Towards Dependency-Aware Cache Management for Data Analytics Applications

Yinghao Yu, *Student Member, IEEE*, Chengliang Zhang, *Student Member, IEEE*, Wei Wang, *Member, IEEE*, Jun Zhang, *Senior Member, IEEE*, and Khaled Ben Letaief, *Fellow, IEEE*

Abstract—Memory caches are being used aggressively in today's data analytics systems such as Spark, Tez, and Piccolo. The significant performance impact of caches and their limited sizes call for efficient cache management in data analytics clusters. However, prevalent data analytics systems employ rather simple cache management policies—notably Least Recently Used (LRU) and Least Frequently Used (LFU)—that are *oblivious* to the application semantics of data dependency, expressed as directed acyclic graphs (DAGs). Without this knowledge, cache management can, at best, be performed by "guessing" the future data access patterns based on history, which frequently results in inefficient, erroneous caching with a low hit rate and a long response time. Worse still, the lack of data dependency knowledge makes it impossible to retain the *all-or-nothing* cache property of cluster applications, in that a compute task cannot be sped up unless all the dependent data has been kept in the main memory. In this paper, we propose a novel cache replacement policy, named Least Reference Count (LRC), which exploits the application's data dependency information to optimize the cache management. LRC keeps track of the *reference count* of each data block, defined as the number of dependent child blocks that have not been computed yet, and always evicts the block with the smallest reference count. Furthermore, we incorporate the all-or-nothing requirement into LRC by coordinately managing the reference counts of all the input data blocks for the same computation. We demonstrate the efficacy of LRC through both empirical analysis and cluster deployments against popular benchmarking workloads. Our Spark implementation shows that, the proposed policies well address the all-or-nothing requirement and significantly improve the cache performance. Compared with LRU and a recently proposed caching policy called MEMTUNE, LRC improves the caching performance of typical workloads in production clusters by 22 and 284 percent, respectively.

Index Terms—Cloud computing, data analytics system, cache management, dependency-awareness, all-or-nothing caching

1 Introduction

Data analytics systems are undergoing a fundamental shift to in-memory computation. The ever-growing demand for interactive, iterative data analytics and the stalling speed of disk I/O force the system to cache a large volume of data in memory to provide low latency [3], [4], [5], [6]. Despite the increasing availability of high-RAM machines, the size of in-memory caches remains orders-of-magnitude smaller than that of the data. Efficient cache management, therefore, plays a key role in large-scale data analytics systems.

Caching is a classical problem that has been well studied in storage systems [6], [7], databases [8], [9], operating systems [10], and web servers [11], [12]. Nevertheless, caching in data analytics clusters has two defining aspects that differentiate it from that in those previous systems. First, data analytics applications have clear semantics of data dependency, expressed as directed acyclic graphs (DAGs) of compute tasks. The DAG is readily available to the cluster scheduler upon the submission of an application. The DAG sketches out the application's task execution plan which dictates the *future* data access pattern, i.e., how the data will be computed and reused as input of descendant tasks. For example, Fig. 1 depicts the DAG of a Spark job, where the four `map` tasks respectively take four data blocks as input and transform them to four intermediate datasets, which are then fed to the two `zip` tasks to compute the final output.

Second, caching in data analytics clusters is characterized by the *all-or-nothing* requirement at two levels. At a low level, for a compute task with multiple input data blocks—which we call *peers* in this paper—the task can only be sped up when *all* of those peers are cached in memory. At a high level, for a *compute stage* where multiple tasks are scheduled in parallel, the stage completion time can only be reduced when *all* those tasks get sped up by caching—meaning, all the input data blocks in that stage should be kept in memory. In Fig. 1, blocks A_2 and C_2 (blocks B_2 and D_2) are input peers of a `zip` task to compute block E (block F). Caching only one peer provides no benefit because the task will still need to read the other peer from disk. Furthermore, if the two parallel `zip` tasks are scheduled in the same stage, the

- *Y. Yu and K. B. Letaief are with the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {yyuau, eekhaled}@ust.hk.*
- *C. Zhang and W. Wang are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {czhangbn, weiwa}@ust.hk.*
- *J. Zhang is with the Department of Electronic and Information Engineering, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: jun-eie.zhang@polyu.edu.hk.*

Manuscript received 30 Oct. 2018; revised 4 Sept. 2019; accepted 26 Sept. 2019. Date of publication 1 Oct. 2019; date of current version 8 Mar. 2022. (Corresponding author: Yinghao Yu.)
Recommended for acceptance by H. Huang.
Digital Object Identifier no. 10.1109/TCC.2019.2945015

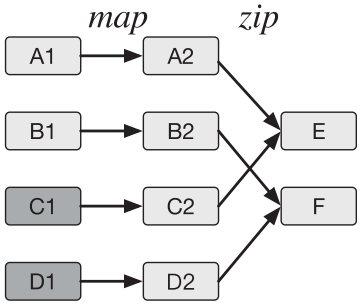


Fig. 1. The data dependency DAG of an example application. Each block represents a dataset. Two of the input blocks, A_1 and B_1, are cached in memory, while the other two input blocks, C_1 and D_1, are stored on disk. Blocks A_2, B_2, C_2, and D_2 are intermediate datasets derived from each of the four input datasets via mapping. Block E (F) is the final result derived from both A_2 and C_2 (B_2 and D_2).

stage completion time cannot be reduced unless all four input blocks A_2, B_2, C_2, and D_2 are cached in memory.

However, existing caching policies in prevalent data analytic systems [4], [13], [14] are *oblivious* to the data dependency DAGs of applications. Instead, they predict application-specific data access patterns based on the historical information, notably the frequency and recency of data accesses. For example, Spark [4] and Tez [15] employ the classical LRU policy to evict data blocks that are *least recently used* in the presence of high memory pressure. As we shall show in Section 2, using recency and frequency to predict future data accesses can be highly inaccurate and may result in a low cache hit rate. Moreover, without the knowledge of the data dependency DAGs, it is not possible to meet the all-or-nothing caching requirement.

In this paper, we investigate how the application semantics of data dependency should be exploited to optimize cache management. Ideally, the solution should take full use of the DAG information and can be easily implemented for a wide range of in-memory analytics frameworks. Our response is a novel dependency-aware cache management policy, which we call Least Reference Count (LRC). The reference count is defined, for each data block, as the number of *unmaterialized child blocks* derived from it, which is equivalent to the number of tasks whose computations depend on that block but have not been executed yet. As the name suggests, the LRC policy always evicts the data block whose *reference count* is the smallest. With minor modifications, the LRC policy can easily meet the all-or-nothing caching requirement at both the task and stage levels, using *conservative* and *aggressive* eviction strategies, respectively (details in Section 3.2).

LRC provides benefits over existing cache management policies in four aspects. First, LRC can timely detect *inactive* data blocks with *zero* reference count. Such blocks are unlikely to be used again in the remaining computations[1] and can be safely evicted from the memory. Second, compared with the historical information such as the block access recency and frequency, the reference count serves as a more accurate indicator of the likelihood of future data access. Intuitively, the higher the reference count a block has, the more child blocks depend on it, and the more likely the block is needed in the downstream computation. We show through empirical studies in Section 2 that caching data blocks with the highest reference count increases the hit ratio by up to 119 percent as opposed to caching the most recently used data. Third, LRC can be generalized to meet the all-or-nothing requirements by coordinately managing the reference counts of data blocks at task and stage levels. Finally, the fact that the reference count can be accurately tracked at runtime with negligible overhead makes LRC a lightweight cache manager for data analytics systems.

We have prototyped LRC as a pluggable memory manager in Spark (details in Section 4). To evaluate the efficacy of LRC in production clusters, we conducted experiment through Amazon EC2 deployment against SparkBench [16], a popular benchmarking suite for Spark. Experimental results show that LRC retains the same application performance using only 40 percent of cache space as compared to LRU, the default cache management policy used in Spark. When operating with the same memory footprint, LRC is capable of reducing the application runtime by up to 60 percent. Our implementation can be easily adapted to other DAG-based data analytics systems and can also be extended to multi-tenant cache sharing systems such as Alluxio [17].

Two preliminary versions of this article have appeared in [1] and [2]. In this version, we make substantial improvements over the two previous submissions. First, we have obtained a deeper understanding of the all-or-nothing caching requirement of data analytics systems at two levels (Section 2.4), i.e., the task and stage level requirements, which generally applies to all DAG-based analytics systems. In comparison, the conference papers either do not respect it at all [1] or only acknowledge the task-level requirement [2]. Second, we have strengthened the evaluation of LRC using trace-driven simulations (Section 5.3) with the recently released production trace [18] from the Alibaba Group. We have demonstrated the prominent performance advantages of LRC policies against LRU and another related work MEMTUNE [19] in production workloads. We have also open-sourced our simulator [20] for ease of other researchers to test their own cache policies against the Alibaba trace. Third, we have provided motivating examples as well as extensive experimental results (Section 5.1), demonstrating the need to take care of the all-or-nothing requirement at two levels. Fourth, we have extended the LRC cache manager to support the two-level caching requirement in practical systems (Section 4.1) and provided operational guidance on how to switch between the two based on the available resources. Lastly, we have expanded the discussions on the implementation overhead of LRC (Section 4.2) and covered more related works in the literature review (Section 6).

2 Inefficiency of Existing Cache Management Policies

In this section, we present the background information and motivate the need for a new cache management policy through empirical studies. Unless otherwise specified, we shall limit our discussion to the context of Spark [4]. However, nothing precludes applying the discussion to other data analytics frameworks such as Tez [15] and Storm [13].

1. Unless re-computation is needed due to machine failures or stragglers.

2.1 Semantics of Data Dependency

Cluster applications such as machine learning, web search, and social network typically consist of complex *workflows*, which are specified as directed acyclic graphs (DAGs) of compute tasks. For example, in Spark, data is managed through an easy-to-use memory abstraction called Resilient Distributed Datasets (RDDs) [4]. An RDD is a collection of immutable datasets partitioned across a group of machines. Each machine stores a subset of RDD partitions (blocks), either in-memory or on-disk. An RDD can be created directly from a file in a distributed storage system (e.g., HDFS [21], Amazon S3 [22], and Tachyon [6]), or it can be derived from other RDDs through a user-defined transformation. This process leads to a *data flow* model, in which the programmer defines how the RDDs are transformed from one to another. The programmer can use RDDs in *actions*, which are operations that either return a value to the application, or export data to a stable storage system. Spark computes RDDs lazily, until the first time they are used in actions. An action then triggers the execution of a *compute job* along with its DAG of RDDs. Fig. 1 illustrates such a job DAG in Spark.

Whenever a job is submitted to the Spark driver, its DAG of RDDs becomes readily available to a driver component, `DAGScheduler` [4]. The `DAGScheduler` then traverses the job DAG using depth-first search (DFS) and continuously submits *runnable* tasks (i.e., those whose parent RDDs have all been computed) to the cluster scheduler to compute unmaterialized RDDs. In particular, the tasks used to compute blocks from the same RDD form a *stage*. The cache manager can easily retrieve the DAG information from the `DAGScheduler`. This information sheds light into the underlying data access patterns, based on which the cache manager can decide which RDD block should be kept in memory.

