A Shared Memory Cache Layer across Multiple Executors in Apache Spark

Wei Rang
Department of Computer Science
UNC at Charlotte
Charlotte, USA
wrang@uncc.edu

Donglin Yang
Department of Computer Science
UNC at Charlotte
Charlotte, USA
dyang33@uncc.edu

Dazhao Cheng
Department of Computer Science
UNC at Charlotte
Charlotte, USA
dazhao.cheng@uncc.edu

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378179

***Abstract*—Memory caches are being widely adopted in today's data-intensive computing frameworks to maximize the benefit of in-memory access. Various memory architectures, designs and management algorithms have been well explored so that application performance can witness speedup by orders of magnitude. For example, Apache Spark provides intermediate data consistency in memory between computation tasks, eliminating a significant amount of disk I/Os and dramatically reducing data processing times. However, the memory space of individual executors is isolated so far in vanilla Spark and apparently inefficient given the fact that these memory demands of different executors vary a lot over tasks, jobs and applications. In this work, we propose a new shared in-Memory cache layer, i.e., iMlayer, among these parallel executors, which are co-hosted on the same slave machine in Apache Spark. It aims to improve the overall hit rate of data blocks by caching and evicting these blocks uniformly across multiple executors. The critical insight of iMlayer is to develop a novel eviction strategy to efficiently manage the shared cache space among executors to maximize the cache hit rate as well as application performance. We evaluate iMlayer based on the three representative workloads from HiBench. Our results demonstrate that iMlayer with the new eviction strategy improves the cache hit rate by 45%, 16% and 27%, effectively reducing the overall job runtime 47%, 43% and 38% compared to vanilla Spark, respectively.**

***Index Terms*—Memory Utilization, Cache Layer, Eviction Policy, Re-reference Distance, Multiple Executors, Apache Spark**

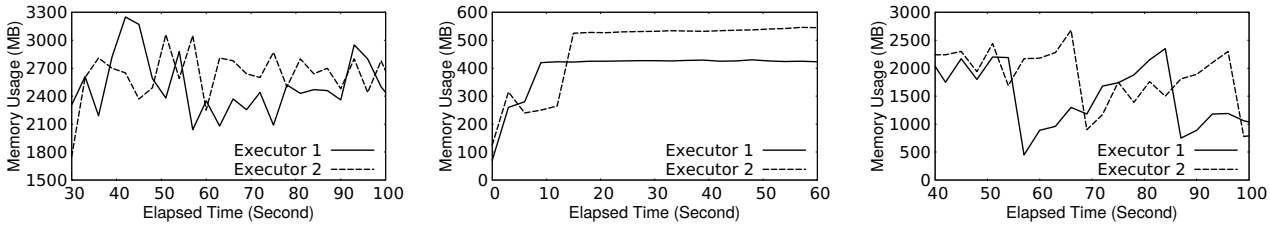
I. Introduction

Memory plays a pivotal role in many popular distributed in-memory computing frameworks, such as Spark [1]. In such systems, frequent I/O operations can be significantly reduced so that application performance would be sped up by orders of magnitude via caching input and intermediate data into a specific memory space. To guarantee proper memory utilization, a well-designed management strategy is essential for these in-memory computing frameworks, especially with increasing memory resources in cluster nodes. Accordingly, many memory allocation strategies [2] have been adopted in data-parallel frameworks. For example, a huge amount of memory is preferred for a single executor on Apache Spark cluster in order to cache more intermediate data. In-memory computation is a significant feature of Spark platform so that computation efficiency could be further improved by avoiding frequent I/O operations between memory space and local disk. However, a large memory space assignment always increases Garbage Collections (GC) overhead for the JVM based in-memory computing framework, i.e., Spark [3]. Given the fact that the execution process has to be paused when it is suffering from GC operations, setting a large memory space for individual executors is not always beneficial and should be avoided. It indicates a larger memory may not always guarantee a better performance due to frequent GC operations.

