Improving Resource Efficiency of Performance-Optimized In-Memory Big Data Analytics With a Hybrid Static Dynamic Approach

Le-Le Li, Wei-Chung Hsu, *Member, IEEE*, Yeh-Ching Chung, *Senior Member, IEEE*, and Zhi-Bin Yu, *Member, IEEE*

***Abstract*—Leveraging dynamic resource allocation (DRA) or machine learning (ML) to tune the configuration parameters of big data analytical applications (e.g., Apache Spark) for optimal performance is popular currently. However, we find that big data applications tuned by DRA or ML techniques typically waste substantial resources. To address this issue, we devise a `hybrid static dynamic` mechanism to squeeze resources from these applications without degrading performance. The static mechanism, named GAP, squeezes the gap amount of resources between an application's peak used and allocated resource amounts before it starts its next execution. The dynamic mechanism, called α-CAP, dynamically determines whether more resources can be squeezed in a fine-grained time interval manner. We implement our GAP and α-CAP mechanisms atop YARN and call it PRESSER. We employ all 103 TPC-DS queries and six Hibench workloads to evaluate it. The results show that PRESSER can improve the average CPU and memory utilization of our cluster by 31% and 20%, respectively. Consequently, it doubles the throughput of the experimented Spark cluster without degrading performance. Furthermore, compared to two most recent performance-resource co-optimization frameworks, PRESSER squeezes at least 1.5$\times$ more CPU or memory resources than them with high performance guarantee.**

***Index Terms*—Data analytics, resource efficiency, spark, performance-resource optimization.**

I. Introduction

BECAUSE of the higher performance compared to Apache Hadoop, Apache Spark [1] has become the de facto standard of big data analytical platform in cloud computing. A typical Spark application generally runs in an iterative manner (e.g., every hour or day) [2], [3], [4], each run with different data. One Spark cluster typically consists of hundreds or even thousands of dedicated physical servers [5]. Spark applications therefore consume a large portion of CPU cycles as well as memory in cloud data centers.

Making the execution time of a Spark application as short as possible is important for both customers and cloud providers. A large body of performance optimization studies have therefore been conducted by tuning the configuration parameters. These studies can be classified into rule-based [6], cost-model-based [7], [8], and machine learning-based [4], [9], [10], [11] approaches. Currently, machine learning-based approaches are increasingly popular. Cloud giants such as Microsoft and Alibaba pervasively employ dynamic resource allocation (DRA) or machine learning (ML) to tune the configuration parameters of their in-memory big data analytical applications (e.g., Spark) for optimal performance [2], [12]. We call these tuned applications MLOS (ML-Optimized Spark) or DRAOS (Dynamic Resource Allocation Optimized Spark) applications.

Optimizing resource efficiency is another concern for cloud operators. For the same performance of an application, cloud providers hope it consumes resources as few as possible. For the same amount of resources, cloud providers hope as more as possible applications can be served. In other words, the resource utilization should be as high as possible. However, even with the state-of-the-art cluster management and scheduling techniques, the cluster resource utilization is still not as high as desired (e.g., $< 45\%$) [13]. However, we find that MLOS or DRAOS applications do not necessarily optimize resource efficiency. Instead, they waste substantial resources. For instance, we observe that over 40% of the allocated memory is wasted for MLOS applications on average in our experiments (see Section II-C).

On the other hand, recent studies, including Restune [14], GeneralBO [11], and UDAO [15], [16], try to co-optimize the performance and resource efficiency at the same time for Spark applications. However, Restune and GeneralBO rely on ML-based performance models which need a long time (high overhead) to collect a large number of training examples. UDAO designs a unified framework using multi-objective optimization to co-optimize the performance and CPU cost of a Spark application. However, it could not improve resource efficiency for the application during execution.

