Cache management in MASCARA-FPGA: from coalescing heuristic to replacement policy

Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau, Laurent d'Orazio

► To cite this version:

Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau, Laurent d'Orazio. Cache management in MASCARA-FPGA: from coalescing heuristic to replacement policy. DaMoN 2022 - 18th International Workshop on Data Management on New Hardware, Jun 2022, Philadelphia, United States. pp.1-5, 10.1145/3533737.3535096 . hal-03907912

HAL Id: hal-03907912

https://inria.hal.science/hal-03907912v1

Submitted on 6 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cache management in MASCARA-FPGA: from coalescing heuristic to replacement policy

VAN LONG NGUYEN HUU, Nokia Bell Labs, Univ Rennes, CNRS, IRISA, and B<>com , France
JULIEN LALLET, Nokia Bell Labs and B<>com , France
EMMANUEL CASSEAU, Univ Rennes, CNRS, IRISA , France
LAURENT D'ORAZIO, Univ Rennes, CNRS, IRISA and B<>com , France

Abstract. We presented ModulAr Semantic CAching fRAmework (MASCARA) that deployed Semantic Caching (SC) to perform a fast query processing based on Field Programmable Gate Arrays (FPGAs) accelerators. In addition of the accelerators, cache management plays an important role to address coalescing strategy and replacement policy so as to maximize the performance of FPGA caching. Therefore, in this paper, we present a coalescing heuristic with a new replacement function that leverages advantages of traditional strategies and overcomes their drawbacks. The proposed heuristic reduces response time, improves data availability, and saves cache space with respect to the semantic locality of query workload.

1 INTRODUCTION

Recently, acceleration for the data management system (e.g., Spark [2]) with a specific computing hardware (e.g., Field Programmable Gate Array FPGA), has declined by both industry and academia, such as, [15], [11], [13], [18], [16], [7], and [3]. Moreover, using a caching technique, for example, Semantic Caching (SC) [10], [5] which could finely exploit data and knowledge in a query or reduce data transfer, should also be considered. Therefore, we have proposed a Modular Semantic Caching framework (MASCARA) [8] that relies on the acceleration via FPGA kernels (e.g., Query Trimming) [12].

State of the art. MASCARA-FPGA speeds up the execution of a query or a sequence of queries. To reach this goal, in addition to the appropriate accelerators, dealing with cache management is important to maximize the performance. In particular, cache management in SC consists of two main procedures: *the coalescing strategy* and *the replacement policy*. The first one determines how the data regions are formed, merged and partitioned, to ensure the optimal granularity of the cached elements while minimizing the overhead of using the cache space. Meanwhile, the second one determines how semantic information is added to and removed from the cache. The coalescing strategy could be solved by the straightforward approaches, such as: *Nerver Coalescing* or *Always Coalescing* ([4], [9], [6], [17], or [14]). Since the above solutions are limited by the capabilities of CPU, the contributions were not interested in studying the effects of different strategies. In general, both of them have major side effects to FPGA caching (i.e., MASCARA-FPGA), such as, cache space utilization, the granularity problem, or the responses times of the system. Furthermore, they can be exacerbated if we do not have an appropriate replacement policy that should be adapted for query workloads with general semantic locality.

Contribution. Therefore, we propose a heuristic solution that leverages their advantages and minimize their side effects. In particular, it decides whether to coalesce data regions based on the recency of usage (*temporal locality*) and percentage of response contribution (*spatial locality*) that are presented through a new replacement function. It is worth to note that our solution is flexible when the semantic locality of query workload changes according to their applications. Experiments with TPC-H benchmark [1] validates that MASCARA-FPGA with the heuristic has: a fast response time, a high cache efficiency, and a low overhead in memory usage. For example, with two kernels, the response time of MASCARA-FPGA is faster up to 9.23 times than the MASCARA-Server (baseline) with $1GB$ of dataset.