Another alternative solution is increasing the parallelism of task execution in such systems. It may deploy multiple executors on the same machine so that each of them will be allocated with a relatively smaller memory space. Although smaller memory may not cache as much data as larger ones do, this method can effectively decrease GC times, which has been widely applied in many systems [4] [5]. Such deployment of multiple executors also makes parallel granularity higher than the case of a single executor, which indeed accelerates the processing speed. However, the challenge is to design a fine-grained allocation policy as each executor's input data sets are not identical.. Furthermore, the memory demand in different computing stages of a process varies a lot over time. These two factors result in memory utilization imbalance among executors (i.e., tasks). Thus, we aim to fill in this gap by caching and managing intermediate data across multiple executors to improve and balance memory utilization.

In this work, we propose a new shared in-Memory cache layer, i.e., iMlayer, among these parallel executors which are co-hosted on the same slave machine in Apache Spark. The critical insight of iMlayer is to develop a novel eviction strategy, Next Re-reference Distance (NRD), to efficiently manage the shared cache space, i.e., iMCache, across executors to maximize overall cache hit rate as well as application performance. Specifically, this paper makes the following contribution:

- We empirically study the time-varying and imbalanced memory utilization when slave machines in Apache Spark are hosting multiple executors. We further investigate the performance impacts by applying different caching management policies.
- We design a shared memory cache space (i.e., iMCache), which is deployed between on-heap memory and local disk, to cache and manage intermediate data across multiple executors so that I/O operations can be decreased. Specifically, we propose a novel cache eviction policy



(a) Kmeans (b) SVD (c) PageRank

Fig. 1. Imbalanced real-time memory utilization between two executors running the same workloads.

named Next Re-reference Distance (NRD) to achieve effective cache eviction management in iMCache layer.

- We evaluate iMlayer based on the three representative workloads from HiBench [6]. Our results demonstrate that iMlayer with NRD eviction strategy improves the cache hit rate by 45%, 16% and 27%, effectively reducing the overall job runtime 47%, 43% and 38% compared to vanilla Spark, respectively.

The rest of this paper is organized as follows. Section II gives background and motivations. Section III describes the detailed system design and embedded eviction policy. Section IV presents experimental results. Section V reviews related work. Section VI concludes the paper.

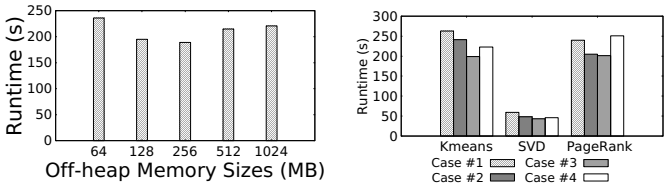
II. Motivation

To quantify memory utilization with multiple executors under various heap sizes and their respective impacts on performance, we conducted an empirical study with workloads from HiBench [6], a comprehensive benchmark suite. Spark-on-YARN mode is applied to flexibly configure executor numbers, CPU cores and memory sizes when running different workloads. We use Spark version 2.2 and Hadoop version 2.8.0 in the experiments. The default Least Recent Used (LRU) eviction policy is adapted for cache management.

1) Imbalanced Memory Utilization: Figure 1 depicts the memory usage traces of $Kmeans$, SVD and $PankRage$ under Case #2 in Table I, especially imbalanced memory utilization reflected by gaps between trace lines. Figure 1(a) shows the memory usages of two executors for $Kmeans$ fluctuate over time and dynamically exist gaps between them. In the case of SVD, the memory demand increases very fast right after the workload starts running and then stays in a relatively stable memory usage status. The reason is SVD's CPU-bound property requires less amount of memory storing data but more CPU resources. Moreover, we find there are consistent gaps between the two executors after 20_{th} seconds. The above observations demonstrate that (1) memory demands are dynamic over time; (2) memory usages between the two executors are imbalanced (shown as gaps between two lines). In particular, Figure1(c) shows $PankRage$ has a couple of peaks and valleys due to the suffering of GCs. These collapses between different executors result in low utilization and are detrimental to workload performance.

TABLE I
Resource Configurations

Case	#Executor	#Core	Memory	Total Resource
#1	1	8	8 GB	8 Cores & 8 GB
#2	2	4	4 GB	8 Cores & 8 GB
#3	4	2	2 GB	8 Cores & 8 GB
#4	8	1	1 GB	8 Cores & 8 GB



(a) Runtime with Sizes (b) Job runtime

Fig. 2. Figure (a) shows the runtime impact with different shared off-heap memory sizes. Figure (b) shows impact of parallelism in terms of job runtime.