It is worth emphasizing that the availability of data dependency DAGs of compute jobs is not limited to Spark, but generally found in other parallel frameworks such as Apache Tez [15]: the Tez programming API allows the programmer to explicitly define the workflow DAG of an application, which is readily available to the Tez scheduler beforehand.

2.2 Recency- and Frequency-Based Cache Management

Despite the availability of the application's DAG, prevalent cache management policies are agnostic to this data dependency information. Instead, they simply predict data access patterns based on the historical information, namely the *recency* and *frequency* of data accesses.

- *Least Recently Used (LRU)*: The LRU policy [23] makes room for new data by evicting the cached blocks that have not been accessed for the longest period of time. LRU is the *de facto* cache management policy employed in today's in-memory data analytics systems [4], [6], [13], [15]. It predicts the access pattern based on the short-term data popularity: the recently accessed data is assumed to be likely used again in the near future.
- *Least Frequently Used (LFU)*: The LFU policy [10] keeps track of the access frequency of each data block, and the one that has been accessed the least frequently has the highest priority to be evicted. Unlike LRU, LFU predicts the access pattern based on the long-term data popularity, meaning the frequently accessed data is assumed to be likely used again in the future.

Both LRU and LFU are easy to implement. However, their obliviousness to the application semantics of data dependency frequently results in inefficient, even erroneous, cache decisions, as we show next.

TABLE 1
An Overview of SparkBench Suite [16]

Application Type	Workload
Machine Learning	*Logistic Regression* *Support Vector Machine (SVM)* *Matrix Factorization*
Graph Computation	*Page Rank* *SVD Plus Plus* *Triangle Count*
SQL Queries	*Hive* *RDD Relation*
Streaming Workloads	*Twitter Tag* *Page View*
Other Workloads	*Connected Component* *Strongly Connected Component* *Shortest Paths* *Label Propagation* *Pregel Operation*

2.3 Data Access Pattern

To illustrate the need for being dependency-aware, we characterize the data access patterns and their correlations to the dependency DAGs in typical analytics benchmarks through empirical studies. We show that simply relying on the recency and frequency information for cache management would waste a large portion of memory to persist *inactive data* that will never be used in downstream computations.

Methodology. We ran SparkBench [16], a comprehensive benchmarking suite, in an Amazon EC2 [24] cluster consisting of 10 `m4.large` instances. We measured the memory footprints and characterized the data access patterns of 15 applications in SparkBench, including machine learning, graph computation, SQL queries, streaming, etc. Table 1 summarizes the workload suite we used in our empirical studies.

Data Access Patterns. Our experiments have identified two common access patterns across the benchmark applications.

(1) *Most data goes inactive quickly and will never be referenced again*.Fig. 2 shows the distribution of inactive data cached in memory throughout the execution of 15 applications. We find that the amount of inactive data accounts for a large portion of memory footprint during the execution, with the median and 95th percentile being 77 and 99 percent, respectively. The dominance of inactive data blocks is in line with Spark's data flow model where the intermediate datasets are likely to be consumed by "nearby" computations in the DAG, and therefore have short life

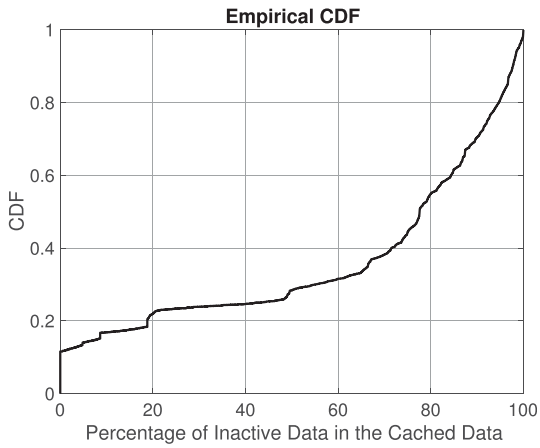


Fig. 2. Distribution of inactive data cached during the execution of SparkBench [16].

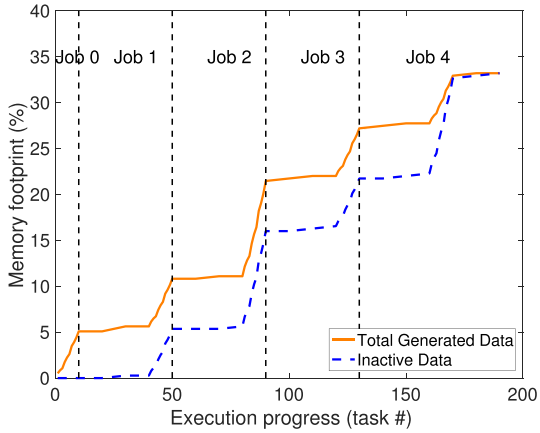


Fig. 3. Memory footprints of intermediate data generated and inactive blocks cached during the execution of *Connected Component*, an application in SparkBench [16].

cycles. Keeping inactive data long in memory wastes cache spaces, inevitably resulting in a low hit rate.

(2) *Data goes inactive in* waves, *in alignment with the generation of new data*. We further micro-benchmarked the memory footprint of total generated data against that of inactive data blocks cached during the execution of each application. Fig. 3 shows the results for a representative application that computes the connected components of a given graph. The x-axis measures the execution progress, in terms of the number of tasks completed; the y-axis measures the memory footprint, i.e., the amount of cached data normalized by the memory capacity. We see in Fig. 3 that the data is produced and consumed in *waves*, where the boundaries between two waves are well aligned with the submission of new jobs in the dependency DAG. This clearly indicates that when a child RDD has been computed, its parents are likely to go inactive.

Inefficiency of Existing Cache Policies. We learn from the empirical studies that the key to efficient cache replacement is to timely evict inactive data blocks. Unfortunately, neither LRU nor LFU is capable of doing so. We refer back to the example of Fig. 1, where each block is of a unit size, and the memory cache can persist two blocks in total. We start with LRU. Without loss of generality, assume that the two blocks A_1 and B_1 are already in memory at the beginning, with the recency rank as $B_1 > A_1$, i.e., from the most- recently-used (MRU) position to the least-recently-used (LRU). Fig. 4 illustrates what happens with the LRU policy when data block A_2 is materialized and then cached. Since block A_2 is derived from A_1, the latter is first referenced as an input at time t_1 and is elevated to the MRU position. Soon later, block A_2 has been materialized at time t_2 and is cached at the MRU position, pushing the least-recently-used block B_1 out of the memory. However, this would incur expensive tear-and-wear cost, in that block B_1 will soon be reloaded in memory to compute block B_2. In fact, we see that the optimal decision is to evict block A_1, as it becomes inactive and will never be used again. This simple example shows that LRU is unable to evict inactive data in time, but it has to wait *passively* until the data demotes to the LRU position, which may take a long time.

We note that the LFU policy suffers from a similar problem. In the previous example, to cache block A_2, the LFU policy would also evict B_1 while retaining block A_1 in memory, because block A_1 has a historical access record (in the computation of A_2), but B_1 does not.

This example demonstrates that recency- and frequency-based cache replacement policies cannot timely evict inactive data even in a simple scenario. In production clusters with massive volumes of inactive data generated by multiple jobs, the cache efficiency of LRU/LFU can only be worse.

2.4 All-or-Nothing Cache Requirement

Prevalent cache algorithms, be it recency- or frequency-based, settle on the cache hit ratio as their primary optimization objective. However, the cache hit ratio fails to capture the *all-or-nothing* requirement of data-parallel tasks and may not be 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 [25]. A task cannot be sped up unless *all* its dependent blocks, which we call *peers* in this paper (e.g., block A_2 and block C_2 in Fig. 1), are cached in memory.

Methodology. To demonstrate the all-or-nothing property in data analytics systems, we ran the Spark job with only one `zip` stage in an Amazon EC2 cluster consisting of 10 `m4.large` instances [24]. The job DAG is illustrated in Fig. 5. Each of the two RDDs A and B is configured as 200 MB. We repeatedly ran the `zip` job in rounds. In the

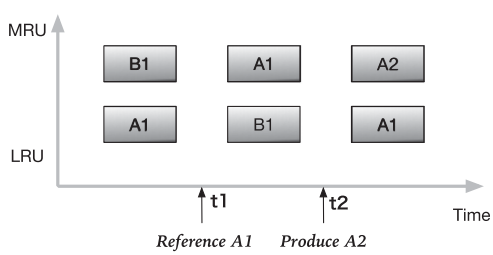


Fig. 4. An example showing that applying LRU in the example of Fig. 1 is unable to timely detect and evict inactive data. The cache capacity is two units.

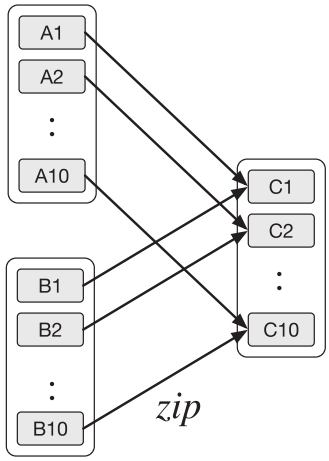


Fig. 5. DAG of a Spark job with only one `zip` stage and three RDDs A, B and C. Each block in RDD C depends on two corresponding blocks from RDD A and RDD B.

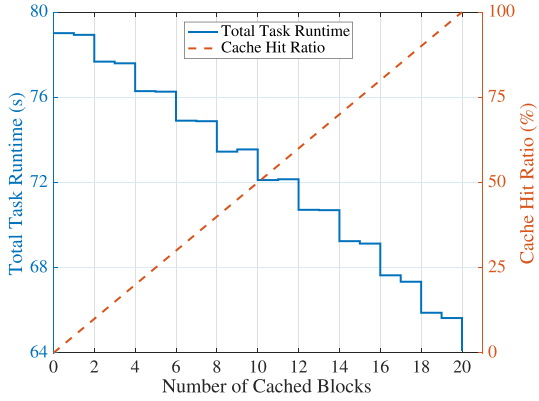


Fig. 6. Total task runtime of the example job in Fig. 5, with the cached RDD blocks increasing one at a time in the order of A_1, B_1, A_2, B_2, ..., A_{10}, B_{10}.

first round, no data block is cached in memory. In each of the subsequent rounds, we sequentially add one more block to the cache, following the caching orders $A_1, B_1, A_2, B_2, \ldots, A_{10}, B_{10}$. Eventually, all 20 blocks were cached in memory in the final round. In each round, we measured the cache hit ratio and the total runtime of all 10 tasks.

The All-or-Nothing Requirement at Two Levels. Fig. 6 depicts the *total* task runtime against the number of RDD blocks cached in memory. We observe the all-or-nothing requirement at two levels: task and stage.

1) *A compute task is only sped up when all of its input peer blocks are cached in memory.* As shown in Fig. 6, despite the linearly growing cache hit ratio with more in-memory blocks, the task completion time is notably reduced only after both of the two peering blocks A_i and B_i have been cached.
2) *For a compute stage where all the tasks are executed in parallel, the stage gets sped up only when the dependent peers of all the tasks are cached in memory.* In our experiment, if the 10 `zip` tasks are scheduled for parallel execution, the completion time of the `zip` stage is bottlenecked by the slowest task. Therefore, speeding up only a subset of the 10 tasks provides no benefit. To reduce the runtime of the parallel-executed `zip` stage, all of the 20 input blocks should be cached.