To improve resource efficiency of performance-optimized Spark applications, we face three challenges. The first one is how to quickly quantify the over-allocated amount of CPU and memory resources for a MLOS or DRAOS application before

Received 14 October 2024; revised 28 March 2025; accepted 6 June 2026. Date of publication 1 July 2026; date of current version 11 August 2026. An earlier version of this paper was presented at the IEEE Publication Technology Group [DOI: XXXX]. Recommended for acceptance by M. Kaaniche. *(Corresponding author: Zhi-Bin Yu.)*

Le-Le Li is with the School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen 518172, China.

Wei-Chung Hsu and Yeh-Ching Chung are with the School of Data Science, The Chinese University of Hong Kong, Shenzhen 518172, China.

Zhi-Bin Yu is with the Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS), Shenzhen 518055, China (e-mail: zb.yu@siat.ac.cn).

Digital Object Identifier 10.1109/TC.2026.3704381

execution. Second, during execution, we observe that there is a large and varying unused resource amount between allocated and actual used ones for a MLOS or DRAOS application. It is challenging to dynamically squeeze unused resources from the application during execution without degrading its performance. Third, since the total squeezed amount of resources dynamically varies because the squeezed applications exit randomly, it is challenging to correctly re-provision them for co-located applications to improve cluster utilization.

To address the above issues, we propose a hybrid static dynamic mechanism named PRESSER to optimize the resource including CPU and memory efficiency of MLOS or DRAOS applications with low overhead. The static mechanism, named GAP, squeezes the gap amount of resources between a MLOS or DRAOS application's allocated and peak used resource amount before it starts its next execution. The dynamic mechanism, called α-CAP dynamically determines whether more resources should be further squeezed from it or not during execution. If the application's used resource amount is less than the α percentile of the allocated one, α-CAP squeezes resources from it. The squeezed amount is the gap amount between the α percentile allocated and the used resource amounts. Otherwise, α-CAP would not squeeze resources. Moreover, we devise a resource pool to store the squeezed resources and supply them dynamically for other co-located applications to improve cluster resource utilization.

In particular, this paper makes the following contributions.

- We find that optimizing performance of a Spark application (MLOS or DRAOS application) by ML-based configuration tuning or dynamical resource allocation does not necessarily optimize its resource efficiency.
- We propose a hybrid static dynamic mechanism named PRESSER to safely squeeze resources from MLOS or DRAOS applications without degrading their performance, and provide the squeezed resources for other co-located applications.
- We evaluate PRESSER on a 6-node cluster by all 103 SparkSQL queries from TPC-DS [17] and six representative workloads from Hibench [18]. The results show that PRESSER can squeeze 29% and 42% of CPU and memory on average from performance-optimized applications, respectively. By reprovisioning these squeezed resources to run more applications, PRESSER doubles the throughput of the cluster, significantly improving resource efficiency. As a result, PRESSER can improve the average utilization of CPU and memory by 31% and 20% in our cluster, respectively.
- We compare PRESSER against two recent performance-resource co-optimization techniques, GeneralBO [11] and UDAO [15] for Spark. The results show that PRESSER can save at least 1.5× resources than them without degrading the MLOS or DRAOS applications' performance.

The rest of this paper is organized as follows. Section II depicts the background and motivation. Section III describes the system design. Section IV depicts the experimental setup and Section V presents the results and analysis. Section VI describes the related work and Section VII makes a conclusion.

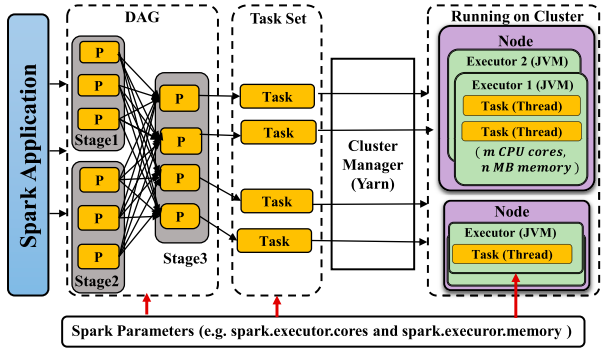

Fig. 1. An overview of spark running on Hadoop Yarn.

