

# LERC: Coordinated Cache Management for Data-Parallel Systems

Yinghao Yu<sup>†</sup>, Wei Wang<sup>†</sup>, Jun Zhang<sup>†</sup>, Khaled B. Letaief<sup>†‡</sup>

<sup>†</sup>Hong Kong University of Science and Technology

<sup>‡</sup>Hamad bin Khalifa University, Doha, Qatar

<sup>†</sup>{yyuau, weiwa, eejzhang, eekhaled}@ust.hk <sup>‡</sup>kletaief@hbku.edu.qa

**Abstract**—Memory caches are being aggressively used in today’s data-parallel frameworks such as Spark, Tez and Storm. By caching input and intermediate data in memory, compute tasks can witness speedup by orders of magnitude. To maximize the chance of in-memory data access, existing cache algorithms, be it recency- or frequency-based, settle on *cache hit ratio* as the optimization objective. However, unlike the conventional belief, we show in this paper that simply pursuing a higher cache hit ratio of individual data blocks does not necessarily translate into faster task completion in data-parallel environments. A data-parallel task typically depends on multiple input data blocks. Unless *all* of these blocks are cached in memory, no speedup will result. To capture this *all-or-nothing* property, we propose a more relevant metric, called *effective cache hit ratio*. Specifically, a cache hit of a data block is said to be *effective* if it can speed up a compute task. In order to optimize the effective cache hit ratio, we propose the Least Effective Reference Count (LERC) policy that persists the dependent blocks of a compute task *as a whole* in memory. We have implemented the LERC policy as a memory manager in Spark and evaluated its performance through Amazon EC2 deployment. Evaluation results demonstrate that LERC helps speed up data-parallel jobs by up to 37% compared with the widely employed least-recently-used (LRU) policy.

## I. INTRODUCTION

Memory cache plays a pivotal role in data-parallel frameworks, such as Spark [1], Tez [2], Piccolo [3] and Storm [4]. By caching input and intermediate data in memory, I/O-intensive jobs can be sped up by orders of magnitude [1], [5]. However, compared with stable storage, memory cache remains constrained in production clusters, and it is not possible to persist all data in memory [6]. Efficient cache management, therefore, becomes highly desirable for parallel data analytics.

Unlike caching in storage systems, operating systems and databases, cache management in data-parallel frameworks is characterized by the defining *all-or-nothing* requirement. That is, a compute task cannot be accelerated unless *all* of its dependent datasets, which we call *peers* in this paper, are cached in memory. In Hadoop [7] and Spark [8], data-parallel operations, such as *reduce*, *join* and *coalesce*, are typically performed on multiple datasets. Fig. 1 shows a Spark job consisting of two *coalesce* tasks. Task 1 (Task 2) fetches two data blocks *a* and *b* (blocks *c* and *d*) and coalesces them into a larger block *x* (block *y*). In this example, blocks *a* and *b*

This work was supported by the Hong Kong Research Grants Council under Grant No. 16200214.

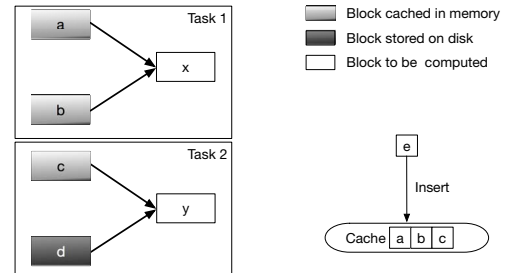


Fig. 1: A Spark job with two *coalesce* tasks. Task 1 (Task 2) coalesces data blocks *a* and *b* (data blocks *c* and *d*) into a larger block *x* (block *y*). Blocks *a*, *b* and *c* are initially persisted in a 3-entry cache, while block *d* is on disk. Another block *e* is to be inserted in the cache before Task 1 and Task 2 are scheduled. All the blocks are of unit size.

(blocks *c* and *d*) are *peers* of each other. Caching only one peer provides no benefit, as the task computation is bottlenecked by the reading of the other peer from disk.