2 COALESCING IN MASCARA-FPGA

We presented the SC concept and prototype of MASCARA-FPGA in [8, 12]. If we have N data region DR and the answer of an incoming query Q overlaps them, the result could be the formation of $2N + 1$ new DR, where $N + 1$

of them are the answer to the query. The following question arises: *should we combine some, all, or none of the $N+1$ DR into one or more larger DR* [9] (as shown in Figure 1).

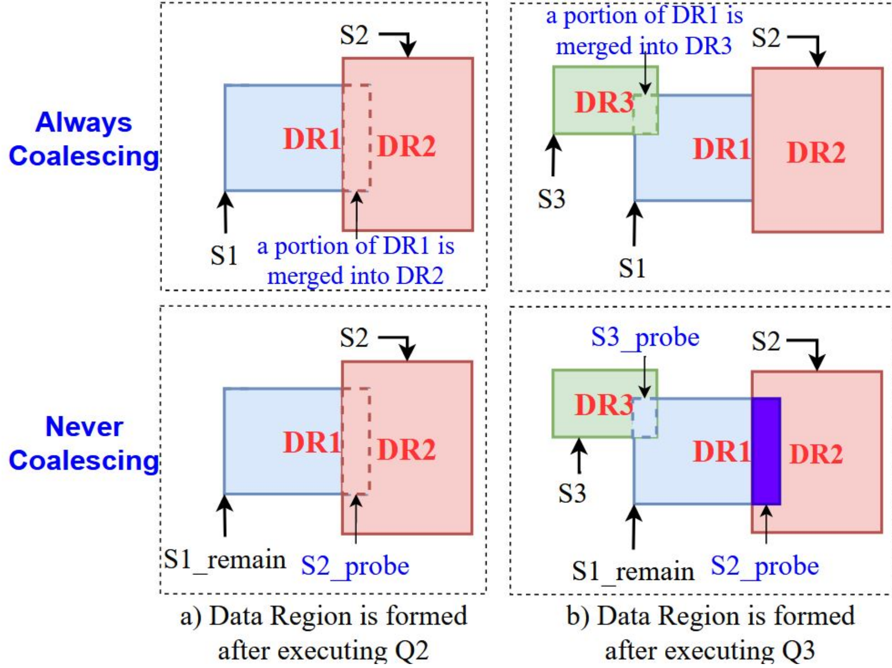


Fig. 1. Data Region in Always and Never Coalescing

2.1 Conventional strategies

Always Coalescing will coalesce *all* the *DR* which contributes to the answer of the most recent Q, so that only one *DR* corresponds to the query. Without generating a large number of S, it results to a good performance because we can avoid the overhead of *Query Trimming* accelerators. The worst case occurs when the response of Q takes a large portion of cache, resulting to a poor space utilization. In contrast, *Never Coalescing* does not coalesce any data region that contributes to the most recent queries, resulting in a drastically increase of number and complexity of S and *DR*. Thus, this solution could have a negative impact on responses times even if we have multiple parallel accelerators on FPGA.

Obviously, applying individually the two solutions raises the issues of efficiency, space utilization, or end-to-end response time of cache. Therefore, it seems reasonable to use them alternatively by checking the current situation of data regions and/or their future contributions in query answers. In particular, the segments S representing corresponding *DR* should be measured and indexed by their "*profit*" S_V in the cache. The function of calculating S_V is based on *temporal* (e.g., LRU) and *spatial locality* (e.g., contribution to query answering). In other words, S_V is now considered to use in both coalescing and replacement.

2.2 Heuristic: from coalescing to replacement

In this section, we present how we manage the cache through a heuristic which considers both coalescing and replacement. Given the recency of usage, we assume that the most recent coalescing/replacement value is V_{max},

which is increased by one for each new Q. Meanwhile, the coalescing/replacement value for each data region is S_V. The process of finding the *profit* of a *DR* is divided into two steps: 1) computing the *intermediate profit* and deciding whether to merge based on its value, and 2) computing *future profit* of the its remaining part (as shown in Algorithm 1).