Figure 2(a) depicts the job runtime achieved under different shared off-heap memory size configurations with two executors deployment model. It is obvious that the runtime decreases with the growth of the off-heap memory at the beginning stage, and the best performance comes with 256MB configuration. However, the performance dropped when the memory size is over 256MB, the reason behind this phenomenon laid on the fact that larger memory comes with more GC activities [3] leading to a longer runtime. This observation demonstrates a reasonable amount of shared memory can benefit the application performance while oversharing memory may incur performance degradation.

2) Impact of Parallelism: In our experiment, 8 CPU cores and 8GB memory were configured as the total available resource on each slave node to simulate a mainstream configuration. As shown in Table I, we set up 4 resource allocation cases. Figure 2(b) shows the runtimes of workloads ($Kmeans$, SVD and $PageRank$) under different resource allocation cases, i.e., with different execution parallelism. The results demonstrate the number of executors indeed has a significant influence on application performance. On one hand, allocating all resource to only one executor cannot guarantee the best performance compared to the model of running multiple executors. On the other hand, too many executors also may hurt the overall job runtime. Figure 2(b) also shows the best performance is achieved in Case #3, #2 and #3 for

$Kmeans$, SVD and $PageRank$, respectively. It demonstrates that running multiple executors on a single slave node is necessary and beneficial, especially when the memory size of individual machines keeps growing recently.

III. System Design

A. Architecture Overview

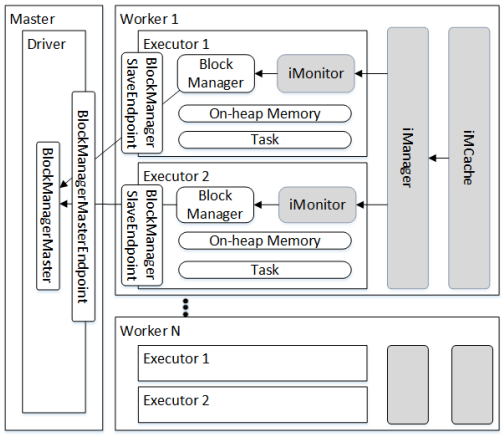


Fig. 3. System architecture of iMlayer.

Unlike the default memory management strategy in Apache Spark, iMlayer allows multiple executors on a single node to share specific off-heap memory space with each other. It aims to improve the hit rate of intermediate data blocks by caching and evicting data uniformly across multiple executors on the same hosting machine. Figure 3 shows the overall architecture of iMlayer.

- $iMCache$ is a cache memory space donated from individual executors and replaces the isolated off-heap memory region exclusive to each executor in vanilla Spark architecture. It is responsible for recording every data block's reference information, i.e., Next Re-reference Distance (NRD), which denotes blocks' re-referring possibilities in the future.
- $iManager$ is responsible for managing the data blocks cached in iMCache with a unified caching and evicting operation. By leveraging global data referring information, it evicts less possibly used data and makes more free space for the coming blocks in order to guarantee a higher overall hit rate.
- $iMonitor$ is running on each executor and responsible for maintaining the block information belongs to the individual executor, e.g., owner's executorID, storage location and reference statistics. It periodically reports the data block's location in iMCache to the original $BlockManager$.

B. Memory Sharing Policy in iMCache

Figure 4 depicts that iMlayer integrates these isolated executors via sharing the off-heap memory spaces from multiple

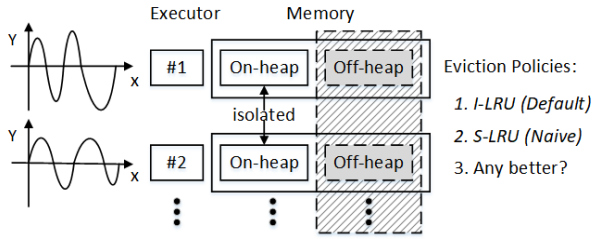


Fig. 4. Memory sharing among multiple executors.

executors. By default, each Spark executor is allocated with exclusive on-heap and off-heap memory space, respectively. LRU policy is applied to manage the data eviction in isolation. When multiple executors (e.g., Executor #1 and #2 in Figure 4) are deployed on a single node, the computational behaviors of different executors may not be identical so that the data access demand can vary a lot over time. For example, the left two graphs show data reference tendency, where x-axis represents the timeline and y-axis depicts dynamic data block access rates. In this case, such isolated memory management of individual executors could not always provide a good performance due to its management strategy.