II. Background and Motivation

A. Performance Guarantee for Spark Data Analytics

Apache Spark [1] has become increasingly popular for big data analytics in cloud. Compared to Hadoop, Spark leverages a data structure RDD (Resilient Distributed Dataset) that can be computed in parallel to achieve significantly higher performance. As Fig. 1 shows, a Spark application comprises a DAG (Directed Acyclic Graph) which consists of a set of stages. Each stage consists of a number of RDDs. At runtime, a stage contains multiple parallel tasks and each task processes a data partition in a RDD (e.g., P in Fig. 1).

The Spark scheduler submits the tasks within a stage to a cluster manager to execute concurrently. A number of Spark executors are launched on a host and each executor contains one or more Spark tasks. Concretely, each executor performs as a JVM process with a fixed amount of resources (e.g., m CPU cores and n MB memory of Executor 1 in Fig. 1) and each task is a thread.

The resource efficiency, as well as performance of a Spark application are influenced by a large number of configuration parameters [9] because they specify the resource amount (e.g., CPU core
and memory size) used by an application [2], [4]. For example, *spark.executor.cores* specifies the number of CPU cores allocated to an executor and *spark.executor.memory* specifies its memory size.

Currently, tuning the configuration parameters is widely used for cloud providers to achieve near-optimal performance and guarantee the SLA (Service Level Agreement) for customers [11], [13]. In this paper, we assume the SLA for a Spark application is that its runtime should be below a user-specified threshold, which is also in line with prior work [3], [19]. We set different SLAs obtained from the test by Databricks [20] for various queries (e.g., 57 seconds for TPC-DS Q1), as shown in Fig. 2. Prior studies have proposed many configuration tuning approaches to guarantee SLA. Among these, the ML-based ones are widely used by cloud giants, such as Tencent and Alibaba [2], [11]. We call such Spark applications tuned by machine learning (ML)-based methods MLOS (ML-Optimized Spark) ones. Another popular resource configuration tuning approach is *Dynamic Resource Allocation* (DRA) [21], which is a rule-based tuning approach provided by Apache Spark. We

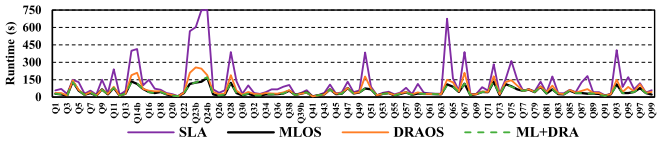

Fig. 2. Both MLOS and DRAOS TPC-DS queries guarantee the performance SLA given by DataBricks [20]. DRA+ML-optimized (ML+DRA) spark applications achieve similar performance with MLOS ones.

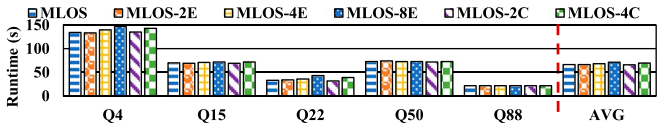

Fig. 3. Execution time comparison for five MLOS applications with/without removing resources. MLOS-2E, MLOS-4E, MLOS-8E denote removing two, four, and eight executors per MLOS application, respectively. MLOS-2C and MLOS-4C represent removing (2 CPU cores, 2 GB RAM) and (4 CPU cores, 4 GB RAM) from each executor.

TABLE I
INTRODUCTION TO MLOS AND DRAOS APPLICATIONS

	MLOS	DRAOS
Similarities	Performance optimization by configuration tuning	Performance optimization by configuration tuning
Differences	Leveraging ML to tune a large number of configuration parameters	Scaling executors up and down during execution based on pending tasks
Examples	LOCAT [9], DAC [4]	DRA [21]
Advantages	Considering the complex impact on performance from a large number of parameters (e.g. over 30 parameters [11])	Easily deploy and adaptive to workload changes
Disadvantages	Multiple executions are required to find near-optimal parameters for an application	Prone to over-request executors due to maximum threshold as $2^{31}-1$ [11]

use the term DRAOS to represent the applications tuned by DRA. DRAOS is also widely used in industry due to its ease of use, such as Microsoft [12]. In short, Table I summarizes the similarities, differences, advantages, and disadvantages between MLOS and DRAOS applications.