Inefficiency of Existing Cache Policies. To meet the all-or-nothing caching requirement, the algorithm should identify which blocks are peers of each other, so that they can be cached altogether as a whole. This information can only be learned from the data dependency DAG. However, existing cache policies, such as LRU and LFU, are *oblivious* to the DAG information, and thus are unable to retain the all-or-nothing property.

To summarize, simply relying on the historical information, be it access recency or frequency, is incapable of detecting inactive data in time and satisfying the all-or-nothing requirement. Efficient cache management therefore should factor in the semantics of data dependency DAG. We show how this can be achieved in the next section.

3 Dependency-Aware Cache Management

In this section, we present a new cache management policy, Least Reference Count (LRC). LRC is aware of the application's data dependency DAG. We incrementally develop the LRC algorithm. We start with a simple baseline which makes use of the DAG information without considering the all- or-nothing caching requirement. We shall then generalize the baseline to meet this requirement at different levels.

3.1 Least Reference Count (LRC): The Baseline Form

We begin with the definition of the *reference count*, based on which we propose the LRC algorithm in its baseline form.

Definition 1 (Reference count). *For each data block b, the* reference count *is defined as the number of child blocks derived from b, but not yet computed.*

As a concrete example, we refer back to Fig. 1. Upon the submission of the job DAG, blocks A_1, B_1, C_1 and D_1 all have the same reference count 1, as each of them has only one unmaterialized RDD block depending on it.

Definition 2 (Baseline LRC). *The Least Reference Count (LRC) policy keeps track of the* reference count *of each data block, and whenever needed, it evicts the data block whose* reference count *is the smallest.*

Desirable Properties. The baseline LRC algorithm has two desirable properties that make it highly efficient.

First, with LRC, those inactive data blocks with zero reference count can be quickly identified and evicted by the algorithm. Continuing the example of Fig. 1, assume blocks A_1 and B_1 are in memory at the beginning, and the cache is full. Once block A_2 has been computed, block A_1 becomes inactive with zero reference count and is thus safe to evict to make room for block A_2.

Second, compared to recency and frequency, a data block's reference count is a more accurate indicator of its likelihood of future access. Intuitively, the higher the reference count a block has, the more compute tasks depend on it as an input, and the more likely the block is needed in the downstream computation.

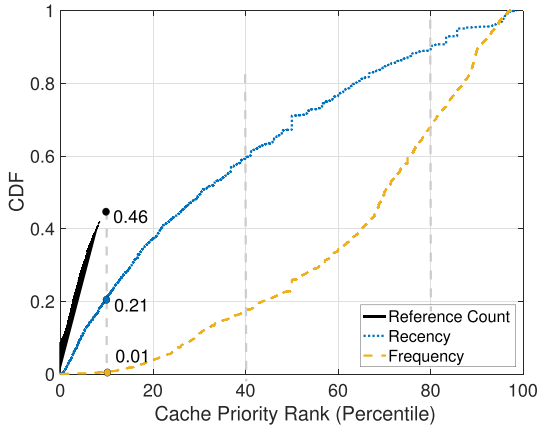


Fig. 7. Distribution of the cache priority ranks of accessed data blocks with respect to recency, frequency, and reference count. The workloads cover all SparkBench applications [16].

To validate this intuition, we ran SparkBench applications on an Amazon EC2 cluster with 10 `m4.large` instances. Specifically, whenever a data block is accessed, we respectively measured the *cache priority rank* (in percentile) of the block in terms of three metrics: the recency of last access, the historical access frequency, and the reference count. A block that is top ranked in terms of a certain metric (say, top 1 percent in terms of recency) is likely persisted in memory if the corresponding cache policy is used (say, LRU). Fig. 7 shows the CDF of the measured ranks across all 15 SparkBench applications. We see that the reference count consistently gives higher ranks than the other two metrics, meaning it is the most accurate indicator of which data will be accessed next. For example, suppose that each data block is of a uniform size, and the cluster's memory can cache only 10 percent of the entire data blocks. Caching the data blocks with the highest reference count leads to the highest cache hit ratio (46 percent), which is $2.19\times$ $(46\times)$ of that based on the recency (frequency).

Discussion. In real-world data analytics applications, the intermediate blocks may differ in sizes. At the first glimpse, it seems that the block size should be taken into consideration while determining the caching preferences. Noting that the performance penalty (gain) of a cache miss (hit), i.e., I/O delay (speedup), is proportional to the block size, the reference count directly measures the potential benefit of caching each byte of the block. Therefore, to improve the overall cache efficiency, there is no need to factor in the block size for the ranking of caching priorities.

3.2 LRC for the All-or-Nothing Requirement

While the baseline LRC algorithm makes use of the data dependency DAG, it does not explicitly address the all-or-nothing caching requirement, which plays a key role in speeding up data analytics applications. Fortunately, with some minor modifications to the calculation of reference count, the baseline algorithm can be easily generalized to meet this requirement at different levels.

A Toy Example. In a cluster environment, depending on the number of available compute slots, parallel tasks in a compute stage are usually scheduled in "waves". Intuitively, the stage completion time can only be reduced when *all* tasks scheduled in one wave are sped up simultaneously. We refer back to the example in Fig. 1. Let T_E and T_F respectively denote the computation time of materializing blocks E and F. To see whether the computation time of the `zip` stage can be effectively reduced by caching, we have to differentiate between the following two cases.

Case-1: There is only one available compute slot in the cluster for the `zip` stage. In this case, blocks E and F will be computed *sequentially* in two "waves", and the computation time of the `zip` stage is $T_{zip} = T_E + T_F$. Therefore, speeding up any of the two `zip` tasks—by caching its two input peers (i.e., zipping blocks A_2 and C_2 to derive E, and B_2 and D_2 to derive F)—helps reduce the total computation time of the `zip` stage. In other words, meeting the all-or-nothing caching requirement at the task level is sufficient to speed up the entire computation, as the two tasks cannot be executed in parallel.

Case-2: There are two available slots in the cluster for the `zip` stage, with which blocks E and F are computed in parallel. In this case, the computation time of the `zip` stage is bottlenecked by the slower one of the two tasks, i.e., $T_{zip} = \max\{T_E, T_F\}$, and speeding up only one task brings no benefit. This requires the all-or-nothing caching requirement to be satisfied at the stage level: to accelerate the computation of the `zip` stage, all the four dependent blocks A_2, B_2, C_2, and D_2 must be cached in memory.

Intuitions. From the toy example, we learned that the level at which the all-or- nothing caching requirement should be achieved depends on how tasks are scheduled. Unfortunately, this is the runtime information that cannot be obtained *a priori*. Cluster scheduling is highly dynamic in production cloud platforms, where multiple users and applications contend for compute resources. In such a shared environment, it is very hard, if not impossible, to predict how many slots will be allocated to an application, and if those slots are sufficient to accommodate all the parallel tasks of that application.

Given such an uncertainty, we propose two replacement strategies, one conservative and the other aggressive, that respectively achieve the all-or-nothing requirement at the task and stage levels. The conservative strategy is *pessimistic* about the available compute resources (e.g., Case 1 in the previous example) and assumes that tasks are likely to be scheduled in sequence. Based on this assumption, the algorithm *conservatively* evicts data blocks, only when caching them cannot speed up a single task. Through this conservative eviction strategy, the algorithm expects to speed up as many tasks as possible.

The aggressive strategy, on the other hand, makes an optimistic assumption that the available slots are sufficient to accommodate all the parallel tasks of a compute stage (e.g., Case-2 in the previous example). The algorithm hence strives to achieve the all-or-nothing requirement at the stage level. It *aggressively* evicts all the input blocks of a compute stage whenever it is not possible to speed up all tasks of that stage.

With minor modifications to the definition of reference count, the baseline LRC algorithm can be easily extended to implement both replacement strategies.

LRC with Conservative Replacement. We start with the conservative replacement that achieves the all-or-nothing requirement at the task level. Intuitively, if an input block of

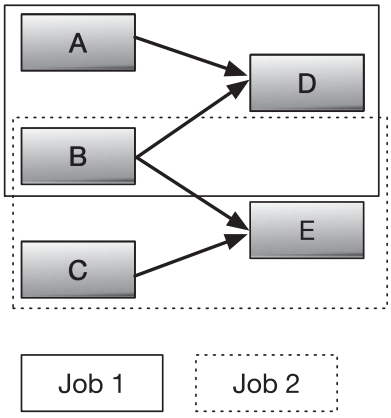


Fig. 8. An example DAG to illustrate the workflow of LRC-Online, where blocks D and E are respectively computed by Job 1 and Job 2. The former is submitted earlier than the latter.

a task has been evicted, the task cannot be sped up. We therefore have the following definition:

Definition 3. *We say a compute* task *can be sped up if none of its input blocks have been evicted, i.e., they are either cached in memory or have not been materialized.*

We redefine the reference count for each data block as *the number of unscheduled tasks that depend on it and can be sped up*. The LRC algorithm always evicts the data block with the least reference count. As an illustrative example, we refer back to Fig. 1 and assume that block A_2 has been evicted from the memory. As a result, the `zip` task used to derive block E cannot be sped up, and the reference count of the other input block C_2 becomes 0. This suggests that block C_2 should be evicted out of the memory as well, as caching it cannot speed up any task.

LRC with Aggressive Replacement. We next present another LRC alternative which aggressively evicts data blocks if the all-or-nothing requirement cannot be met at the stage level. Suppose that the cluster has a sufficient number of slots to accommodate all parallel tasks of a compute stage, and the stage completion time is bottlenecked by the slowest task. We have the following definition:

Definition 4. *We say a compute* stage *can be sped up if all of its unscheduled tasks can be sped up.*

To achieve the all-or-nothing requirement at the stage level, we redefine the reference count for each data block as *the number of uncompleted stages which depend on that block and can be sped up*. Intuitively, the higher the reference count of a data block, the more stages that can be sped up by caching that block. The LRC algorithm always evicts the data block with the least reference count.

We refer back to Fig. 1 and assume parallel execution of the two `zip` tasks. If block A_2 is not cached in memory, the reference counts of blocks B_2, C_2, and D_2 are all 0 as caching them cannot speed up the `zip` stage.

Discussion. Depending on the compute resources provisioned in the cluster, the LRC algorithm should be configured with conservative or aggressive replacement accordingly. Intuitively, when the compute resources are heavily contended, the conservative strategy is likely to outperform the aggressive alternative. In this case, tasks are likely executed in sequence, and the stage completion time depends on the number of tasks that are sped up. The conservative strategy seeks to cache the input peers for as many tasks as possible and hence can achieve better performance than the aggressive strategy: the latter is too ambitious to speed up all tasks in a stage, missing the opportunity to accelerate some of them.

On the other hand, when the cluster has sufficient amount of resources to accommodate parallel computation of all tasks in each compute stage, meeting the all-or-nothing caching requirement becomes highly relevant. The aggressive replacement strategy therefore arises as a better choice that helps reduce the stage completion time effectively. We shall confirm these points through evaluations in Section 5.1.

3.3 LRC-Online

Accurately computing the reference count requires extracting the *entire* data dependency DAG in an application. In frameworks such as Spark and Tez, an application typically runs as a workflow of multiple jobs, where a downstream job is submitted after all its upstream dependents have completed. For each job, the DAG information is available only after the job is submitted. Therefore, for an application, the data dependency between its constituent jobs is usually runtime information that cannot be known *a priori*, especially when the jobs are executed iteratively. We address this challenge in two cases.