To tackle this problem, we separate the original on-heap and off-heap memory space, and then combine off-heap memory segments from different executors together as a unified cache space (i.e., iMCache as shown in Figure 4), so that executors can share iMCache with each other. Then iMCache has to employ an efficient cache management policy to guarantee memory usage flexibility with multiple executors.

C. Memory Allocation of iMCache

We adopt a fuzzy model with multiple inputs and single output (MISO) to predict the off-heap memory size of each executor should denote. It is based on the relationship investigation of task completion time, total memory allocation and number of executors. The fuzzy model is often used to capture the complex relationship between resource allocation and a job's fine-grained execution progress. Given the periodic and repeatable feature of workloads, we design an off-line fuzzy model based on workloads' historical running logs. We formulate a fuzzy model as:

$$y_i(t) = R_i(r(t), e_m, s(t), \xi(t)).$$

This formula describes the relationship between input variables and output variable. The input variables are as following: $r(t)$ is the total memory allocation, e_m represents the number of executors deployed on a single node, the memory size of each executor should denote is expressed as $s(t)$ and the regression vector is $\xi(t)$. The output $y_i(t)$ is the average task completion time for each job. As many Spark applications have predictable structure in terms of computation and communication, iMlayer predicts the $s(t)$ for a specific job based on monitoring the similar job's previous runs [7].

To obtain the above model, some necessary parameters, i.e., $y(t), r(t), e_m, s(t)$ are first parsed from the workload's

Algorithm 1 Cache Donation

Input: app_type, app_log
Output: $iMCache$: memory size in terms of $s(t)$
1: // Building Fuzzy Model
2: **function** CALOPTMEM(app_type, app_log)
3: // Abstract parameters $y(t), r(t), e_m, s(t)$
4: mem_para = parse(app_log)
5: //Build Fuzzy model
6: R_i = bldfuzzy(mem_para)
7: // Get $s(t)$ of each executor running specific workload based on relation model R
8: $s(t)$ = getSharedMem(app_type, R)
9: set $iMCache$ according to $s(t)$
10: set $cache\_mem$ = allocMem($s(t)$)
11: **end function**

historical logs. A pattern model is built (as shown in Line 3-6 in Algorithm 1) to describe the relationship among those parameters following the proposed fuzzy model. In particular, different workloads may follow various patterns so that we use R to maintain all possible relation models. When a workload is deployed, our architecture decides the memory size that each executor should denote based on its type and performance model. After getting the parameter $s(t)$ for each executor, the size of iMCache is determined, which is shown by Line 4-10 in Algorithm 1.

D. Eviction Policy in iManager

Given the fact that the memory reference behaviors of Spark applications vary dramatically by different stages, jobs and applications, the naive cache eviction policy (i.e., LRU) may not guarantee a good and stable performance. As the shared cache memory space is typically much larger than the default exclusive memory of each executor, we propose a new eviction policy based on the next re-reference distance, which concept is adopted in [8] [9].

1) Next Re-reference Distance (NRD): We first introduce a metric, i.e., Next Re-reference Distance (NRD), to predict the possibility of a cache data block to be referenced again. Moreover, M-bit per cache block is used to denote one of its 2^M possible Next Re-reference Distance. NRD of each data block cached in iMCache dynamically gets updated once a block reference operation is requested. An NRD of zero implies that a cache block is predicted to be re-referenced in the near future while NRD of saturation (i.e., 2^m-1) means that a cache block is supposed to be re-referenced in a longer future. Quantitatively, data blocks with small NRDs are supposed to be re-referenced sooner than blocks with larger NRDs.