We conduct an experiment in our 6-node cluster to show that the performance of a DRAOS or MLOS TPC-DS query can satisfy its performance SLA tested by DataBricks [20], as shown in Fig. 2. In detail, the P50 (50-percentile), P75, P90, P99, and maximum SLAs over these 103 TPC-DS queries are 58, 130, 387, 747, and 750 seconds, respectively. In contrast, the P50, P75, P90, P99, and maximum execution times for MLOS queries are 31, 66, 111, 134, and 167 seconds, respectively. As for DRAOS applications, these execution times are 41, 67, 150, 244, and 256 seconds, respectively. Although DRA+ML-optimized (ML+DRA) applications can be feasible, their performance is close to MLOS because MLOS applications have already fine-tuned the resource related parameters for a Spark workload, making DRA less effective.

B. Recurrent and Long-Running Data Analytics

A large fraction of data analytics including MLOS or DRAOS applications in shared-nothing enterprise clusters are usually periodic and long-running [3], [4], [11], [13], [19], [22]. That is, hundreds of thousands of MLOS or DRAOS jobs periodically run on dedicated clusters where container technology like Docker [23] is widely deployed and each Spark executor runs within a container, such as MaxCompute of Alibaba [24]. For example, 75% of more than 1.7 million daily Spark jobs are periodic (hourly, daily, etc) at ByteDance [13]. Furthermore, most customers at Azure HDInsight use Spark SQL to run their ETL workloads periodically [22]. Additionally, these recurrent data analytics are usually stable and long-running [25]. For example, a previous analysis [22] of 90,224 daily Spark production applications across 3,245 clusters at Microsoft shows that around 70% of the applications do not share compute resources with others in the same cluster during execution. Besides, Morpheus [3] presents that the average runtimes of five typical types of Spark jobs at Microsoft are 173s, 605s, 1400s, 6300s, and 24570s.

C. Motivation

Although MLOS or DRAOS applications can satisfy SLA well, we observe salient resource inefficiency exists in these applications by the following experiments.

1) MLOS Applications With Fewer Resources: We conduct an experiment by providing fewer CPU and memory resources for MLOS applications to observe how their performance changes. The input data sizes for all these MLOS applications are 500GB. To represent different resource utilization for Spark applications, we select three queries with pronounced resource bottlenecks from the TPC-DS [26] and make them as MLOS applications as described in Section IV-B. They are $Q4$, $Q22$, and $Q88$. $Q22$ is *CPU-bound*, $Q4$ is *memory-bound*, and $Q88$ is regarded as *IO-bound* according to the analysis in [26]. In addition, we randomly pick up two other queries (e.g., $Q15$ and $Q50$) from TPC-DS.

We try two approaches to provide less CPU and memory resources for a Spark application: (1) decreasing the number of Spark executors (e.g., MLOS-E shown in Fig. 3) or (2) decreasing the number of CPU cores and memory size per Spark executor (e.g., MLOS-C shown in Fig. 3). Fig. 3 compares the execution times of five example MLOS applications with or without providing less resources using the two approaches above. Each application runs seven times and the average execution times are plotted.