Recurring Applications. Production trace studies reveal that a large portion of cluster workloads are recurring applications [26], which are run periodically when new data becomes available. For these applications, we can learn their DAGs from previous runs and apply offline LRC algorithms directly.

Non-Recurring Applications. For *non-recurring* applications such as interactive ad-hoc queries, offline LRC algorithms cannot be used for optimal eviction decision because the application's DAG dependency is gradually revealed as the workflow executes and new jobs are submitted. We therefore propose an online solution called *LRC-Online* that approximates the eviction decisions generated in the offline settings. Upon the submission of a new job, LRC-Online parses the reference counts of the blocks in that job and updates the reference count record it maintains at runtime. For blocks that already exist, LRC-Online adds its reference count in the newly submitted job to its remaining reference count. Fig. 8 gives an example illustrating the workflow of LRC-Online. We consider a simple application consisting of two jobs, Job 1 and Job 2, that respectively compute blocks D and E. The complete DAG of application is not known *a priori* but gradually revealed upon a job submission. Let Job 1 be submitted first, and the reference counts of blocks A and B are set to 1. While this is inaccurate as block B is also used to compute block E, LRC-Online will soon correct its value after Job 2 has been submitted.

Similar to the offline solution, LRC-Online can enforce conservative or aggressive replacement policy when updating *effective* reference counts at runtime in order to retain the all-or-nothing property. Without the offline information of the entire application, LRC-Online can only meet the all-or-nothing caching requirements in the currently submitted jobs. Furthermore, the all-or-nothing requirement at

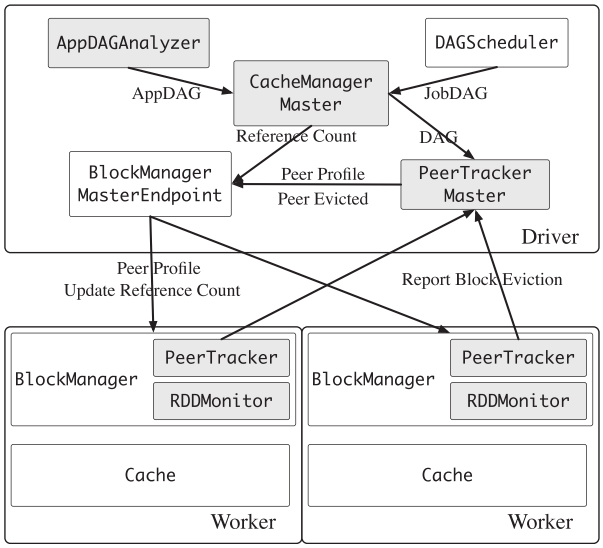


Fig. 9. Overall system architecture of the proposed application-aware cache manager in Spark. Our implementation modules are highlighted as shaded boxes.

runtime can be challenging to capture precisely. On one hand, the requirement may be different levels for different jobs running concurrently. On the other hand, the requirement also changes with time. If we can predict which tasks will be scheduled in parallel, e.g., based on prediction or learning techniques, we can enforce the two replacement strategies to different jobs accordingly. We can also dynamically switch between the two at runtime. We leave this to future exploration but concentrate on defining and characterizing the two atomic replacement strategies in this paper.

LRC-Online extends the applicability of LRC algorithms to the shared environments where multiple applications run in a cluster and have common input datasets. In this case, both inter- and intra-application dependencies exist, but cannot be predicted as they depend on the actual job scheduling order. LRC-Online can update the reference counts once new dependency information becomes available.

We have evaluated the performance of LRC-Online (see Section 5). Our results show that the performance of LRC-Online remains close to that of LRC with the offline DAG information.

TABLE 2
Key APIs of our Spark Implementation

API	Description
`parseDAG`	The `CacheManagerMaster` parses the DAG information obtained from the `DAGScheduler` (or the `AppDAGAnalyzer`) and returns the profiled reference count for each RDD and the peer information.
`broadcastReferenceCount`	The `CacheManagerMaster` sends the parsed reference count profile to the corresponding `RDDMonitor`.
`broadcastPeerProfile`	The `PeerTrackerMaster` broadcasts the peer profile to the `PeerTrackers` via the `BlockManagerMasterEndpoint`.
`updateReferenceCount`	Upon receiving the reference count updating message, the `RDDMonitor` updates reference counts of the corresponding RDD blocks.
`decrementReferenceCount`	When an RDD block is referenced, the `RDDMonitor` deducts the its reference count in the maintained profile.
`evictBlocks`	When the cache is full, the `BlockManager` evicts the blocks in the ascending order of reference count until enough space is released.
`reportBlockEviction`	The `PeerTracker` informs the `PeerTrackerMaster` when an RDD block from a complete-group is evicted out of memory.
`broadcastPeerEviction`	Upon receiving a block-eviction report, the `PeerTrackerMaster` broadcasts the block-eviction message to all the `PeerTrackers`.
`checkOnEvictionMessage`	Upon receiving a block-eviction message, the `PeerTracker` checks whether the evicted block belongs to any complete-group. If yes, notify the `RDDMonitor` to decrease the reference counts of blocks in this group correspondingly.
`checkOnBlockEviction`	When a block is evicted out of memory, the `PeerTracker` first checks whether this block belongs to any complete-group. If yes, it sends a block-eviction message to the `PeerTrackerMaster`.

4 Implementation

In this section, we describe our implementation of LRC as a pluggable cache manager in Spark.

4.1 Spark Implementation

Architecture Overview: Fig. 9 gives an architecture overview of our cache manager, where the shaded boxes highlight our implementation modules.

The cache manager consists of two centralized controllers, i.e., `CacheManagerMaster` and `PeerTrackerMaster`, on the driver node and two distributed components, i.e., `RDDMonitor` and `PeerTracker`, on each worker node. The `CacheManagerMaster` implements the main logic of the LRC and LRC-Online policies. It obtains the data-dependency DAGs from `AppDAGAnalyzer` for recurring applications or from `DAGScheduler` in an online fashion for non-recurring applications. The `PeerTrackerMaster` incorporates the all-or-nothing property into the cache manager. It profiles the peer information in the DAGs obtained from the `CacheManagerMaster`, and communicates with the `PeerTrackers` in worker nodes when necessary. We summarize the key APIs of our implementation in Table 2.

Workflow. Whenever an application is submitted to the Spark driver, the `AppDAGAnalyzer` first detects whether the application has been executed before. For a recurring application, the `AppDAGAnalyzer` exposes the entire application DAG learned from previous runs to the `CacheManagerMaster` to implement the *LRC* policy. For a non-recurring application, the `CacheManagerMaster` obtains the job DAGs from `DAGScheduler` instead to implement the *LRC-Online* policy. Once the `CacheManagerMaster` receives a DAG, it parses the reference count profile and peer information, which are later broadcast via the

BlockManagerMasterEndpoint to RDDMonitors and PeerTrackers in worker nodes, respectively. The RDDMonitors maintain the reference count information locally and helps to make the eviction decisions whenever the cache is full. The PeerTrackers report the caching status of the peer blocks to meet the all-or-nothing requirement.

Intuitively, to implement LRC with *conservative* or *aggressive* replacement, the PeerTrackerMaster has to keep track of the caching status of every peer block. However, such a naive approach requires massive additional message exchanges between PeerTrackerMaster and PeerTrackers. We manage to reduce the communication overhead of the system by dividing peer blocks into *complete-group*s. A complete-group is defined as a group of *cached* (or to-be-computed) blocks that are able to speed up a task (stage) for LRC with conservative (aggressive) replacement. By definition, a complete-group becomes invalid once any of its blocks is evicted out of memory. In that case, the reference count of blocks in the complete-group will be updated, once and for all. Therefore, only one message exchange is needed for any complete-group. Specifically, upon a block eviction, the PeerTracker checks whether this block belongs to any complete-group. If so, it removes these complete-groups from its record and notifies the RDDMonitor to update the reference counts of blocks in these groups. A block eviction report is sent to the PeerTrackerMaster and then broadcasted to other workers, in which the PeerTrackers and RDDMonitors will update the reference counts for its peer blocks accordingly.

4.2 Discussions

Communication Overhead: We manage to reduce the communication overhead with two approaches.

First, each worker maintains the reference count profile locally and synchronizes with the controller with the least message exchanges possible. The CacheManagerMaster sends reference count updates to the corresponding workers only when necessary. In particular, there are two cases when an update is required: 1) When a new job DAG is received from the DAGScheduler, the CacheManagerMaster notifies workers to update the reference count of the corresponding RDD blocks; 2) when an RDD block has been referenced, and the block has replicas on the other workers, all those workers should reduce a reference as well to have a consistent reference count of the block. By default, RDD blocks are not replicated across the cluster, so our implementation checks the configuration first to see if the second case needs to be considered.

Second, by introducing complete-groups, our implementation for all-or-nothing support only exchanges messages if necessary. To prove this property, we first show that at most one broadcasting is required for an entire complete-group in our implementation. By managing blocks in complete-groups locally in the PeerTrackers, we obviate the need of tracking the caching status of peer blocks individually. Once a block in a complete-group is evicted, the reference counts of the other blocks in the same group need simultaneous updates, which can be achieved using one broadcast message. Once the eviction message gets broadcast, the complete-group can be removed, and no more update messages are required for this group. We next show that, if a block in a complete-group is evicted, the broadcast message is required to be sent to all workers. Due to RDD's lazy execution nature, it is possible that some of the blocks in a complete-group might not be computed yet when an eviction happens to a member block. As a result, the block eviction message should be broadcast to all workers.

Fault Tolerance. It is possible that a worker may lose connection to the driver at runtime, which results in a task failure. In this case, the reference count profile maintained by the CacheManagerMaster will be inaccurate as the failed tasks will be rescheduled soon. To address this inconsistency issue, the CacheManagerMaster records the job ID upon receiving a job DAG from DAGScheduler. In this way, the CacheManagerMaster can quickly detect job re-computation if the same job ID has been spotted before. The consistency check for the reference count can then be applied.

5 Evaluations

In this section, we evaluate the performance of LRC policies through Amazon EC2 deployment and trace-driven simulations. First, we investigate the impact of the all-or-nothing caching requirement with synthetic workloads (Section 5.1). We compare the efficacy of LRC with aggressive and conservative replacement with different resource provisions, in an effort to simulate the varying production cluster environment. Next, we evaluate the overall performance of LRC against typical application workloads in SparkBench suite [16] (Section 5.2). We investigate how being dependency-aware helps speed up a *single* application (Section 5.2.1) with much shorter runtime, and how such a benefit can be achieved even when the DAG information is profiled online. We next evaluate the performance of LRC in a *multi-tenant* environment (Section 5.2.2) where multiple applications run in a shared cluster, competing for the memory caches against each other. Finally, we conduct simulations driven by production traces [18] from the Alibaba Group (Section 5.3). We show the performance advantages of LRC over existing baselines, and demonstrate the necessity of complying with the all-or-nothing caching requirement in real clusters. We summarize the highlights of our evaluations as follows.