However, existing cache management policies are *agnostic* to this all-or-nothing cache requirement. Instead, they simply optimize the block *cache hit ratio*, regardless of whether a cache hit can *effectively* speed up a compute task. For instance, the popular *least-recently-used* (LRU) policy employed by prevalent parallel frameworks [1], [2], [4], [9] caches the data blocks that have been recently used, counting on their future access to optimize the cache hit ratio. To illustrate that LRU may violate the all-or-nothing cache requirement for data-parallel tasks, we refer back to the example in Fig. 1. We assume that four unit-sized blocks *a*, *b*, *c* and *d* have been materialized and will be used by two *coalesce* tasks to compute blocks *x* and *y*. The cache can persist 3 blocks and initially holds blocks *a*, *b* and *c*. Suppose that another block *e* will be inserted to the cache, forcing one in-memory block to be evicted. Among the three cached blocks, block *c* is the *only* right choice of eviction, as caching it alone without its peer *d* speeds up no task. However, with the LRU policy, block *c* will remain in memory unless it is the least recently used. We shall show in Sec. II that the recently proposed Least Reference Count (LRC) policy [10] for data-parallel frameworks may run into a similar problem.

The all-or-nothing property calls for *coordinated* cache man-

agement in data-parallel clusters. In this work, we argue that cache policy should optimize a more relevant metric, which we call the *effective cache hit ratio*. In particular, we say a cache hit of a block is *effective* if it helps speed up a compute task, i.e., all the other peers of this block are in memory. Referring back to the previous example, caching block  $c$  without block  $d$  is ineffective as it speeds up no task. Intuitively, the effective cache hit ratio measures to what degree the all-or-nothing property can be retained and how likely compute tasks can get accelerated by having all their dependent datasets kept in memory.

In order to optimize the effective cache hit ratio, we design a coordinated cache management policy, called Least Effective Reference Count (LERC). LERC builds on our previously proposed LRC policy [10] yet retains the *all-or-nothing* property of data-parallel tasks. LERC always evicts the data block with the smallest *effective reference count*. The effective reference count is defined, for each data block  $b$ , as the number of *unmaterialized* blocks whose computation depends on block  $b$  and *can be sped up by caching*—meaning, the input datasets of the computation are all in memory.

We have implemented LERC in Spark and evaluated its performance in a 20-node Amazon EC2 cluster against representative workloads. Our prototype evaluation shows that LERC outperforms both LRU and LRC, reducing the job completion time by 37.0% and 18.6%, respectively. In addition, we have confirmed through experiments that compared to the widely adopted cache hit ratio, the proposed effective cache ratio serves as a more relevant metric to measure cache performance in data-parallel systems.

## II. INEFFICIENCY OF EXISTING CACHE POLICIES

In this section, we introduce the background information and motivate the need for coordinated cache management in data-parallel systems.

### A. Recency- and Frequency-Based Cache Replacement Policies

Traditional cache management policies optimize the cache hit ratio by evicting data blocks based on their access recency and/or frequency. LRU [11] and LFU [12] are the two representative algorithms.

- **Least Recently Used (LRU):** The LRU policy always evicts the data block that has not been accessed for the longest period of time. LRU bets on the *short-term* data popularity. That is, the recently accessed data is assumed to be likely used again in the near future. LRU is the default cache replacement policy in many popular parallel frameworks, such as Spark [1], Tez [2] and Storm [4].
- **Least Frequently Used (LFU):** The LFU policy always evicts the data block that has been accessed the least times. Unlike LRU, LFU bets on the *long-term* data popularity. That is, the data accessed frequently in the past will likely remain popular in the future.

Recency and frequency can also be used in combination, e.g., LRFU [13] and K-LRU [14]. All of these cache algorithms predict future data access based on historical information, and

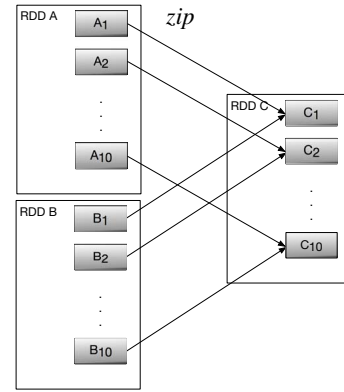


Fig. 2: DAG of the zip job in Spark with three RDDs  $A$ ,  $B$  and  $C$ . Each block in RDD  $C$  depends on two corresponding blocks from RDD  $A$  and RDD  $B$ .

are typically employed in operating systems, storage, database and web servers where the underlying data access pattern cannot be known *a priori*.

### B. Dependency-Aware Cache Management

In data-parallel frameworks such as Spark [1] and Tez [2], compute jobs have rich semantics of data dependency in the form of directed acyclic graphs (DAGs). These dependency DAGs dictate the underlying data access patterns, providing new opportunities for dependency-aware cache management.