We made a couple of interesting observations here. For one, MLOS-2E, MLOS-2C, and MLOS applications without resources reduction achieve almost the same execution time on average. This indicates removing a small amount of resources yields marginal performance degradation and still guarantees SLA. Second, by removing more resources from these MLOS applications, their performance degrades differently. In detail, MLOS-8E or MLOS-4C produces only moderate performance degradation (i.e. $< 5\%$ slowdown) for three MLOS queries while making significant slowdowns for $Q4$ and $Q22$. In particular, MLOS-8E results in 9% and 30% slowdowns for $Q4$ and $Q22$ compared to the MLOS ones, respectively. Note that these

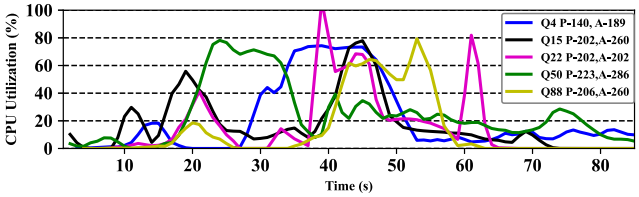

Fig. 4. CPU utilization of five example MLOS applications (y-axis) and x-axis denotes their execution times in seconds. The CPU utilization is calculated by using the number of allocated CPU cores to divide that of the used ones. $P-202$ denotes the number of *peak* used CPU cores by $Q15$ is 202 and $A-260$ denotes the allocated number of CPU cores for $Q15$ is 260. The maximum CPU utilization (78%) for $Q15$ is given by the ratio of the number of peak used CPU cores (202) over that of allocated CPU cores (260). the same applies to other MLOS applications.

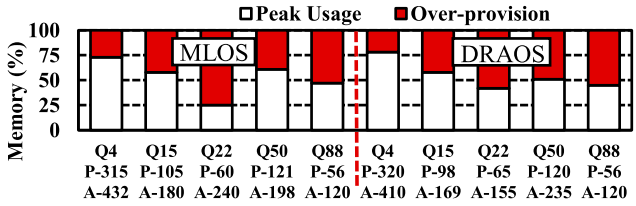

Fig. 5. The memory utilization of five MLOS (left) and DRAOS (right) applications. P - Peak used memory (GB), A - Allocated memory (GB).

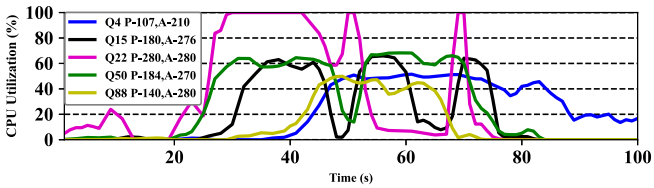

Fig. 6. CPU utilization of five example DRAOS applications with input data of 500 GB and x-axis denotes their runtimes in seconds.

numbers might be significantly different on different hardware platforms since performance related metrics highly depend on the hardware capabilities. In summary, these observations indicate that it is highly possible to squeeze an appropriate amount of resources from each MLOS application to improve resource efficiency while guaranteeing SLA.

2) CPU Utilization of MLOS Applications: To answer why squeezing some resources does not influence a MLOS application's performance, we observe its resource utilization during execution. We define CPU utilization by using the number of allocated CPU cores to divide that of the used ones. Fig. 4 illustrates the CPU utilization of the same five example MLOS applications used in Section II-C1. As can be seen, the CPU utilization is generally low at most time during their executions. For example, $Q50$'s average CPU utilization is as low as 32%. For another example, although the peak CPU utilization of the CPU-bound query, $Q22$, is the highest among them, its average CPU utilization is still only 30%.

Moreover, Fig. 4 shows that there are large gaps between the allocated number of CPU cores (e.g., $A-189$) and the number of peak used CPU cores (e.g., $P-140$) of the applications except for $Q22$. In detail, the numbers of over-allocated CPU cores for applications $Q4$, $Q15$, $Q50$, and $Q88$ are 49, 58, 63, and 54, respectively.

In addition, Fig. 4 shows that during up to 50% of $Q15$'s execution time, its actual consumed CPU cores are below 30% of its allocated ones. It is only around 8% of $Q15$'s execution time when it uses the largest number of CPU cores. In summary, the time for the five MLOS applications when they use the peak number of used CPU cores is short and the peak number is less than that of allocated CPU cores except $Q22$. Previous studies also made similar observations for non-optimized applications [25].