- In the presence of the all-or-nothing requirement, the cache hit ratio is not closely relevant to the cache performance. In other words, increasing the cache hit ratio does not necessarily result in a shorter application completion time.
- In a cluster with insufficient compute resources, the LRC policy with conservative replacement has the shortest runtime, which is 34.0 and 19.5 percent faster compared with the LRU and LRC policies, respectively. When the computing power is sufficient to allow for parallel executions of an entire compute stage, the LRC policy with aggressive replacement achieves the fastest execution speed, decreasing the runtime by 30.2 and 15.9 percent over the LRU and the baseline LRC policy, respectively.

TABLE 3
Workload and Cluster Settings for Experiments on Conservative/Aggressive Replacement

Compute resources	Nodes	Tenants	Tasks per tenant	Total input size
Insufficient	10	10	100	4 GB
Sufficient	50	20	50	8 GB

- For typical workloads in production clusters, LRC can reduce the application runtime by up to 60 percent compared with the default LRU policy.
- In most cases, LRC-Online well approximates LRC and consistently outperforms the LRU policy across applications.
- Trace-driven simulations show that LRC achieves 22.3 and 284.4 percent higher average cache hit ratio compared with LRU and MEMTUNE [19], a cache replacement policy that also leverages data dependency. LRC with aggressive replacement has 21 percent higher 5th percentile stage acceleration ratio over the baseline LRC, indicating a better chance to speed up the compute stages.

Experimental Settings.Our implementation is based on Spark 1.6.1. In order to highlight the performance difference of memory read-write and disk I/O, we disabled the OS page cache using memory buffer by triggering direct disk I/O from/to the hard disk. If not specified otherwise, each node we used in the EC2 deployment is an `m4.large` instance [24], with a dual-core 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processor and 8 GB memory.

5.1 Conservative and Aggressive Replacement

We start with evaluations on the impact of the all-or-nothing caching requirement. We use synthetic workloads to demonstrate how well the all-or-nothing requirement is addressed by our proposed policies with conservative/aggressive replacement under different cluster settings.

Workloads. In this experiment, we simulate multiple tenants submitting Spark `zip` [25] jobs in parallel. In each job, two files each of size 400 MB are first partitioned into N blocks stored across the cluster. After that, these blocks are zipped into key-value pairs, where the keys are the blocks from the first file, and the values are the blocks from the second file. Notice that only when both the key and value are cached in memory will a `zip` task be sped up.

Cluster Deployment. We deploy two clusters consisting of 10 and 50 nodes, respectively, to simulate the cases with insufficient and abundant compute resources in accordance with the two levels of all-or-nothing requirement. In the first scenario, we have 10 tenants competing for the compute slots in the 10-node cluster. Each file is decomposed into 100 blocks, i.e., $N = 100$. Each tenant has 100 `zip` tasks to run, which cannot be executed in parallel. In the latter case with a 50-node cluster, we increase the number of tenants to 20 and have $N = 50$. Therefore, the cluster is capable to schedule all the 50 tasks of a tenant for parallel execution. The workloads and cluster settings are summarized in Table 3.

Metric.Ultimately, we improve the cache management in order to speed up the data analytics computations. The metric we use to evaluate, hence, is the average application runtime. Besides, we propose two metrics, *task acceleration ratio* and *stage acceleration ratio*, to effectively measure how much the computation is sped up by the memory caches. Specifically, we define that a compute task is sped up if all of its input data is cached in memory, and that a compute stage is sped up if all of its tasks are sped up. The task/stage acceleration ratio is defined as the percentage of tasks/stages being sped up by the cache. The baseline for our deployment is the LRU policy.

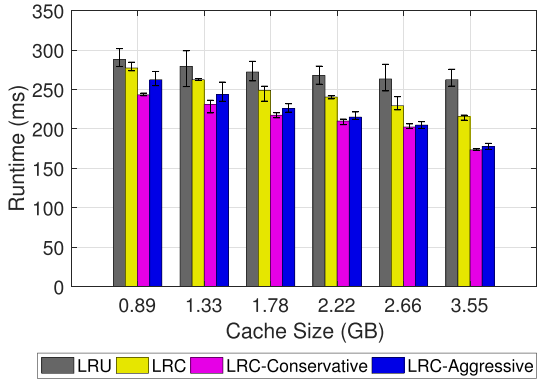


Fig. 10. Runtime with different cache policies on a 10-node cluster.

5.1.1 Cluster with Insufficient Compute Resources.

In the first set of experiments, we compare the cache perform of four policies, i.e., LRU, LRC, LRC with conservative replacement, and LRC with aggressive replacement in a cluster with insufficient compute resources. We measure the total experiment runtime, i.e., the make span of the 10 submitted jobs, the cache hit ratio, and the task acceleration ratio.

Runtime. Fig. 10 shows the average completion time over 10 repeated runs, where the error bars depict the maximum and minimum values. To our expectation, as the size of RDD cache increases, the total experiment runtime decreases under all the four cache policies. In all cases, LRC consistently outperforms the default LRU policy. The two policies taking care of the all-or-nothing requirement further reduces the experiment completion time, where LRC with conservative replacement achieves the shortest runtime. When the cache size is 3.55 GB, for instance, the average runtimes under the four policies are 262 s (LRU), 215 s (LRC), 173 s (LRC with conservative replacement), and 178 s (LRC with aggressive replacement) respectively. The LRC policy with conservative replacement speeds up the job execution by 34.0 and 19.5 percent compared with the LRU and LRC policies, respectively.

Cache Hit Ratio. Fig. 11 shows that LRC achieves the highest cache hit ratio, while LRC with conservative/aggressive replacement closely follows. This is because LRC aims to optimize the cache hit ratio, and it outperforms LRU by taking advantage of the data dependency information. The LRC policies with conservative/aggressive replacement also make use of this information, but they give up on retaining those ineffective cache hits that are unable to speed up the computation. It is for this reason that the cache hit ratio is slightly compromised. In particular, LRC with aggressive replacement sacrifices

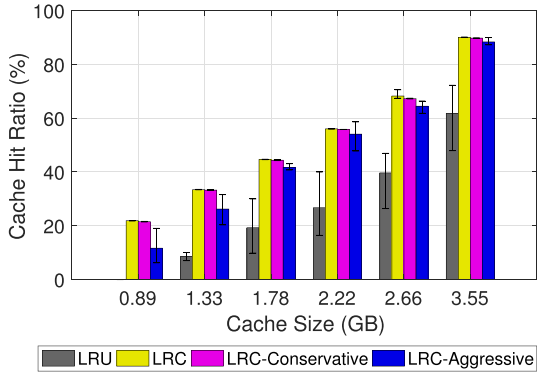


Fig. 11. Cache hit ratio with different cache policies on a 10-node cluster.

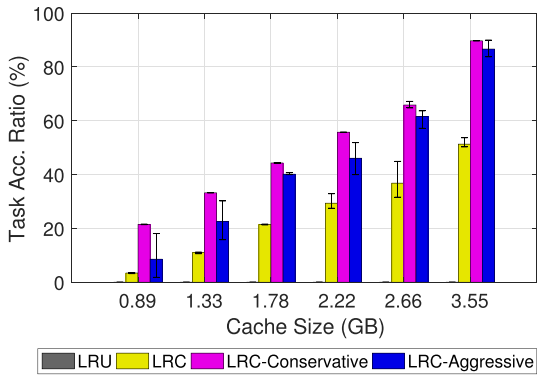


Fig. 12. Task acceleration ratio with different cache policies on a 10-node cluster. Notice that the task acceleration of LRU is always near zero.

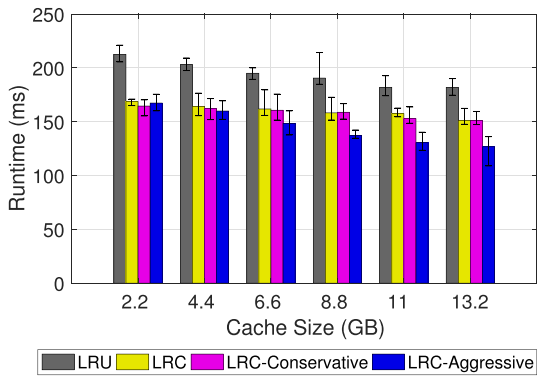


Fig. 13. Runtime with different cache policies on a 50-node cluster.

more than LRC with conservative replacement as it only favors the data blocks that speed up an entire compute stage.

Task Acceleration Ratio. Fig. 12 shows that LRC with conservative replacement always achieves the highest task acceleration ratio. The smaller the cache size is, the more advantageous it becomes compared with LRC, which precisely matches the performance in runtime shown in Fig. 10.

We therefore conclude that when the compute resources are constrained compared to the demands, the task acceleration ratio is a more relevant metric than the cache hit ratio. In such a scenario, the LRC policy with conservative replacement is able to make the best use of the cache resources to effectively speed up the computation.

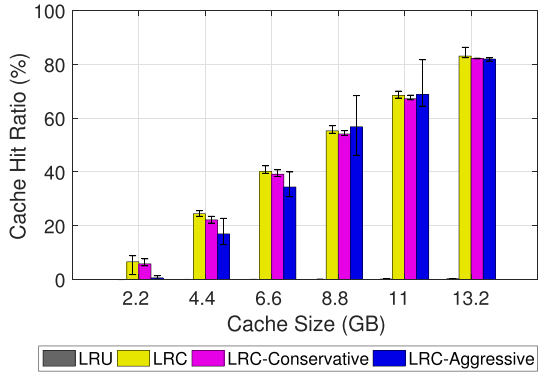


Fig. 14. Cache hit ratio with different cache policies on a 50-node cluster.

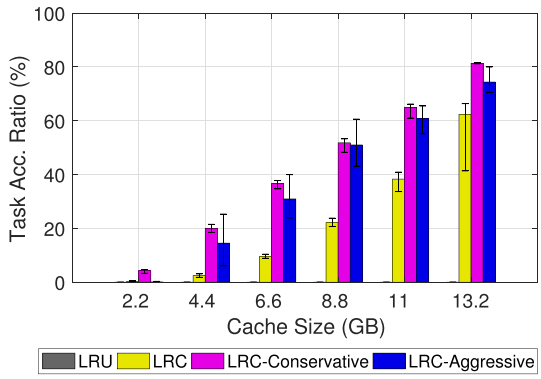


Fig. 15. Task acceleration ratio with different cache policies on a 50-node cluster.

5.1.2 *Cluster with Sufficient Compute Resources.*

Now we investigate the scenario where the computing power is abundant to execute the tasks of an entire compute stage in parallel with a cluster of 50 nodes. We measure the performance in experiment runtime, the cache hit ratio, the task acceleration ratio and the stage acceleration ratio for comparison. The results are shown in Figs. 13, 14, 15, and 16.

Runtime. Fig. 13 shows that the execution time is the shortest under the LRC policy with aggressive replacement. The advantage gets more prominent when the RDD cache size increases. For instance, with an RDD cache of 13.2 GB, the average runtimes under the four policies are 182 s (LRU), 151 s (LRC), 151 s (LRC with conservative replacement), and 127 s (LRC with aggressive replacement), respectively. The LRC policy with aggressive replacement speeds up the job execution by 30.2 and 15.9 percent compared with the LRU and LRC policies, respectively.

Cache Hit Ratio and Task Acceleration Ratio. Figs. 14 and 15 show the cache hit ratio and task hit ratio, respectively. As observed previously, the LRC policy achieves the highest cache hit ratio, and the LRC policy with conservative replacement achieves the highest task hit ratio. Nonetheless, it takes the two policies nearly the same amount of time to finish the 50 jobs, which is longer than the execution time under the LRC policy with aggressive replacement.