For example, Fig. 2 shows the DAG of a Spark zip job [8]. Spark manages data through an abstraction called Resilient Distributed Datasets (RDDs) [1]. Each RDD is partitioned into multiple blocks across different machines. In Fig. 2, there are three RDDs  $A$ ,  $B$  and  $C$ , each consisting of 10 blocks. RDD  $C$  is a collection of key-value pairs obtained by zipping RDD  $A$  with RDD  $B$ . The computation of each block  $C_i$  depends on two blocks  $A_i$  (key) and  $B_i$  (value).

In Spark, the data dependency DAG is readily available to DAGScheduler upon a job submission. The recently proposed Least Reference Count (LRC) policy [10] takes advantage of this DAG information to determine which RDD block should be kept in memory.

**Least Reference Count (LRC):** The LRC policy always evicts the data block with the least *reference count*. The reference count is defined, for each block, as the number of unmaterialized blocks depending on it. In the example of Fig. 2, each block of RDD  $A$  and RDD  $B$  has reference count 1.

Intuitively, the higher the reference count an RDD block has, the more to-be-scheduled tasks depend on it, and the higher probability that this block will be used in the near future. In many applications, some RDD blocks would be used iteratively during the computation with much higher reference counts than others, e.g., the training datasets for cross-validation in machine learning [15]. With LRC, these blocks would have a higher chance to be cached in memory than others.

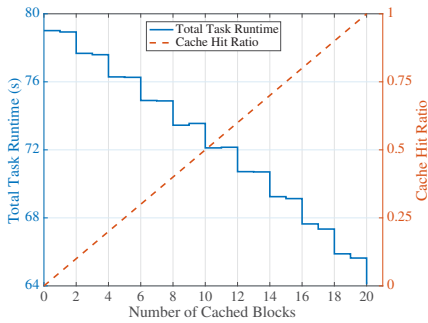


Fig. 3: Total task runtime of the example job in Fig. 2, with the cached RDD blocks increasing one at a time in the order of  $A_1, B_1, A_2, B_2, \dots, A_{10}, B_{10}$ .

### C. All-or-Nothing Cache Requirement

Prevalent cache algorithms, be it recency/frequency-based or dependency-aware, settle on the cache hit ratio as their optimization objective. However, the cache hit ratio fails to capture the *all-or-nothing* requirement of data-parallel tasks and is not directly linked to their computation performance. The computation of a data-parallel task usually depends on multiple data blocks, e.g., `join`, `coalesce` and `zip` in Spark [8]. A task cannot be sped up unless *all* its dependent blocks, which we call *peers* (e.g., block  $A_1$  and block  $B_1$  in Fig. 2), are cached in memory.

**Measurement study:** To demonstrate the all-or-nothing property in data-parallel systems, we ran the Spark `zip` job in an Amazon EC2 cluster with 10 `m4.large` instances [16]. The job DAG is illustrated in Fig. 2, where each of the two RDDs  $A$  and  $B$  is configured as 200 MB. We repeatedly run the `zip` job in rounds. In the first round, no data block is cached in memory. In each of the subsequent rounds, we add one more block to cache, following the caching order  $A_1, B_1, A_2, B_2, \dots, A_{10}, B_{10}$ . Eventually, all 20 blocks are cached in memory in the final round. In each round, we measure the cache hit ratio and the total runtime of all 10 tasks. Fig. 3 depicts our measurement results against the number of RDD blocks in memory. Despite the linearly growing cache hit ratio with more in-memory blocks, the task completion time is notably reduced only after the two peering blocks  $A_i$  and  $B_i$  have been cached.

**Inefficiency of existing cache policies:** The first step to meet the all-or-nothing cache requirement is to identify which blocks are peers with each other and should be cached together as a whole. This information can only be learned from the data dependency DAG, but it has not been well explored in the literature. Many existing cache policies, such as LRU and LFU, are *oblivious* to the DAG information, and are unable to retain the all-or-nothing property. The recently proposed LRC [10] policy, though DAG-aware, does not differentiate the peering blocks and hence suffer from the same inefficiency as LRU. By referring back to the previous example in Fig. 1, we see that blocks  $a, b$  and  $c$  have the same reference count of 1 and would have an equal chance to get evicted by LRC. In other

words, LRC would evict a wrong block (other than  $c$ ) with probability 67%. To the best of our knowledge, PACMan [5] is the only work that tries to meet the all-or-nothing requirement for cache management in parallel clusters. However, PACMan is *agnostic* to the semantics of job DAGs, and its objective is to speed up data sharing across different jobs by caching *complete* datasets (HDFS files). Since PACMan only retains the all-or-nothing property for each individual dataset, if a job depends on multiple datasets, completely caching a subset of them provides no performance benefits.