The key role of NRD is to prevent blocks with longer re-reference distance from occupying the limited cache space for too long time. Without any historical or external block reference information, NRD of each block is calculated by statical prediction. Since always assigning a 0 or 2^m NRD to newly inserted data block could not guarantee robustness across all block reference sequences. If the newly inserted data block is assigned with a 0 NRD, its re-reference distance will be updated so frequently that NRD fails to describe the real re-reference sequence. Oppositely, set newly data block's NRD to be 2^m causing longer cache occupation. So we assign the NRD of newly inserted data block to be 2^m-1, which value could guarantee the freshness of data blocks in cache. Additionally, always assigning 2^m-1 instead if 2^m brings more time to learn and improve the re-reference distance prediction.

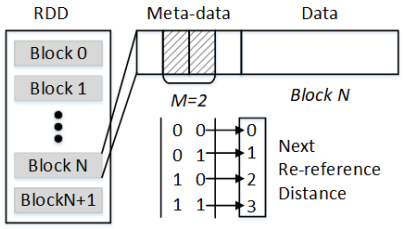


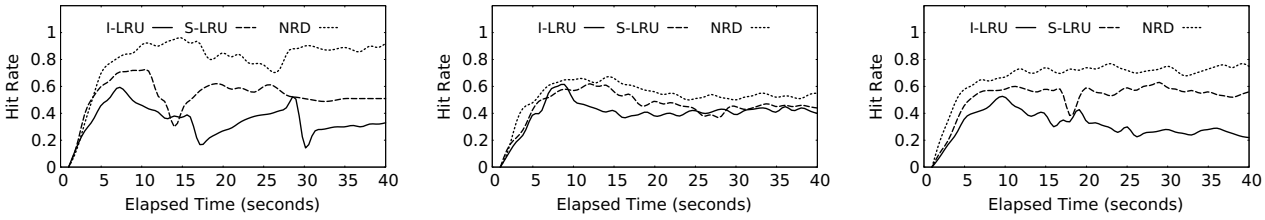
Fig. 5. NRD based data structure of an RDD block.

An eviction strategy in terms of NRD is implemented in iMCache. Figure 5 depicts an example of the data structure of an NRD based RDD. Each RDD contains several data blocks. A data block is highlighted to illustrate its components. For each data block shown in this example, it mainly has two parts: *Meta-data* is used to describe attributes of data block while *Data* contains the real contents of each block. In particular, the shadowed section belongs to the meta-data part is a 2-bit (*M*=2) marker. We use this section to denote four (2^2 = 4) possible situations of NRD (0, 1, 2 and 3). As is mentioned above, the NRD of each newly inserted data block is set to be *3*, which is denoted as *11* in the marker section of *Meta-data* part. Its metric gets decreased if this block is just re-referred. When cache space is full, data blocks with large NRD (*3* in this example) is the victim to be evicted while blocks whose NRD being *0* are highly kept in the cache space.

IV. EVALUATION

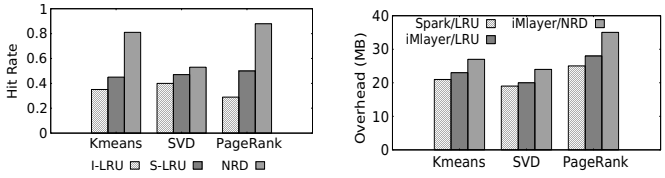
A. Experiment Setup

The evaluation testbed consists of 9 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 8 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. To test our prototype, we use three workloads from the HiBench big data benchmarking suite [6]. We run each workload 5 times to get an average performance so that accuracy could be guaranteed. We compare our method with the following two representative approaches. (1) I-LRU (Isolated Least Recently Used): the default management policy adopted by Spark, and merely evicting blocks based on the last time they are referenced. (2) S-LRU (Shared Least Recently Used): we introduce the iMCache and use LRU in this shared



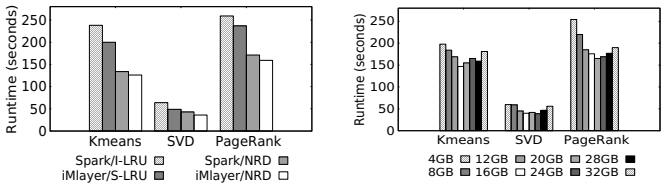
(a) Kmeans (b) SVD (c) PageRank

Fig. 6. Hit rate traces under the three eviction policies with different workloads.