Stage Acceleration Ratio. Fig. 16 compares the stage hit ratio of the four policies, where the LRC police with aggressive replacement consistently outperforms the others. Surprisingly, the LRC policy (the LRC policy with conservative

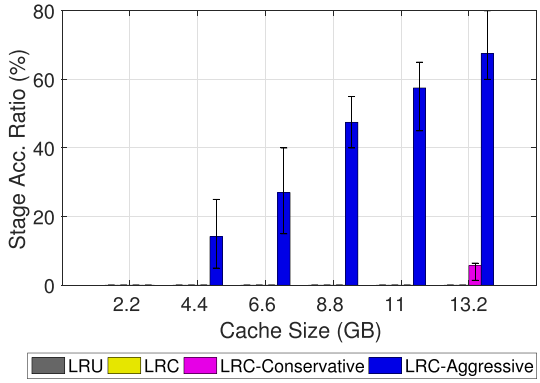


Fig. 16. Stage acceleration ratio with different cache policies on a 50-node cluster.

TABLE 4
Impact of Memory Caches on the Application Runtime

Workload	Cache All	Cache None
Page Rank	56 s	552 s
Connected Component	34 s	72 s
Shortest Paths	36 s	78 s
K-Means	26 s	30 s
Pregel Operation	42 s	156 s
Strongly Connected Component	126 s	216 s
Label Propagation	34 s	37 s
SVD Plus Plus	55 s	120 s
Triangle Count	84 s	99 s
Support Vector Machine (SVM)	72 s	138 s

We compare against the two extreme cases: caching all data in memory versus caching none.

policy) fails to speed up any compute stage in most cases, despite of the high cache hit ratio (task acceleration ratio) achieved. This explains why they need a longer execution time than the LRC policy with aggressive replacement in this case.

Notice that the task (stage) acceleration 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. When the `zip` stage starts, only the values are cached. Therefore, no task can be sped up, let alone speeding any compute stage.

We therefore conclude that when the compute resources are sufficient to allow for parallel execution of entire stages, the stage acceleration ratio is the most relevant metric for cache performance. In such a scenario, the LRC policy with strict replacement works as the best choice to effectively speed up the computation.

5.2 Macro-Benchmark

To evaluate the performance of the LRC policy in production clusters, we do experiments using typical application workloads in SparkBench suite [16]. Notice that the applications in SparkBench involve very little all-or-nothing requirements, where most of the tasks depends on a single RDD block.[2] Therefore, the LRC policies with conservative/aggressive replacement have trivial contribution to the cache performance over the baseline LRC policy. In the following experiments, we focus on the performance of LRC and LRC-Online against the default LRU policy.

5.2.1 Single-Tenant Experiment

We start with a simple scenario where a single tenant runs an application in a small private cluster consisting of 20 nodes. We ran typical application workloads in SparkBench and measured the cache hit ratio as well as the application runtime.

Relevance of Memory Caches. It is worth emphasizing that memory caches may become irrelevant for some applications. For example, we have observed in SparkBench that some applications are compute-intensive, and their runtime is mostly dictated by the CPU cycles. Some applications, on the other hand, need to shuffle large volumes of data, and their performance is bottlenecked by the network. These applications benefit little from efficient cache management and do not see a significant runtime improvement even if the system caches all data in memory.

In order to differentiate from these applications, we respectively measured the application runtime in two extreme cases: 1) the system has large enough memory and caches all data, and 2) the system caches no data at all. We summarize our measurement results in Table 4. We see that some SparkBench applications, notably *K-Means*, *Label Propagation* and *Triangle Count*, have almost the same runtime in the two cases, meaning that memory caches are irrelevant to their performance. We therefore exclude these applications from evaluations but focus on four memory-intensive workloads whose performance critically depends on cache management: *Page Rank*, *Pregel Operation*, *Connected Component*, and *SVD Plus Plus*. The input data size varies from 120 MB to 540 MB.

Cache Hit Ratio and Application Runtime. We ran each application using three cache replacement policies, i.e., LRU, LRC, and LRC-Online, 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 cache hit ratio and the application runtime against different cache sizes and depict the results in Figs. 17 and 18, respectively. The results have been averaged over 5 runs.

As expected, the less availability of the memory caches in the cluster, the smaller the cache hit ratio (Fig. 17), and the longer the application runs (Fig. 18). Regardless of the cache size, the two LRC algorithms consistently outperform LRU, the default cache management policy in Spark, across all applications. The benefits achieved by the LRC policy, in terms of the application speedup, varies with different cache sizes as well as the application workloads. In particular, compared with the default LRU policy, our LRC algorithm reduces the runtime of *Page Rank* by 60 percent (from 170 s to 64 s) when the cluster cache size is 5.5 GB. Table 5 summarizes the largest runtime savings of LRC over LRU for each application we evaluated.

2. Shuffling is not considered for all-or-nothing requirement, as shuffle blocks are not stored in RDD caches.

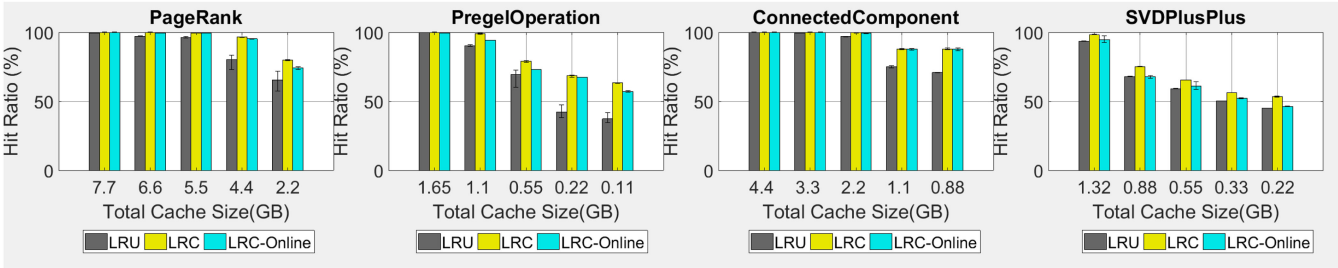


Fig. 17. Cache hit ratio under the three cache management policies with different cache sizes.

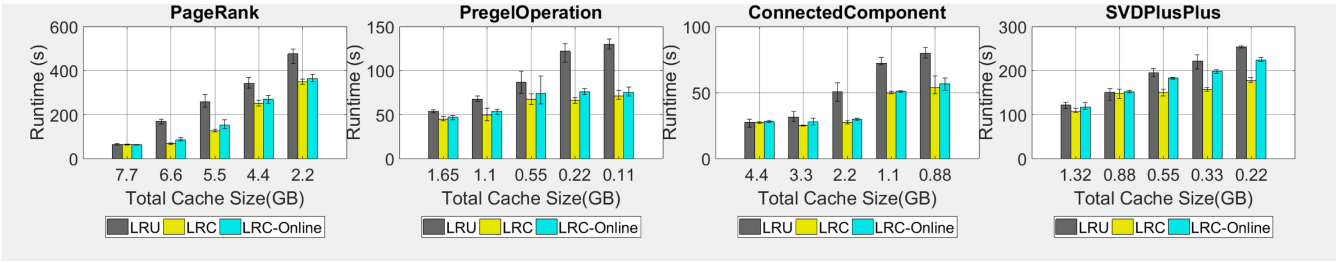


Fig. 18. Application runtime under the three cache management policies with different cache sizes.

TABLE 5
Summary of the Maximum Reduction of Application Runtime Over LRU

Workload	Cache Size	LRU	LRC	LRC-Online	Speedup by LRC	Speedup by LRC-Online
Page Rank	6.6 GB	169.3 s	68.4 s	84.5 s	59.58%	50.06%
Pregel Operation	0.22 GB	121.9 s	66.3 s	75.9 s	45.64%	37.74%
Connected Component	2.2 GB	50.6 s	27.6 s	29.9 s	45.47%	40.97%
SVD Plus Plus	0.88 GB	254.3 s	177.6 s	223.9 s	30.17%	11.96%

The efficiency of LRC policy can also be illustrated from a different perspective, in that LRC requires much smaller cache spaces than that of LRU to achieve the same cache hit ratio. For example, to achieve the target hit ratio of 0.7 for *Pregel Operation*, LRU requires 0.55 GB memory caches. In comparison, LRC requires only 0.22 GB, an equivalent of 60 percent saving of cache spaces.

We make another interesting observation that for different applications, the cache hit ratio has different impacts on their runtime. For example, the workload of *Page Rank* suffers from the most significant slowdown, from 67 s to 320 s, when the cache hit ratio decreases from 1 to 0.85. The reason is that the computation of *Page Rank* consists of some large RDDs, and their cache miss critically increases the total runtime. For *Connected Component*, salient slowdown is observed when the cache hit ratio drops from 0.7 to 0.4; for *SVD Plus Plus*, we observe a linear slowdown with respect to the decrease of cache hit ratio.

LRC-Online. As discussed in the previous section, when the *entire* application DAG cannot be retrieved *a priori*, we can profile the submitted job DAGs at runtime using LRC-Online. We now evaluate how such an online approach can approximate the LRC policy with offline knowledge of application DAG. We see through Figs. 17 and 18, that LRC-Online is a close approximation of LRC for all applications except *SVD Plus Plus*. As illustrated in Table 5, with online profiling, LRC can only speed up the application by 12 percent, as opposed to 30 percent provided by the offline algorithm.

We attribute the performance loss of LRC-Online to the fact that the datasets generated in the current job might be referenced by another in the future, whose DAG is yet available to the *DAGScheduler*. Therefore, the reference count of the dataset calculated at the current stage may not be accurate. To quantify the inaccuracy of online profiling, we measure *reference distance*, for each data reference, as the number of intermediate jobs from the *source job* where the data is generated to the *destination job* where the data is used. Intuitively, the longer the reference distance is, the greater chance it is that referencing the block in the future may encounter a cache miss. This is because without knowing the entire application DAG beforehand, LRC-Online can only tell the data dependency in the current job, and will likely evict all the generated data blocks after the source job has completed.

Table 6 summarizes the average reference distance of the data blocks generated in each application. All applications but *SVD Plus Plus* have reference distance less than 1, meaning that most of the generated data is likely used either by the current job or the next one. This result is in line with the observation made in Fig. 3, where intermediate data goes inactive in waves and in lockstep with job submission. *SVD Plus Plus*, on the other hand, has the longest reference distance, which explains its performance loss with LRC-Online.

To summarize, LRC-Online is a practical solution that well approximates LRC and consistently outperforms the LRU policy across applications.

TABLE 6
Average Reference Distance

Workload	Average Reference Distance
PageRank	0.95
PregelOperation	0.73
ConnectedComponent	0.74
SVDPlusPlus	1.71

5.2.2 Multi-Tenant Experiment

We now investigate how our cache manager performs in a multi-tenant environment through a 50-node EC2 deployment. Notice that the Spark driver unifies the indexing of jobs and RDDs for all tenants. Once a job is submitted, the RDDs in its DAG are indexed incrementally based on the unified index. In this case, the offline RDD reference count is unavailable because the order of actual job submission sequence from multiple tenants is uncertain due to runtime dynamics. Therefore, the data dependency DAG can only be determined online. For this reason, we compare the performance of LRC-Online against LRU.

