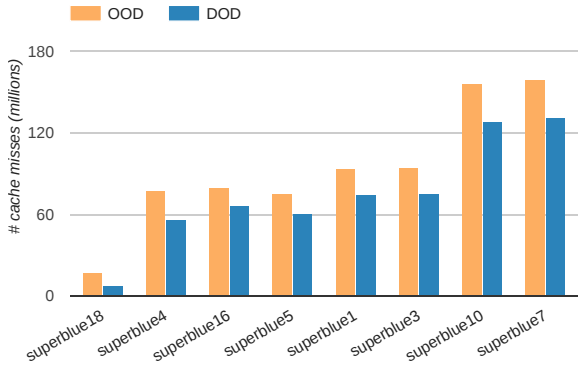

Figure 10: Cache miss results for the parallel implementation of register clustering problem.

The reduction of cache misses for DOD also reduced the runtime. Figure 11 shows the runtime results (y axis) for the parallel implementation. The runtimes of the DOD were between 264ms and 769ms, whereas the runtimes of OOD were between 300ms and 896ms. Therefore, the DOD programming model is, on average, 15.5% faster than OOD when both are parallelized.

Note that, while for the sequential implementation DOD is on average 7.5% faster than OOD, in the parallel implementation this difference increases to 15%. Such results show that the speedup achieved by the parallel implementation is higher when applied using the DOD programming model, even though the difference in the number of cache misses between the implementations was lower for the parallel scenario (20% compared to 24%). Indeed, considering only DOD versions, the parallel implementation achieved an average speedup[1] of 3.04 with respect to the sequential one, while when comparing the two OOD versions the speedup was of only 2.78. A possible reason for these results may be that the memory access time is a more relevant factor in the parallel implementations than in the sequential ones, making the DOD programming even more efficient in this scenario.

5 CONCLUSION

Despite the large amount of data handled by EDA tools, typically those tools do not employ any special data organization to explore cache locality. This work investigated the impact of data organization in EDA in the context of the register clustering problem. To evaluate the such impact, we implemented sequential and parallel versions of K-means (a classic algorithm to solve register clustering), using two different programming models (OOD and DOD).

[1]The *speedup* metric represents the ratio between the execution times of the sequential solution and the parallel solution for a given number of threads.

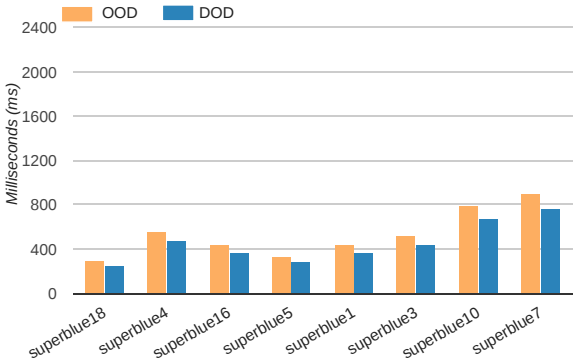

Figure 11: Runtime results for the parallel implementation of register clustering problem.

The experimental results showed that the DOD programming model achieved lower number of cache misses and runtime when compared to OOD in both the sequential and parallel implementations. When comparing the parallel DOD implementation, which achieved the best results, to the traditional approach (OOD sequential implementation), the reduction on number of cache misses was up to 75% (42% on average). Correlating the runtime of both implementations, the parallel DOD was up to 70% faster than sequential OOD, being on average 47.5% faster. These experimental results confirm that DOD programming model is an adequate way to represent data for register clustering problems.

As future work we intend to evaluate the impact of the chunk size used by OpenMP in the parallel implementations. The chunk size represents the amount of data (loop iterations) allocated to each thread. By properly sizing the chunk according to the cache block size, it may be possible to improve the cache locality even further. Another possible future work consists on sorting the arrays of properties according to the algorithms being executed. Some algorithms may take advantage if the data follows a specific sorting scheme. For example, for a legalization algorithm, it may be interesting if the cell locations are sorted according to circuit rows. By storing the properties in independent arrays, as in the DOD programming model, it is possible to apply different sorting schemes for each property, taking advantage of different access patterns. Finally, we intend to investigate how the DOD approach can benefit softwares running on GPU and FPGA-CPU architectures.

ACKNOWLEDGMENTS

This work was partially supported by the Brazilian Federal Agency for the Support and Evaluation of Graduate Education (CAPES), through M.Sc. grants, and by the Brazilian Council for Scientific and Technological Development (CNPq), through Project Universal (457174/2014-5) and PQ grant 310341/2015-9.

REFERENCES

[1] Wing-Kai Chow, Chak-Wa Pui, and Evangeline FY Young. 2016. Legalization algorithm for multiple-row height standard cell design. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[2] Federal University of Santa Catarina Embedded Computing Lab. 2017. Ophidian: an Open Source Library for Physical Design Research and Teaching. https://github.com/eclufsc/ophidian. (2017).
[3] 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*. ACM, 33–40.
[4] 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.
[5] M. Guthaus, G. Wilke, and . Reis. 2013. Revisiting automated physical synthesis of high-performance clock networks. *TODAES* 18, 2 (2013), 31:1–31:27.
[6] 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.
[7] Silicon Integration Initiative. 2017. Open Access. http://www.si2.org/openaccess/. (2017).
[8] Jinwook Jung, Iris Hui-Ru Jiang, Gi-Joon Nam, Victor N Kravets, Laleh Behjat, and Yin-Lang Li. 2016. OpenDesign flow database: the infrastructure for VLSI design and design automation research. In *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 42.
[9] Andrew B Kahng, Hyein Lee, and Jiajia Li. 2014. Horizontal benchmark extension for improved assessment of physical CAD research. In *Proceedings of the 24th edition of the great lakes symposium on VLSI*. ACM, 27–32.
[10] 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.
[11] 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.
[12] University of Michigan. 2017. UMICH Physical Design Tools. https://www.src.org/library/publication/p013527/. (2017).
[13] OpenMP. 2017. The OpenMP API. http://openmp.org/. (2017).
[14] David Papa, Charles Alpert, Cliff Sze, Zhuo Li, Natarajan Viswanathan, Gi-Joon Nam, and Igor Markov. 2011. Physical synthesis with clock-network optimization for large systems on chips. *IEEE Micro* 31, 4 (2011), 51–62.
[15] David A Patterson and John L Hennessy. 2013. *Computer organization and design: the hardware/software interface*. Newnes.
[16] Shokri Z Selim and Mohamed A Ismail. 1984. K-means-type algorithms: a generalized convergence theorem and characterization of local optimality. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 1 (1984), 81–87.
[17] Chao-Hung Wang, Yen-Yi Wu, Jianli Chen, Yao-Wen Chang, Sy-Yen Kuo, Wenxing Zhu, and Genghua Fan. 2017. An effective legalization algorithm for mixed-cell-height standard cells. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 450–455.
[18] Gang Wu, Yue Xu, Dean Wu, Manoj Ragupathy, Yu-yen Mo, and Chris Chu. 2016. Flip-flop clustering by weighted K-means algorithm. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.
[19] C Yeh, G Wilke, Hongyu Chen, and others. 2006. Clock distribution architectures: A comparative study. In *ISQED*. 85–91.