In summary, we have shown through a toy example and measurement study that optimizing the cache hit ratio—the conventional performance metric employed by existing cache algorithms—does not necessarily speed up data-parallel computation owing to its all-or-nothing cache requirement. We shall design a new cache management policy that meets this requirement based on the peer information extracted from the data dependency DAGs.

## III. LEAST EFFECTIVE REFERENCE COUNT

In this section, we define the *effective cache hit ratio* as a more relevant cache performance metric for data-parallel tasks. To optimize this new metric, we propose a coordinated cache management policy, called Least Effective Reference Count (LERC). We also present our implementation of LERC as a cache manager in Spark.

### A. Effective Cache Hit Ratio

In a nutshell, a cache hit of a block is *effective* if it can speed up the computation. In data-parallel frameworks, a compute task typically depends on multiple data blocks. We call all these blocks *peers* with respect to (w.r.t.) this task. Caching only a subset of peering blocks provides no speedup for the task. Formally, we have the following definition.

**Definition 1** (Effective Cache Hit). *For a running task, the cache hit of a dependent block is effective if its peers w.r.t. the task are all in memory.*

We now define the *effective cache hit ratio* as the number of effective cache hits normalized by the total number of block accesses. This ratio directly measures how much compute tasks can benefit from data caching.

By referring back to the previous example in Fig. 1, each of the four blocks  $a, b, c$  and  $d$  is used only once in the computation. Initially, blocks  $a, b$  and  $c$  are in memory. Since block  $d$  is on disk, evicting its peer  $c$  has no impact on the effective cache hit ratio, which remains at 50% (i.e., two effective cache hits for  $a, b$  out of 4 block accesses). Evicting block  $a$  (or  $b$ ), on the other hand, results in no effective cache hit.

The effective cache hit ratio directly measures the performance of a cache algorithm. Algorithms that optimize the traditional cache hit ratio may not perform well in terms of this new metric. Back to the previous example, the LRC policy [10] evicts each of the three in-memory blocks,  $a, b$  and  $c$ , with an equal probability (cf. Sec. II-C), leading to a low effective

cache hit ratio of  $\frac{1}{3} \times 50\% + \frac{2}{3} \times 0\% = 16.7\%$  in expectation. The LRU policy can be even worse as the effective hit ratio is 0, unless block  $c$  is the least recently used.

A naive approach to optimize the effective cache hit ratio is the *sticky eviction* policy. That is, the peering data blocks stick to each other and are evicted as a whole if any of them is not in memory. However, such a sticky policy can be highly inefficient. A data block might be a shared input of multiple tasks. Caching the block, even though not helping speed up one task (i.e., not all peers w.r.t. the task are in memory), may benefit another. With the sticky policy, this block would surely be evicted, and no task can be sped up. We propose our solution in the next subsection.

### B. Least Effective Reference Count Caching

We start with the definition of the *effective reference count*. In data-parallel frameworks, a block may be referenced by many tasks (i.e., used as input). We differentiate *effective reference* from the others by the following definition.

**Definition 2** (Effective Reference). *Let block  $b$  be referenced by task  $t$ . We say this reference is effective if task  $t$ 's dependent blocks, if computed, are all cached in memory.*

For a data block, the *effective reference count* is simply the number of its effective references, which, intuitively, measures how many downstream tasks can be sped up by caching this block. Based on this definition, we present the *Least Effective Reference Count* (LERC) policy as follows.

**Least Effective Reference Count (LERC):** *The LERC policy evicts the block whose effective reference count is the smallest.*

As a running example, we refer back to Fig. 1. Each of blocks  $a$  and  $b$  has effective reference count 1, while block  $c$  has no effective reference count (its reference by Task 2 is not effective as block  $d$  is not in memory). With the LERC policy, block  $c$  is evicted, which is the optimal decision in this example.