Algorithm 1: Cache management with heuristic

Input: Input: cache with list of S, DR and a query Q
Output: Output: cache updated with coalescing heuristic

```
1  Pass Query Trimming, outputs are: PQ, RQ
2  Execute PQ and RQ
   /* Replacement with the minimal ratio                                  */
3  while r ≠ End Of List do
4  |   r := S_V/size_of_S_D
5  |   Finding the minimal r
6  end
7  Victim := S with DR that has minimal r
8  Remove Victim
9  Add RQ with new DR
   /* Heuristic of coalescing: step 1                                     */
10 Choosing threshold T
11 Assign Q_V := Vmax
12 while S ≠ End Of List do
13 |   p := R_Q/R
14 |   S_V_inter := S_V_ori + (V_max − S_V_ori) ∗ p
15 |   if S_V < Q_V ∗ T then
16 |   |   Coalescing between S_D and Q_D
17 |   end
18 |   else if S_V >= Q_V ∗ T then
19 |   |   No-coalescing between S_D and Q_D
20 |   end
21 end
   /* Heuristic of coalescing: step 2                                     */
22 while S ≠ End Of List do
23 |   pdis := S_V_inter − S_V_ori
24 |   S_V := pdis ∗ (1 − p) + S_V_ori
25 end
```

From row 10 to 21 in Algorithm 1, we measure the percentage of the data region that contributes to the query answer: $p = R_Q/R$, where R_Q is the number of records that match the query answer and R is the total number of records in the region. Then, the new replacement/coalescing value is temporarily updated as follows: $S_V = S_V + (V_{max} - S_V) * p$. Note that $(V_{max} - S_V)$ ensures that the new S_V does not have a higher value than V_{max}. In other words, the new data region with respect to the last query Q will have the highest value S_V. This function is adaptable for all regions in SC, regardless of whether the region contributes to answering the query or not.

Based on the updated S_V for all regions contributing to the query answer, we propose a threshold T as a part of "coalescing filter" $S_V < Q_V * T$ that decides whether to merge all, some or none of them. In general, T can be

scaled from 0 (*Never Coalescing*) to 1 (*Always Coalescing*). Meanwhile, $Q_V = V_{max}$ is the value of the new DR with respect to the new query appears. If $S_V < Q_V * T$, the overlapping part between existed DR and new DR of Q will be merged (coalesced) into the old one. Thus, the number of generated or cached segments are stored, resulting in an efficient response time of MASCARA-FPGA. Otherwise, if $S_V >= Q_V * T$, the new DR of Q will be cached which has the same value S_V as its *predecessor* DR. By this way, this decision increases the data granularity of the cache, resulting in higher efficiency. Although we have not yet explored the cost model related to the "coalescing filter" as well as the optimization problem (e.g., Knapsack or Dynamic Programming), we can adjust the threshold T in practical to find a "reasonable coalescing filter" based on cache performance, cache efficiency, and cache space usage.

As the second step (from row 22 to 25 in Algorithm 1), after merging some of the DR, the remaining ones might shrink into the new parts. Therefore, their *future profit* that could be evaluated for the next queries should be recalculated as follows: $S_V = pdis * (1 - p) + S_{V_ori}$ where S_{V_ori} is the *original profit* of the DR before starting the procedure. $pdis = S_{V_inter} - S_{V_ori}$, consists of the *profit distance*, is the gap between the *intermediate profit* S_{V_inter} and the *original profit* S_{V_ori}.

Example 2.1. As shown in Figure 2, we assume that the contribution of DRi and DRj to the query answer is $p_i = 0.75$ and $p_j = 0.71$, respectively. The last value $V_{max} = 35$ for the appearance of query Q. Although p_i, p_j are nearly equal, their contributions to the answer are different in size (i.e., $DRi > DRj$). From (a) to (b), the S_V of S_i has increased significantly from 10.3 to 18.4. If we choose $T = 0.5$, then $T * Q_V = 0.5 * 35 = 17.5$ does not pass the condition of "coalescing filter" ($17.5 < 18.4$). Thus, cache will keep the overlapping part between DR_i and DR_Q as a totally new one. In contrast, with the same procedure, DRj does not pass the "filter". It means that cache will merge the overlapping part of DRj into DR of Q. At the end of the procedure, in (c), we reevaluate the *future profit* of DR_i, DR_j to prepare for the next Q. For example, using the formula consists of $pdis$, the *future profit* of the remaining DRi is: $S_V = (18.4 - 10.3) * (1 - 0.75) + 10.3 = 12.325$.