(a) Hit rates. (b) Memory overhead.

Fig. 7. (a) shows hit rates among different eviction policies. (b) plots memory overhead of vanilla Spark and iMlayer.



(a) Eviction Policies. (b) iMCache Sizes.

Fig. 8. Performance impacts under different eviction policies and iMCache sizes.

memory space. In contrast, our proposed eviction policy (i.e., NRD) that takes NRD into consideration when managing cached blocks. Unless otherwise noted, all experiments were conducted on resource allocation of Case #2: two executors are deployed on the same worker node, each executor is allocated with 4 CPU cores and 4GB exclusive memory.

B. Improvement on Block Hit Rate

Figure 6 depicts the hit rate variation tendency within a typical period (0s-40s) at runtime. The results show that hit rates from different architectures keep increasing at the beginning stage due to more requested data blocks cached in memory. However, the hit rate decreases occasionally in later stages when the cache memory space is full and cache miss occurs. In this case, the eviction policy should evict several unnecessary blocks in order to load required ones, which explain the rise of hit rate in each curve. Figure 6 further demonstrates that iMlayer outperforms vanilla Spark by achieving higher hit rate and decreasing fluctuations in workloads' tendency. Similar to activities shown in average hit rate(illustrated in Figure 6(a)), hit rate changes of Kmeans and PageRank (plotted in Figure 6(a) and (c) respectively) are more obvious than SVD's changes (shown in Figure 6(b)).

Figure 7(a)) compares the average hit rates achieved by different cases. iMlayer with NRD obtains average 45%, 16% and 27% improvements respectively on these workloads compared to the vanilla Spark with default LRU. The results also demonstrate that iMlayer with LRU achieves 19%, 5% and 6% enhancements compared to the vanilla Spark architecture. A shared cache layer can effectively improve cached blocks hit rate by managing blocks uniformly, as it takes all executors' demands into consideration when evicting cached blocks.

C. Improvement on Job Runtime

Figure 8(a) shows job runtimes of three chosen workloads under various cases. In general, our work deployed with NRD achieves the best performance by obtaining 47%, 43% and 38% improvements compared to vanilla Spark with LRU for all above workloads. Particularly, vanilla Spark deployed with NRD outperforms it adopts LRU (Spark/I-LRU) with about 43%, 32% and 33% improvements respectively. It demonstrates that NRD eviction policy can dramatically improve the job level performance by efficiently managing the data blocks in limited memory space. When applying naive LRU policy in iMCache, it (i.e., S-LRU) outperforms vanilla Spark with LRU about 16%, 23% and 9% in job runtime. It further demonstrates the memory sharing policy can effectively improve job level performance by uniformly managing blocks.

Figure 8(b) investigates the performance impact of iMlayer with various iMcache sizes. In this experiment, we setup 8 cases with various iMCache sizes range from 4GB to 32GB. Moreover, NRD is adopted for cache management. In Figure 8(b), three workloads display similar performance behaviors in terms of the cache size impact. When iMcache size is small (e.g., 4GB, 8GB and 12GB), workloads' runtime is longer because small memory size could not cache enough data blocks so that more frequent I/O operations occurred between memory and local disk. In particular, note that these job runtimes start to drop down if the shared memory sizes are over-provisioned because too large memory size may lead to more GC operations.

D. Overhead Analysis

To analyze the overhead of iMlayer, we conduct experiments to measure the memory space consumptions in the eviction policy process while applying different approaches, i.e., vanilla Spark with LRU, iMlayer with LRU and iMlayer with NRD,

respectively. All these relevant processes are running on java virtual machine. We assign 4GB memory resources to each executor, which follows the practice experiences by existing works [10]. Figure 7(b) demonstrates that the memory overheads caused by different eviction processes are far less than typical executor's memory sizes (i.e., 1GB to 24GB [10]). The overheads of eviction processes deployed in iMlayer are a little higher than vanilla Spark with LRU. The reason is that more information about performance metrics has to be maintained for the shared cache management and NRD needs an extra memory space to record NRD of each data block. In particular, the average memory overheads caused by iMlayer with LRU and NRD are about 10% and 20% more than vanilla Spark with LRU, respectively, which are negligible to the whole system-level memory resource. Considering the performance improvement achieved by iMlayer, such memory overhead is an acceptable trade-off between space and time consumption.