LERC has two desirable properties. First, by prioritizing blocks with large effective reference counts, LERC is able to retain the peering blocks as entities as much as possible, through which the effective cache hit ratio is optimized. Second, the effective reference count can be easily extracted from the job DAG that is readily available to the scheduler.

### C. Spark Implementation

Implementing LERC in data-parallel frameworks, such as Spark, poses some non-trivial challenges. Maintaining a block's effective reference count requires knowing the caching status of its peers on different workers. Synchronizing each block's caching status across workers may incur significant communication overhead. The first approach is to maintain the caching status profile in a centralized manner at the Spark driver. When a block is evicted from memory, the worker reports to the driver. The driver updates the effective reference counts of other peering blocks and broadcasts the updates to all workers. Alternatively, we can let workers maintain the caching status

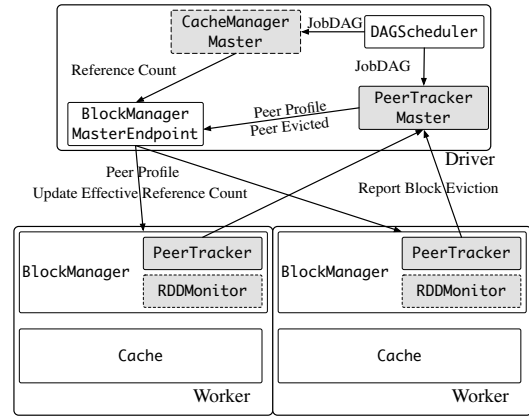


Fig. 4: Overall system architecture of the proposed coordinated cache manager in Spark. Our implementation modules are highlighted as shaded boxes with solid borderlines. Shaded boxes with dashed borderlines are the legacy implementations of the LRC work [10].

in a distributed manner. Upon a block eviction on a worker, all the other workers must be notified immediately to update the effective reference counts. Both approaches broadcast a large amount of information across the network, and are expensive to implement.

We next present our implementation to address this problem.

**Architecture overview.** Fig. 4 presents an architecture overview of our LERC cache manager in Spark. The CacheManagerMaster and RDDMonitor are the legacy modules of our LRC implementation [10]. The CacheManagerMaster in the driver parses the reference count profile and maintains it together with the RDDMonitors in workers. In this paper, we have implemented two components: (1) PeerTrackerMaster that profiles the peer information in the job DAG obtained from the DAGScheduler, and (2) PeerTracker in each worker that reports the status of the peer blocks when necessary.

**Workflow.** When the Spark driver is launched, the PeerTrackerMaster is initialized together with other components in the driver. Upon a job submission, the PeerTrackerMaster obtains the job DAG from the DAGScheduler and parses out the peer information. The peer information profile is then broadcast via the BlockManagerMasterEndpoint to all PeerTrackers in the cluster. The PeerTracker initially labels every *peer-group* (all peering blocks w.r.t. a task) as “complete,” and then changes it to “incomplete” if any of its materialized block has been evicted from memory. Upon a block eviction, the PeerTracker first checks whether this block belongs to any “complete” peer-group. If so, the effective reference counts of blocks in these peer-groups should be updated due to this block eviction. A block eviction report is sent to the PeerTrackerMaster and broadcast to other workers. Upon receiving a block eviction message, the PeerTracker scans all the “complete” peer-groups. If there are “complete”

peer-groups containing the evicted block, the PeerTracker labels them as “incomplete” and informs the RDDMonitor to decrease the effective reference counts of the peering blocks accordingly.

**Communication Overhead.** Our implementation minimizes the number of required communication messages. To see this, we first show that at most one broadcasting is triggered for the entire group of peer blocks in our implementation. By labeling the “complete” peer-groups locally in the PeerTrackers, there is no need to track the caching status of each peer block separately. Only the block eviction in a “complete” peer-group triggers the updating of the peering blocks’ effective reference counts. Once a block eviction message is broadcast, the peer-group becomes “incomplete”, and no more updating messages will be required for this peer-group. We next show that it is necessary to broadcast at least one block eviction message for each group, if any of its peer has been evicted. Since it is possible that some of the evicted block’s peers have not been computed yet, the block eviction message should be broadcast to all workers instead of those with the evicted block’s peers.

#### IV. EVALUATION

In this section, we evaluate the efficacy of the LERC cache manager with synthesized workloads in Amazon EC2. We also confirm that the effective cache hit ratio is a more relevant metric of cache performance than the widely used cache hit ratio.