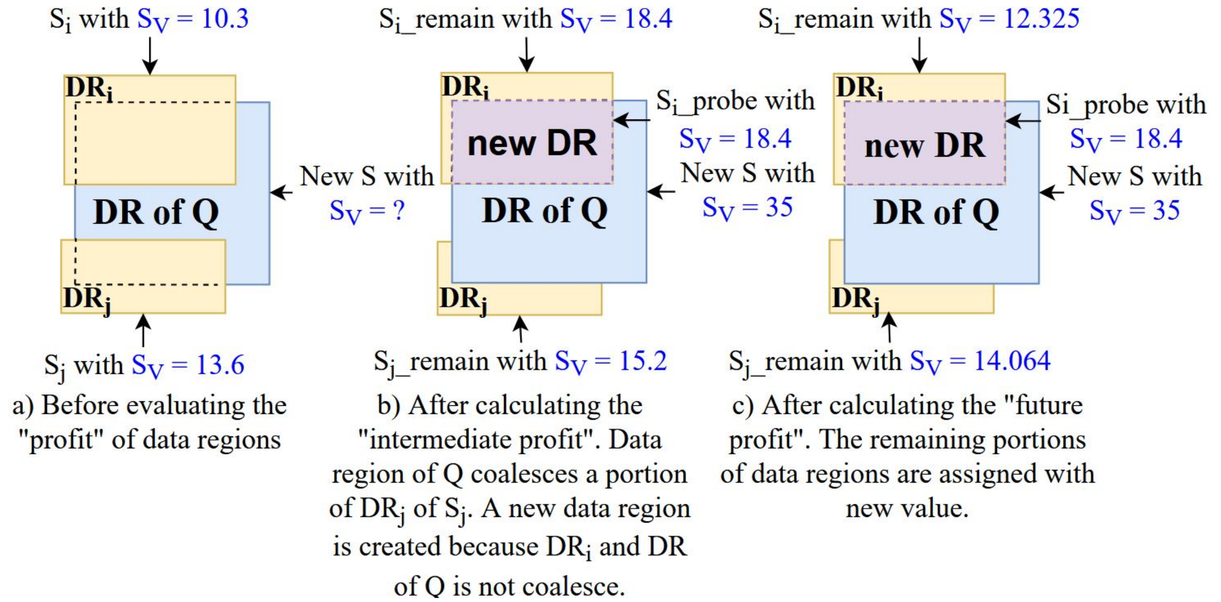


Fig. 2. The coalescing heuristic in cache management

Since S_V is used for both coalescing and replacement, our heuristic overcomes the limitation of LRU in the context of SC. Indeed, considering that *DR* can vary in size, removing it from the cache should depend not only on its contribution to the last query response, but also on its actual size. In other words, we calculate the ratio r between the actual coalescing/replacement S_V and its size s in the cache: $r = S_V / size_of_S_D$ (from row 3 to 9 in Algorithm 1). Thanks to the real representation of r (e.g., $S_V = 12.325$ in the above example) if the cache needs space for a new data region *DR*, the selection of a victim would be more accurate than LRU. It should be noted that *DR* which overlaps the query response are excluded from this procedure. In summary, by approximating both *temporal* and *spatial locality*, the impact of query workload (i.e., semantic locality) could be alleviated in general applications.