In the first set of experiments, we emulated 16 tenants submitting jobs simultaneously to the Spark driver, with different cache sizes. The workload profile of the tenants is given in Table 7. In the second experiment, we fixed the total cache size to be 1.26 GB and increased the number of tenants from 8 to 20.

Figs. 19 and 20 show the results. We find that LRC-Online is capable of achieving larger performance gains over LRU with smaller cache sizes or with more tenants. This suggests that leveraging the application semantics of data dependency is of great significance, especially when the cluster memory caches are heavily competed.

5.3 Trace-Driven Simulations

Finally, we evaluate the performance of LRC policies against industrial workload traces to evaluate the performance of LRC policies at a larger scale. Specifically, we obtain the production workloads released in an Alibaba trace [18]. As the trace involves thousands of nodes, we resort to trace-driven simulations to faithfully replay the Alibaba workloads.

Workload. We use the recently released workload trace [18] collected from an Alibaba's production cluster. This trace reveals the DAG structures of business-critical workloads of Alibaba, covering over 4 million MapReduce-like batch jobs running in a cluster of around 4k machines over 8 days. As the Alibaba trace provides no information about the job's input/output data size, we refer to the coflow benchmark [27] synthesized from a production trace

TABLE 7
Summary of Workloads used in the Multi-Tenant Experiment

Tenant Index	Workload	Input Data Size
1-8	ConnectedComponent	745.4 MB
9-16	PregelOperation	66.9 MB

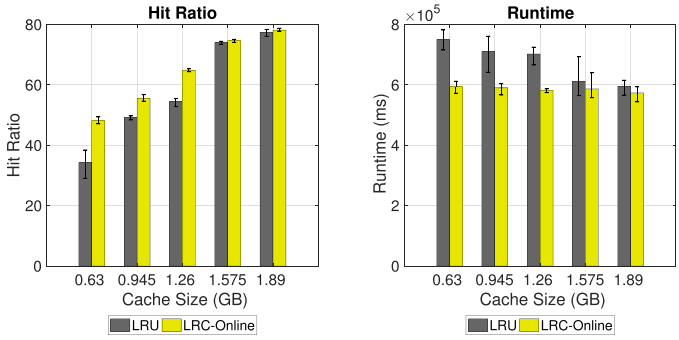


Fig. 19. Cache hit ratio and total runtime with different cache sizes in the multi-tenant experiment.

of a 3000-machine MapReduce cluster at Facebook. This trace provides the shuffled data size in representative MapReduce jobs, which can be used to estimate the amount of intermediate data generated by the MapReduce jobs in the Alibaba trace.

Setup. We selected 100 jobs in the Alibaba trace with the most complex DAG structure (the most number of computing stages), as those jobs generate large amounts of intermediate data, posing a significant pressure to the cache management. We replayed the execution of 100 jobs sequentially, and set the total cache size to 1 GB for each job. For each compute task, we synthesized its output data size by randomly sampling the per-mapper flow size across all jobs in the coflow benchmark [27]. We have open-sourced our simulator [20] for reproducing the results and experimenting with other cache replacement policies against the Alibaba trace.

Baselines. We compare the performance of LRC policies (the baseline LRC and LRC with conservative/aggressive replacement) against LRU and MEMTUNE [19], a memory manager that also leverages the data dependency knowledge to optimize the cache replacement in data analytics systems. MEMTUNE employs three techniques to improve memory management, i.e., dynamically tuning the memory share for computation and data caching, prefetching data for downstream tasks, and employing a replacement strategy that seeks to evict the input data of finished tasks (see Section 6 for the comparison with LRC). Since these three optimization techniques are orthogonal to each other and LRC policies focus on cache replacement, we limit the scope of comparison against MEMTUNE to its cache replacement strategy. The

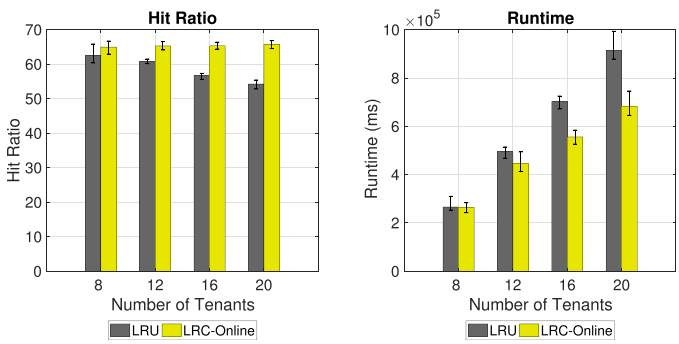


Fig. 20. Cache hit ratio and total runtime with different numbers of tenants.

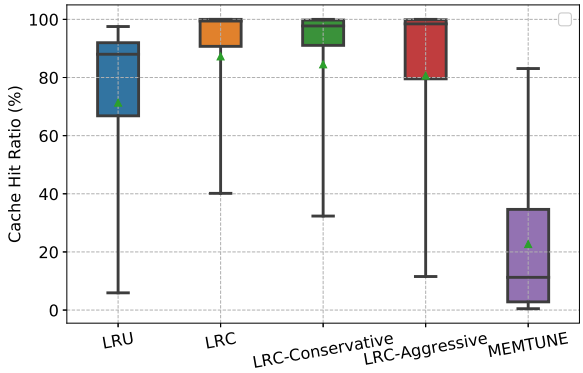


Fig. 21. Cache hit ratio under different cache policies with the Alibaba trace. Boxes show the 25th percentile, mean, and 75th percentile, and whiskers depict the 5th and 95th percentiles. Quantiles are the same in Figs. 22 and 23.

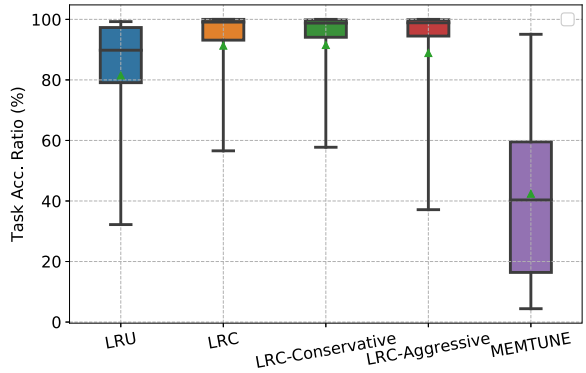


Fig. 22. Task acceleration ratio under different cache policies with the Alibaba trace.

dynamic tuning of memory fractions and data prefetching are also applicable to LRC policies.

Notice that there is no performance difference between LRC policies and their online versions, as the 100 jobs are replayed one after another and there is no data dependency between jobs. In this case, the dependency information obtained online is exactly the same as the offline knowledge.

Results. Fig. 21 shows the boxplot of the cache hit ratio across the 100 jobs profiled in the Alibaba trace. The baseline LRC policy achieves the highest average cache hit ratio (87.3 percent), outperforming LRU (71.3 percent cache hit ratio) by 22.3 percent and MEMTUNE by 284.4 percent (22.7 percent cache hit ratio). The cache replacement strategy of MEMTUNE does not perform well in our evaluation as in the production traces, it is common to have the same data blocks repeatedly referenced by many tasks across multiple stages of a job. With MEMTUNE, *active* data blocks get *evicted* after their first usage, leading to low cache hit ratio. In contrast, LRC maintains a big picture as it tracks the dependency relationship within the entire job. Hence, LRC will not mistakenly evict out the data with frequent usage in the coming compute stages. LRC with conservative and aggressive replacement have a slightly lower average cache hit ratio than the LRC baseline, a price paid for the fulfillment of the all-or-nothing requirements.

Figs. 22 and 23 show the boxplots of task and stage acceleration ratios, respectively. As expected, LRC with conservative (aggressive) replacement achieves the highest average task (stage) acceleration ratio. Compared with MEMTUNE (LRU), LRC with conservative replacement achieves 116.0 percent (12.3 percent) improvement in terms of the average task acceleration ratio. Regarding the stage acceleration ratio, LRC with aggressive replacement outperforms MEMTUNE (LRU) by 114.7 percent (33.3 percent). The stage acceleration ratio of LRC with aggressive replacement is much better than that of the baseline LRC policy. In particular, the tail (5th percentile) stage acceleration ratio is improved by 20.9 percent (from 34.0 percent to 41.1 percent). These results demonstrate the necessity of fulfilling the all-or-nothing caching requirements in order to effectively speed up the computation in production clusters.

While the SparkBench jobs in Section 5.2 exhibit limited all-or-nothing requirements, we have found strong evidence in the Alibaba production workloads that the all-or-nothing caching requirements should be taken care of. Therefore, the all-or-nothing policies should be considered in a workload-specific manner. Nonetheless, the evaluation results with both the SparkBench and the Alibaba trace have shown the performance advantage of LRC over the other existing cache replacement policies.

6 Related Work

Traditional Caching on a Single Machine. Memory caching has a long history and has been widely employed in storage systems [28], databases [29], file systems [30], web servers [31], operating systems [32], and processors [33]. Over the years, a vast amount of caching algorithms have been proposed. These algorithms run on a single machine and can be broadly divided into two categories:

Recency/frequency-based policies: LRU [23] and LFU [10] are the two widely used caching algorithms. As shown in Section 2, neither algorithm adapts well to the data access pattern in data analytic clusters even though they are simple to implement. Recency and frequency have also been combined in many caching policies. For example, Least Recently/Frequently Used (LRFU) [34] and its adaptive version [35] dynamically adjust between LRU and LFU.

Hint-based policies: Many cache policies evict/prefetch data blocks through hints from applications [36], [37],

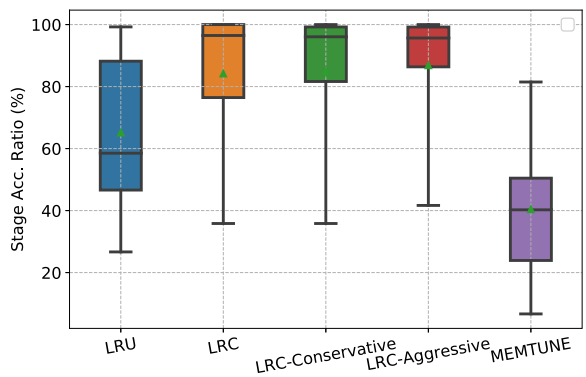


Fig. 23. Stage acceleration ratio under different cache policies with the Alibaba trace.

which are provided by the programmers to indicate what data will be referenced again and when. Nevertheless, inserting such hints can be difficult to the programmer, who has to carefully examine the underlying data access pattern. As an alternative approach, hints can also be inserted by compliers [15] or through predictions [20]. Yet, these techniques cannot handle complex, input-dependent access patterns.

To summarize, traditional caching policies do not assume application semantics, nor do they factor in the data parallel nature of cluster applications.

Cache Management in Parallel Processing Systems. Despite the significant performance impact of memory caches, cache management remains a relatively unchartered territory in data analytics systems. Prevalent analytics frameworks (e.g., Spark [4], Tez [15], and Tachyon [6]) simply employ LRU to manage cached data on cluster machines, which results in a significant performance loss [5], [19].