**Cluster deployment.** Our implementation is based on Spark 1.6.1. In order to highlight the advantage of memory locality, we disabled the OS page cache by triggering direct disk I/O from/to the hard disk. We deployed an Amazon EC2 [16] cluster with 20 nodes of type `m4.large`, each with a dual-core 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processor and 8 GB memory.

**Experiment settings.** In the experiment, we simulated 10 tenants submitting Spark `zip` [8] jobs in parallel. In each job, two files of 400 MB are firstly partitioned into 100 blocks and stored in the 20 machines. Since the 10 jobs operate on different files, the total size of the input blocks is  $400 \text{ MB} \times 2 \times 10 = 8 \text{ GB}$ . The cache manager will decide which blocks should be stored on disk when the cache is full. After that, 100 zip tasks are scheduled for each job to zip the two files into 100 key-value pairs, where the keys are the data of the first file, and the values are the data of the second file. Notice that only when both the key and value are cached in memory will a zip task be sped up. We measure the total completion time of the 10 jobs to compare the performance of different cache policies.

##### A. Job Completion Time

We conducted the experiment using three cache replacement policies, i.e., LRU, LRC, and LERC, with different memory cache sizes. In particular, we configured `storage.memoryFraction` in the legacy Spark to throttle the memory used for RDD caching to a given size. We

measured the total experiment runtime, i.e., the make span of the 10 submitted jobs. We repeated each experiment 10 times and depict the average results in Fig. 5. The error bars show the maximum and minimum completion time in the 10 runs.

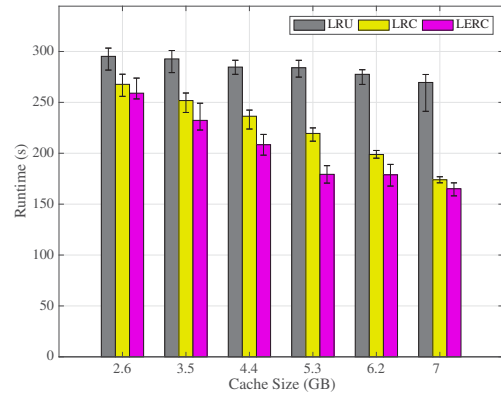


Fig. 5: Experiment runtime under the three cache management policies with different cache sizes.

To our expectation, as the size of RDD cache increases, the total experiment runtime decreases under all the three cache policies. In all cases, LRC consistently outperforms the default LRU policy. LERC further reduces the experiment completion time over LRC. When the cache size is 5.3 GB, for instance, the average runtimes under the three policies are 284 s (LRU), 220 s (LRC) and 179 s (LERC), respectively. The LERC policy speeds up job completion by 37.0% and 18.6% compared to the LRU and LRC policies, respectively.

##### B. Effective Cache Hit Ratio

We now evaluate the relevance of the two metrics, i.e., the effective cache hit ratio and the cache hit ratio, in measuring the cache performance in data-parallel clusters. Both of the two metrics were recorded in the previous experiments. The results are shown in Fig. 6 and Fig. 7.

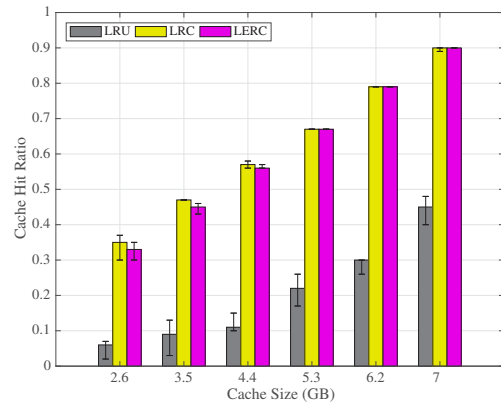


Fig. 6: Cache hit ratio under the three cache management policies with different cache sizes.