3 EXPERIMENTS

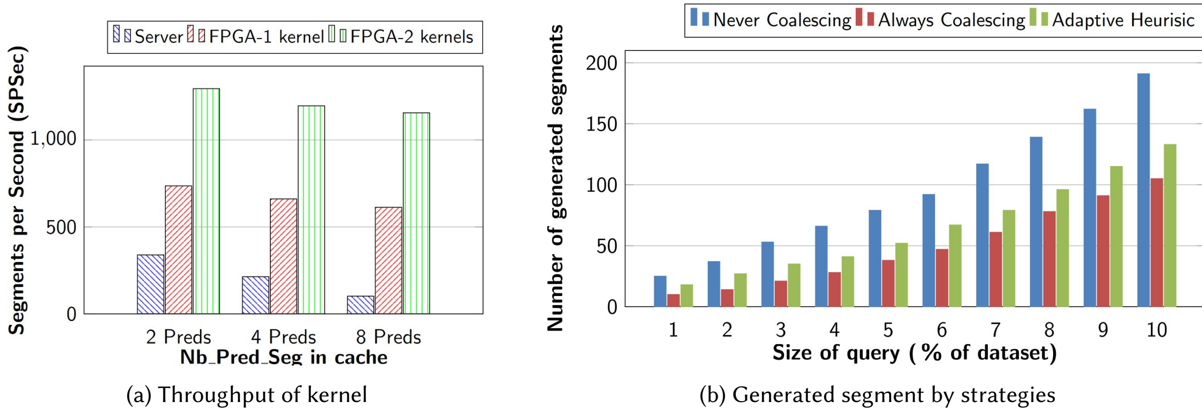
Configuration. A server is equipped with an Intel® Xeon® Gold 5118 CPU running at 2.30 GHz and 64 GB RAM. The FPGA platform is an Alveo U200 board with three 16GB DDR4.

Query workload. Experiments are performed with only one relation (i.e., *lineitem*) of TPC-H [1]. Based on the query Q6 of TPC-H, we generate the range query with the select-project format.

Influencing parameters The parameters that could affect the performance of MASCARA-FPGA are presented in [12]. Moreover, we consider also *Skew* (a fixed fraction of the queries that has semantic centerpoint within a *Hot Region*).

Evaluated prototypes. We evaluate our contribution by comparing several prototypes: (A) MASCARA-Server with *Always Coalescing* LRU, (B) MASCARA-FPGA with *Never Coalescing* LRU, (C) MASCARA-FPGA with *Always Coalescing* LRU, and (D) MASCARA-FPGA with heuristic.

3.1 Segments - Query Trimming throughput



(a) Throughput of kernel

(b) Generated segment by strategies

Fig. 3. Impact of coalescing strategies

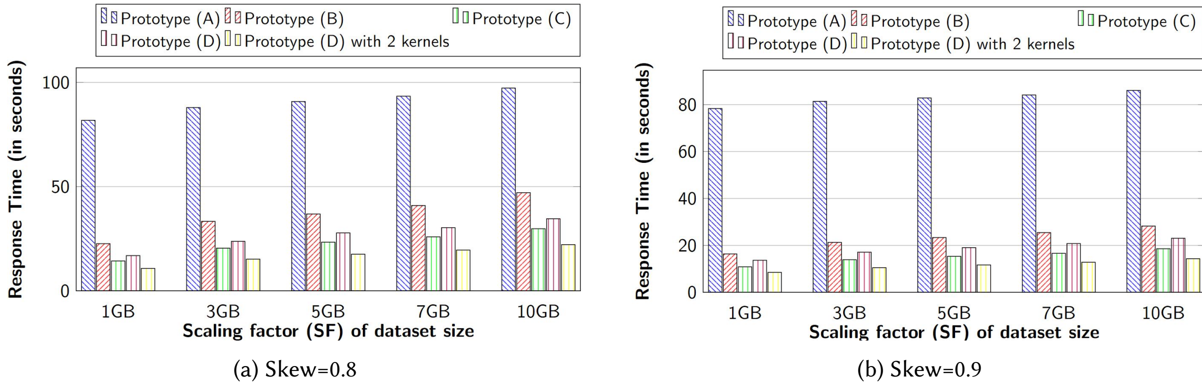
This experiment is performed by measuring the processing time of *Query Trimming* over 1000 segments *S*. Obviously, as shown in Figure 3a, the Query Trimming accelerator has a higher *SPSec* than similar procedure on Server. When *Nb_Pred_Seg* increases, *SPSec* decreases significantly. In particular, *SPSec* of server drops to 102 *SPSec*, meanwhile, *SPSec* of kernel is 661 when *Nb_Pred_Seg* = 4. If we deploy two kernels in parallel, *SPSec* increases from 661 *SPSec* to 1196 *SPSec*. To resume, it can be seen that the Query Trimming on FPGA