V. Related Works

Memory management is a well-studied research topic while the application's performance has been widely improved by various memory strategies [8] [9] [11]. Spark is a typical in-memory computing platform, its performance can be improved by increasing execution parallelism. However, this strategy may introduce additional overheads by GC and I/O operations. Thus, many works focus on optimizing the deployment model of multiple executors/JVMs in Spark. Emerging JVM technologies such as heap ballooning [12] provides mechanisms to release committed memory from the virtual heap space. Tungsten [13] proposes a method to change memory management of JVM from on-heap to off-heap space. However, it is difficult to decide how much memory should be allocated to each executor and which data blocks are supposed to be evicted if any memory tension occurs. In contrast, iMlayer aims to build the same-functional shared memory layer for the local executors on the same slave machine, which offers a convenient way to manage cached data without a centralized server to save and synchronize data location information.

Moirai [14] focuses on cloud resource allocation with performance isolation in terms of requests per time window while Ginseng [15] is for memory pricing and auctioning cloud platforms. These works only focus on pricing memory resources for applications that have a specific shared cache memory server running on VMs. In contrast, iMlayer enables multiple executors to share a dedicated cache space and ensures a higher hit rate via a novel cache eviction policy, i.e., NRD. Moreover, our work focuses on managing the local memory resource on each slave machine, which is more scalable for most distributed systems.

VI. Conclusion

In this work, we propose a new shared in-memory cache layer, i.e., iMlayer, among parallel executors co-hosted on the same slave machine. It aims to improve the overall hit rate of data blocks by uniformly caching and evicting blocks across multiple executors. The key insight of iMlayer is a novel eviction strategy to efficiently manage iMCache among executors to maximize cache hit rate as well as application performance. By leveraging global data referring information, iMlayer evicts less possibly used data and makes more free space for coming blocks. The experimental results demonstrate that iMlayer with the new eviction strategy improves the cache hit rate by 45%, 16% and 27%, and effectively reduces the overall job runtime 47%, 43% and 38% compared to vanilla Spark, respectively.

VII. Acknowledgment

This work was supported by the NSF grants CCF-1908843 and CNS-2008265.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] Z. Liu and T. E. Ng, "Leaky buffer: A novel abstraction for relieving memory pressure from cluster data processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 128–140, 2017.

[3] T. Brecht, E. Arjomandi, C. Li, and H. Pham, "Controlling garbage collection and heap growth to reduce the execution time of java applications," in *ACM Sigplan Notices*, vol. 36, no. 11. ACM, 2001.

[4] H.-N. Vießmann, A. Šinkarovs, and S.-B. Scholz, "Extended memory reuse: An optimisation for reducing memory allocations," in *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, 2018, pp. 107–118.

[5] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 383–392.

[6] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[7] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 99–112.

[8] A. Barai, G. Chennupati, N. Santhi, A.-H. A. Badawy, and S. Eidenbenz, "Modeling shared cache performance of openmp programs using reuse distance," *arXiv preprint arXiv:1907.12666*, 2019.

[9] Y. Yuan, Y. Shen, W. Li, D. Yu, L. Yan, and Y. Wang, "Pr-lru: A novel buffer replacement algorithm based on the probability of reference for flash memory," *IEEE Access*, vol. 5, pp. 12 626–12 634, 2017.

[10] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017.

[11] D. Yang, W. Rang, D. Cheng, Y. Wang, J. Tian, and D. Tao, "Elastic executor provisioning for iterative workloads on apache spark," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 413–422.

[12] N. Bobroff, P. Westerink, and L. Fong, "Active control of memory for java virtual machines and applications." in *ICAC*, 2014, pp. 97–103.

[13] "Project tungsten: Bringing apache spark closer to bare metal," https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[14] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015.

[15] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven memory allocation," in *ACM SIGPLAN Notices*, vol. 49, no. 7. ACM, 2014, pp. 41–52.