Fig. 6 shows that LRC achieves the highest cache hit ratio, while LERC closely follows. This is because LRC aims to optimize the cache hit ratio, and it outperforms LRU by taking

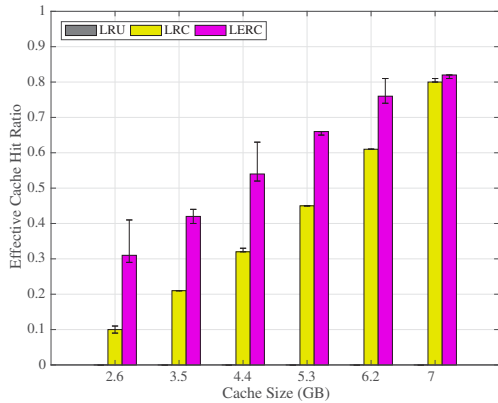


Fig. 7: Effective cache hit ratio under the three cache management policies with different cache sizes. Notice that the effective cache hit ratio is near zero under the LRU policy.

advantage of the DAG information. LERC also takes use of this information, but it gives up on retaining those ineffective cache hits that are unable to speed up tasks. It is for this reason that the cache hit ratio is slightly compromised.

Fig. 7 shows that LERC always achieves the highest effective cache hit ratio. The smaller the cache size is, the more advantageous LERC is. We therefore conclude that LERC is able to make the best use of the constrained cache resources to effectively speed up compute tasks. As the available cache space increases, LRC is more likely to retain an entire peer-group in memory, and its effective cache hit ratio becomes closer to that of LERC. On the other hand, when the cache size is small, many peer blocks will have to be evicted out of memory. Since LERC needs to maintain the effective reference count, a salient communication overhead is incurred. For this reason, in Fig. 5, LERC does not save much job runtime compared with LRC when the cache volume is relatively small, even though it achieves a much higher effective cache hit ratio. As the cache size increases, less communication cost is incurred, and the performance in the effective cache ratio gets in line with the job runtime.

Notice that the effective cache hit ratio of LRU is always near zero in our experiment. Since the tenants submit their jobs in parallel, the first file (keys required by zip) of each job is highly likely to be replaced by the second file (values required by zip) of other jobs arriving *later* under the LRU policy. Therefore, when the zip starts, only the values are cached, resulting in zero effective cache hit.

We conclude this section by noting that although LRC achieves the highest cache hit ratio throughout the experiments, it incurs longer job completion time compared with LERC. On the other hand, LERC achieves a higher effective cache hit ratio over LRC. Its relative advantage in the effective cache hit ratio is consistent with that in the job completion time (when the communication overhead is not a concern). We therefore draw the conclusion that the effective cache hit ratio serves as a more relevant metric for cache performance in our experiments.

## V. CONCLUSIONS

In this paper, we have identified the defining all-or-nothing property in data-parallel systems, i.e., a compute task can only be sped up when *all* of its input datasets are cached in memory. Existing cache management policies are agnostic to this all-or-nothing requirement. Instead, they settle on the cache hit ratio as their optimization objective and hence cannot effectively reduce the task runtime in data-parallel environments. To address this problem, we have proposed the effective cache hit ratio as a more relevant cache performance metric for data-parallel tasks. We have designed a coordinated cache policy, Least Effective Reference Count (LERC), that optimizes this metric by evicting the data blocks with the smallest effective reference count. We have implemented LERC as a pluggable cache manager in Spark, and evaluated its performance through Amazon EC2 deployments. Experimental results validated the relevance of the effective cache hit ratio and the performance advantage of the LERC policy. Compared with the popular LRU policy and the recently proposed LRC policy, LERC speeds up the job completion by up to 37% and 19%, respectively.

## REFERENCES

- [1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012.
- [2] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD*, Melbourne, Australia, 2015.
- [3] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables." in *Proc. USENIX OSDI*, vol. 10, 2010.
- [4] Storm, "<http://storm.apache.org/>."
- [5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: coordinated memory caching for parallel jobs," in *Proc. USENIX OSDI*, 2012.
- [6] S. Sharma, "Expanded cloud plumes hiding big data ecosystem," *Future Generation Computer Systems*, vol. 59, pp. 63–92, 2016.
- [7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Systems and Technologies, 2010*, 2010.
- [8] Spark API Documentation, "<https://spark.apache.org/docs/latest/api.html>."
- [9] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory hadoop jobs," in *Proc. the VLDB Endowment*, vol. 5, no. 12, 2012.
- [10] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LRC: Dependency-aware cache management for data analytics clusters," in *Proc. IEEE INFOCOM*, Atlanta, GA, USA, 2017.
- [11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.
- [12] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, 1971.
- [13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.
- [14] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.
- [15] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. International Joint Conf. Artificial Intell. (IJCAI)*, vol. 14, no. 2, 1995, pp. 1137–1145.
- [16] Amazon Elastic Compute Cloud, "<https://aws.amazon.com/ec2/>," 2016.