outperforms the server based on CPU only when the query complexity (e.g., Nb_Pred_Seg) increases. Moreover, with high *SPSec* of accelerators, the main issue of *Never Coalescing* now is alleviated while keeping its benefits in MASCARA-FPGA.

A large number of Nb_Seg could also cause a bottleneck in Query Trimming [12]. As shown in Figure 3b, when the query size increases, Nb_Seg grows rapidly, especially with *Never Coalescing* (e.g., 191 S for query size = 10%). With $T = 0.5$, NB_Seg of our heuristic is slightly higher than the best one *Always Coalescing* (e.g., 133 S for query size = 10%). In summary, we consider that MASCARA-Server could also apply the heuristic but the performance might not be significant as *Always Coalescing* due to the limited throughput of *Query Trimming* on server. Therefore, it is more preferable to MASCARA-FPGA in which we have high throughput Query Trimming kernels.

3.2 Cache performance

As it can be seen in Figure 4a, the FPGA prototypes (B, C, and D) have lower response time than (A) on server. Obviously, (C) is the fastest because using *Always Coalescing* reduces the number of generated segments, which leads to fast execution of *Query Trimming*. Meanwhile, our heuristic (D) finds a good balance between (B) and (C), and shows a remarkable speed-up (i.e., 4.84 with $SF = 1GB$). We also found that the speed up of the FPGA prototypes decrease when SF increases. For example, at $SF = 10GB$, (B) is 2.07, (C) is 3.26, and (D) is 2.81 times faster than (A). As shown in [8, 12], in this case we need to execute the *RQ* if they appear over a large and full (i.e., $10GB$) dataset on the server. Fortunately, with reasonable hardware consumption (presented in Section 3.5), we can use multiple accelerators (i.e., two kernels) to mitigate this problem.