To our knowledge, MEMTUNE [19] is the most closely related work to LRC, which also leverages the application semantics for cache management. Specifically, MEMTUNE employs three orthogonal techniques to improve memory management: dynamically adjusting the memory fraction for task computation and data caching, prefetching data for coming tasks, and employing a replacement strategy that seeks to evict the input data of finished tasks. We have shown through trace-driven simulations (Section 5.3) that the replacement strategy of MEMTUNE may mistakenly evict out active data, leading to lower cache hit ratios than the LRC policies. That said, the other two orthogonal techniques of MEMTUNE can still be applied to LRC to further improve the memory efficiency. As a matter of fact, starting from version 1.6, Spark now natively supports the dynamic tuning of memory share for computation and data caching.

Caching for all-or-Nothing Requirement. 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.

There are also existing works [38], [39] studying the benefits of scheduling tasks in the same compute stage for DAG-based workloads. However, the all-or-nothing requirements are much more challenging for data caching than for task scheduling. In data analytics applications, a data block is typically referenced by multiple compute stages as input. Therefore, to meet the all-or-nothing caching requirements, LRC needs to traverse all available DAGs to get the dependency relationship across blocks and judiciously determine what dataset to cache. In contrast, a compute instance can only be scheduled to execute a single task at a time. For the task scheduler, the all-or-nothing policy can be easily implemented in a greedy manner, i.e., scheduling tasks in the current compute stage all at once without concerning about the future stages in the DAGs.

7 Concluding Remarks

In this paper, we proposed a dependency-aware cache management policy, Least Reference Count (LRC), which evicts data blocks whose reference count is the smallest. With LRC, inactive data blocks can be timely detected and evicted, saving cache spaces for more useful data. In addition, we identified the defining two-level all-or-nothing property in data analytics systems and tailored the LRC policy with conservative/aggressive replacement to address this requirement under different cluster environments. We have implemented LRC as a pluggable cache manager in Spark, and evaluated its performance through EC2 deployment. Experimental results showed that the LRC policies with conservative/aggressive replacement is able to address the all-or-nothing caching requirements given different provisioning of compute resources in the cluster. Compared to the popular LRU policy, our LRC implementation achieves the same application performance at the expense of only 40 percent of cache spaces. When using the same amount of memory caches, LRC reduces the application runtime by up to 60 percent. In large-scale simulations feeding production traces, LRC improves the caching performance by 22 and 282 percent over LRU and a recently proposed caching policy called MEMTUNE, respectively.

Acknowledgments

This work was supported in part by RGC ECS grant under contract 26213818. Yinghao Yu and Chengliang Zhang were supported by the Hong Kong PhD Fellowship Scheme.

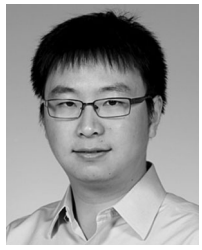
References

[1] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LRC: Dependency-aware cache management for data analytics clusters," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[2] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LERC: Coordinated cache management for data-parallel systems," in *Proc. IEEE Global Commun. Conf.*, 2017, pp. 1–6.
[3] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. USENIX Conf. Operating Syst. Design Implementation*, 2010, pp. 293–306.
[4] 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 Conf. Netw. Syst. Design Implementation*, 2012, pp. 2–2.
[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 Conf. Netw. Syst. Design Implementation*, 2012, pp. 20–20.
[6] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–15.
[7] Memcached, 2009. [Online]. Available: https://memcached.org
[8] Redis, 2016. [Online]. Available: http://redis.io
[9] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306, 1993.
[10] 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.

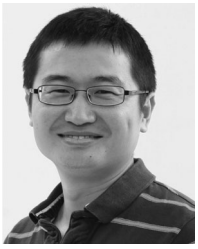
[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *Proc. 18th IEEE Annu. Joint Conf. IEEE Comput. Commun. Soc.*, 1999, pp. 126–134.
[12] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surveys*, vol. 35, no. 4, pp. 374–398, 2003.
[13] Storm, 2012. [Online]. Available: http://storm.apache.org/
[14] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: Increased performance for in-memory hadoop jobs," *VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.
[15] 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 Int. Conf. Manage. Data*, 2015, pp. 1357–1369.
[16] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, Art. no. 53.
[17] Alluxio, 2016. [Online]. Available: http://www.alluxio.org/
[18] Alibaba Trace, 2018. [Online]. Available: https://bit.ly/2Uobjr1
[19] 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 *Proc. IEEE Int. Parallel Distrib. Process. Symp. Des.*, 2016, pp. 383–392.
[20] Cache Policy Simulator, 2019. [Online]. Available: https://bit.ly/2uVjjAk
[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[22] "Amazon S3," 2013. [Online]. Available: https://aws.amazon.com/s3/
[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.
[24] Amazon Elastic Compute Cloud, 2016. [Online]. Available: https://aws.amazon.com/ec2/
[25] Spark API Documentation, 2018. [Online]. Available: https://spark.apache.org/docs/latest/api.html
[26] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 99–112.
[27] Coflow Benchmark, 2015. [Online]. Available: https://bit.ly/2OXks3M
[28] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Comput. Surveys*, vol. 26, no. 2, pp. 145–185, 1994.
[29] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412–418, 1981.
[30] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Trans. Comput. Sys.*, vol. 6, no. 1, pp. 134–154, 1988.
[31] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. Usenix Symp. Internet Technol. Syst.*, 1997, pp. 193–206.
[32] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. Conf. Innovative Data Syst. Res.*, 2011.
[33] J. L. Henning, "Spec cpu2000: Measuring CPU performance in the new millennium," *Comput.*, vol. 33, no. 7, pp. 28–35, 2000.
[34] 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. Comput.*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.
[35] D. Lee, J. Choi, J. Kim, et al., "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 1999, pp. 134–143.
[36] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Trans. Comput. Sys.*, vol. 14, no. 4, pp. 311–343, 1996.
[37] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. 15th ACM Symp. Operating Syst. Principles*, 1995, pp. 79–95.
[38] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 9, pp. 951–967, Sep. 1994.
[39] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

Yinghao Yu received the BS degree in electronic engineering from Fudan University, in 2015. He is currently working toward the PhD degree in the Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology. His research interests include the general resource management in big data systems, with a special focus on the performance optimization of cluster caching. He is a student member of the IEEE.

Chengliang Zhang received the BEng degree from the School of Software at Harbin Institute of Technology, China. Since 2016, he has been working toward the PhD degree in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests cover cloud computing and distributed systems. He currently focuses on system design and performance optimization issues in distributed machine learning and data analytics systems. He is a student member of the IEEE.



Wei Wang (S'11-M'15) received the BEng (Hons.) and MEng degrees from the Department of Electrical and Computer Engineering, Shanghai Jiao Tong University, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2015. He is an assistant professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). He is also affiliated with HKUST Big Data Institute. His research interests cover the broad area of distributed systems, with special emphasis on big data and machine learning systems, cloud computing, and computer networks in general. He was a recipient of the prestigious Chinese Government Award for Outstanding PhD Students Abroad in 2015 and the Best Paper Runner-up Award at USENIX ICAC 2013. He was recognized as the Distinguished TPC Member of the IEEE INFOCOM 2018 and 2019. He is a member of the IEEE.



Jun Zhang (S'06-M'10-SM'15) received the BEng degree in electronic engineering from the University of Science and Technology of China, in 2004, the MPhil degree in information engineering from the Chinese University of Hong Kong, in 2006, and the PhD degree in electrical and computer engineering from the University of Texas at Austin, in 2009. He is an assistant professor with the Department of Electronic and Information Engineering, Hong Kong Polytechnic University (PolyU). His research interests include wireless communications and networking, mobile edge computing and edge learning, distributed learning and optimization, and big data analytics. He co-authored the books Fundamentals of LTE (Prentice-Hall, 2010), and Stochastic Geometry Analysis of Multi-Antenna Wireless Networks (Springer, 2019). He is a co-recipient of the 2019 IEEE Communications Society & Information Theory Society Joint Paper Award, the 2016 Marconi Prize Paper Award in Wireless Communications (the best paper award of IEEE Transactions on Wireless Communications), and the 2014 Best Paper Award for the EURASIP Journal on Advances in Signal Processing. Two papers he co-authored received the Young Author Best Paper Award of the IEEE Signal Processing Society in 2016 and 2018, respectively. He also received the 2016 IEEE ComSoc Asia-Pacific Best Young Researcher Award. He is an editor of the *IEEE Transactions on Wireless Communications*, the *IEEE Transactions on Communications*, and the *Journal of Communications and Information Networks*. He is a senior member of the IEEE.

Khaled Ben Letaief (S'85-M'86-SM'97-F'03) received the BS degree with distinction in electrical engineering from Purdue University at West Lafayette, Indiana, in Dec. 1984, and the MS and PhD degrees in electrical engineering from Purdue University, in Aug. 1986, and May 1990, respectively. He is an internationally recognized leader in wireless communications and networks with research interest in wireless communications, artificial intelligence, big data analytics systems, mobile edge computing, 5G systems and beyond. In these areas, he has more than 600 journal and conference papers and given keynote talks as well as courses all over the world. He also has 15 patents, including 11 US patents. He is well recognized for his dedicated service to professional societies and in particular IEEE where he has served in many leadership positions, including President of IEEE Communications Society, the worlds leading organization for communications professionals with headquarters in New York and members in 162 countries. He is also the founding editor-in-chief of the *IEEE Transactions on Wireless Communications* and served on the editorial board of other premier journals including the IEEE Journal on Selected Areas in Communications Wireless Series (as editor-in-chief). He has been a dedicated teacher committed to excellence in teaching and scholarship with many teaching awards, including the Michael G. Gale Medal for Distinguished Teaching (Highest HKUST university-wide teaching award and only one recipient/year is honored for his/her contributions). He is the recipient of many other distinguished awards including the 2019 Distinguished Research Excellence Award by HKUST School of Engineering (Highest research award and only one recipient/3 years is honored for his/her contributions); 2019 IEEE Communications Society and Information Theory Society Joint Paper Award; 2018 IEEE Signal Processing Society Young Author Best Paper Award; 2017 IEEE Cognitive Networks Technical Committee Publication Award; 2016 IEEE Signal Processing Society Young Author Best Paper Award; 2016 IEEE Marconi Prize Paper Award in Wireless Communications; 2011 IEEE Wireless Communications Technical Committee Recognition Award; 2011 IEEE Communications Society Harold Sobol Award; 2010 Purdue University Outstanding Electrical and Computer Engineer Award; 2009 IEEE Marconi Prize Award in Wireless Communications; 2007 IEEE Communications Society Joseph LoCicero Publications Exemplary Award; and over 15 IEEE Best Paper Awards. From 1990 to 1993, he was a faculty member at the University of Melbourne, Australia. Since 1993, he has been with HKUST where he has held many administrative positions, including the Head of the Electronic and Computer Engineering department. He also served as Chair Professor and HKUST Dean of Engineering. Under his leadership, the School of Engineering dazzled in international rankings (rising from # 26 in 2009 to # 14 in the world in 2015 according to QS World University Rankings). From September 2015 to March 2018, he joined HBKU as Provost to help establish a research-intensive university in Qatar in partnership with strategic partners that include Northwestern University, Carnegie Mellon University, Cornell, and Texas A&M. He is a fellow of the IEEE and a fellow of HKIE. He is also recognized by Thomson Reuters as an ISI Highly Cited Researcher.