(a) Skew=0.8

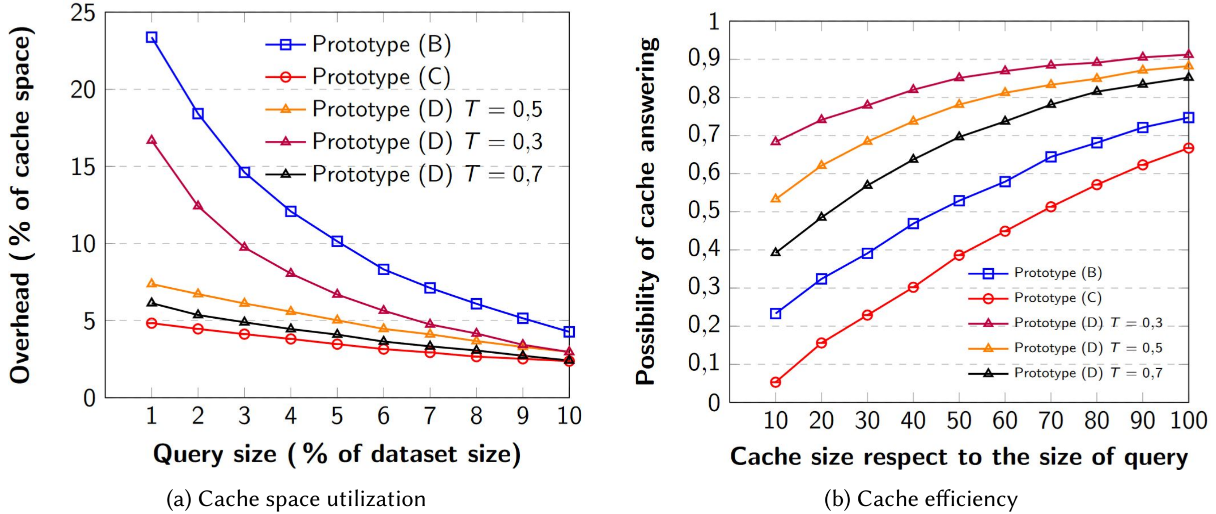
(b) Skew=0.9

Fig. 4. Response Time when changing semantic locality. 50 queries workload. Cache has 1000 S. Hot Region = 20%.

In Figure 4b, we change the semantic locality through *Skew*, to show that our heuristic is preferable for general applications. Indeed, the speed up of FPGA prototypes increases since most of the (probe) queries are executed on FPGA, which is faster than the server thanks to its accelerator *Probe Query Executing*. For example, at $SF = 1GB$, we observe that (B), (C), and (D) are faster than (A) 4.78, 7.22 and 5.73 times, respectively.

3.3 Cache space utilization

We run the experiment through 50 queries with $Nb_Pred_Query = 4$, which has the same response size (as show in Figure 5a). Meanwhile, the cache should be initialized with almost 1000 segments S that have $Nb_Pred_Seg >= 4$ thanks to the warm-up queries.



(a) Cache space utilization

(b) Cache efficiency

Fig. 5. Space overhead and data availability of cache. Hot Region = 20% and Skew = 0.8.

Obviously, with $N+1$ new *DR* that are not merged, *Never Coalescing* will store the duplicated key attributes that leads to significant overhead of cache memory. For example, with a workload of 1% queries, we need more space in prototype (B) than usual (23.38%) to store the fragmented *DR*. Meanwhile, in (C), the cache space requirement has only 4.83%. (D) is slightly higher than (C) (7.37%) because we merge only the *DR* that satisfies our heuristic condition. Moreover, when the query size increases, the space overheads of all strategies decrease and converge at a certain query size (i.e., query size workload = 10%).

The other influencing factor of our heuristic is the threshold T. It is obvious that when T increases, for example $T = 0.7$, the possibility of merging in (D) is high, which allows saving more space in the cache memory. In contrast, reducing T means that the cache prefers to keep the data region independent to increase the data granularity. Depending on the status of the cache, T should be changed dynamically to maintain system performance.

3.4 Cache efficiency

We measure the cache efficiency of our heuristic by varying the size on average between 10 and 200 times of queries in workload (as shown in Figure 5b). Our solution allows to have a highest cache efficiency compared to the conventional approaches. For example, when cache size is small (e.g., ten times the query size), prototype (D) with $T = 0.3$ could answer almost 68% of the queries (i.e., data regions of *PQ*s are 68%). In detail, our heuristic evaluates the *contribution-to-size* ratio that considers both *temporal* and *spatial locality*. However, when cache size increases (i.e., is more than 50 times the query size), the results of the prototypes tend to saturate. For example, when the cache size is 100 times of the query size, the cache efficiencies are 74%, 66%, and 88% with respect to (B), (C), and (D), with $T = 0.3$.

Besides cache space utilization, T also changes cache efficiency of (D). For example, if T changes from 0.3 to 0.5, the cache efficiency decreases from 68% to 53%, where the cache size being times larger than the query size. This efficiency could decrease even further if we increase T from 0.5 to 0.7. Obviously, the ability to answer queries does not differ too much between the prototype when cache size is large enough.

3.5 Hardware resources

The hardware resource consumption of MASCARA-FPGA is shown in Table 1. *Cache Manager: Heuristic* has a small footprint in our prototype thanks to the arbitrary precision fixed-point data type of computing S_V and r. Another positive point is that we only need one *Cache Manager: Heuristic* in MASCARA-FPGA.

Component	Look-Up Tables (LUTs)	Flip Flops (FFs)	36Kb BRAM
Attribute Matching	4476	2914	11
Predicate Matching	17668	16621	84
Semantic Extracting	6838	3682	17
Probe Query execution	7739	5436	31
Query Process Controller	5394	4379	10
Cache Manager: Heuristic	6627	5461	44
Total resources	48712	38493	197
Alveo-U200 usage (%)	5.46	2.1	11.15

Table 1. Hardware resources of MASCARA-FPGA

4 CONCLUSION

In this paper, we propose a coalescing heuristic with new replacement function that takes advantage of the two straightforward approaches: *Always* and *Never Coalescing*. In particular, this heuristic takes into account the *temporal* and *spatial locality* of the data regions with respect to the query response by computing their *profit* in the cache. To summarize, we improved the performance of FPGA caching in three ways: end-to-end response time, cache efficiency, and cache space utilization. As future works, we are interested in: i) Investigating a cost model for automatically selecting the optimal coalescing threshold T based on the cache state in real-time. ii) Extending MASCARA-FPGA towards join queries with fragmented data regions.

ACKNOWLEDGMENTS

This work is funded by N°2019/0477 CIFRE scholarship of ANRT within the project of B<>Com and Nokia Bell Labs, France.

REFERENCES

[1] TPC Professional Affiliates. 2022. TPC-H Benchmark revision 3.0.1. Retrieved May 10, 2021 from https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia). 1383–1394.

[3] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. 2014. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Strathclyde Academic Media, Glasgow, GBR.

[4] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases* (Mumbai, India). 330–341.

[5] Laurent d'Orazio and Julien Lallet. 2018. Semantic Caching Framework: An FPGA-Based Application for IoT Security Monitoring. *OJIOT* 4 (2018), 150–157.

[6] Parke Godfrey and Jarek Gryz. 1999. Answering Queries by Semantic Caches. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA '99)*. Springer-Verlag, Berlin, Heidelberg, 485–498.

[7] P. Gupta. 2016. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)* (Lausanne, Switzerland) *(FPL 2016)*.

[8] Van Long Nguyen Huu, Julien Lallet, Emmanuel Casseau, and Laurent d'Orazio. 2020. MASCARA (ModulAr Semantic CAching fRAmework) towards FPGA Acceleration for IoT Security Monitoring. *OJIOT* 6 (2020), 14–23.

[9] Björn Þór Jónsson, María Arinbjarnar, Bjarnsteinn Þórsson, Michael J. Franklin, and Divesh Srivastava. 2006. Performance and Overhead of Semantic Cache Management. *ACM Trans. Internet Technol.* 6, 3 (2006), 302–331.

[10] Arthur M. Keller and Julie Basu. 1996. A Predicate-Based Caching Scheme for Client-Server Database Architectures. *VLDB J.* 5 (1996), 035–047.

[11] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2010. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA). 1159–1162.

[12] Van Long Nguyen Huu, Laurent d'Orazio, Emmanuel Casseau, and Julien Lallet. 2021. *MASCARA-FPGA Cooperation Model: Query Trimming through Accelerators.* Association for Computing Machinery, 203–208.

[13] M. Owaida, D. Sidler, K. Kara, and G. Alonso. 2017. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines* (Napa, CA, USA). 211–218.

[14] Qun Ren, M.H. Dunham, and V. Kumar. 2003. Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering* 15, 1 (2003), 192–210.

[15] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. 2017. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems* 51 (2017), 142–164.

[16] Bharat Sukhwani, Mathew Thoennes, Hong Min, Parijat Dube, Bernard Brezzo, Sameh Asaad, and Donna Dillenberger. 2015. A Hardware/Software Approach for Database Query Acceleration with FPGAs. *International Journal of Parallel Programming* 43 (2015), 1129–1159.

[17] Andrei Vancea, Laurent d'Orazio, and Burkhard Stiller. 2011. A Scalable Cooperative Semantic Caching (CoopSC) Approach to Improve Range Queries. IEEE, New York, NY, USA, 45–54.

[18] Louis Woods, Jens Teubner, and Gustavo Alonso. 2013. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA). 1073–1076.