3) Memory Utilization of MLOS Applications: We now investigate the memory utilization of MLOS applications. As shown in Fig. 5, like CPU cores, the memory of the five example MLOS applications is severely under-utilized as well. For example, $Q88$ consumes at most 56 GB of memory during its execution which occupies only 46.7% of its allocated memory (120 GB). Among the five MLOS applications, $Q22$ has the largest over-provisioned (up to 75%) while the memory-bound query, $Q4$, achieves the smallest but still wastes 27% of its allocated memory. On average, 47.2% of the allocated memory is wasted for the five MLOS applications. This implies that substantial memory resources can also be squeezed from a MLOS application without degrading its performance. In addition, since performance optimization is the only objective to make MLOS applications, configurations with insufficient resources can not satisfy performance SLA with high probability and will not be chosen by ML-based modeling methods (e.g., Bayesian optimization [9]).

4) CPU/Memory Utilization of DRAOS Applications: After knowing the low resource efficiency of MLOS applications, we intend to know that for DRAOS applications. To this end, we reuse the aforementioned five queries again with the input data of 500 GB and observe their resource utilization. Fig. 6 shows the CPU utilization for them. Compared with CPU utilization of MLOS applications in Fig. 4, the DRAOS application, $Q22$, is able to achieve 100% utilization for a longer duration. It indicates DRAOS applications achieve better CPU resource efficiency than the MLOS ones.

However, the CPU utilization of four DRAOS applications is below 70% and there are also large gaps between the peak used and allocated CPU resources of them except $Q22$. This indicates that a large number of idle CPU cycles can be effectively saved from them and leveraged by other applications. As for memory, Fig. 5 shows that the memory utilization of the five DRAOS applications is below 60% on average. This indicates that substantial memory can be squeezed from DRAOS applications as well.

Why are some DRAOS applications over-provisioned with CPU or memory resources? The answer is that DRA uses the exponential algorithm [21] to boost up the number of Spark executors when having pending tasks. Moreover, the maximum number of executors that the cluster can allocate is $2^{31}-1$ by default. A task only needs one CPU core to execute, but a Spark executor is a resource bundle of CPU cores and memory. Thus, there would be substantial over-provisioned resources if

the number of increased executors is significantly larger than that of the pending tasks. For instance, suppose that a Spark executor maps to a task and consider a scenario where a DRAOS application has four pending tasks at a certain time interval. The DRA requests one and two other executors to run three tasks for the first and second time interval, respectively, according to its exponentially increasing policy [21]. When the third time interval arrives, this DRAOS application still has one pending task. This time, DRA still requests four new executors for only one task, which wastes resources.

5) Avoiding resource over-allocation by merely setting larger degree of parallelism than CPU core count: One may think that the resource over-allocation can be prevented by merely setting the degree of parallelism (DOP) larger than the CPU core count. Indeed, it may work in some cases. However, since DOP interacts with not only CPU core count, but also memory, it is highly possible to degrade the performance and in turn violate SLA in more cases. To confirm this, we conduct an experiment using the memory-bound query $Q4$. We first make it as a MLOS application and record its execution time, which is 134 seconds. Next we set the DOP as 3 times its allocated amount of CPU cores and run Q4 five times. We find the average execution time of $Q4$ is 187 seconds, which is larger than its SLA threshold (150 seconds) due to insufficient memory for Spark tasks during execution.

III. PRESSER Architecture

We have demonstrated that substantial CPU and memory resources are wasted by MLOS (Machine Learning based Optimized Spark) and DRAOS (Dynamic Resource Allocation Optimized Spark) applications for guaranteeing SLA. We now describe how to squeeze resources from them without violating performance SLA. To this end, we design a hybrid static dynamic mechanism named PRESSER and implement it by container technology.

A. Overview

Fig. 7 shows that PRESSER consists of four components: resource usage monitor (RUM), static resource squeezing (SRS), dynamic resource squeezing (DRS), and resource collection pool (RCP). (1) RUM monitors the number of allocated CPU cores and memory size, the number of used CPU cores and memory size during the execution of a DRAOS or MLOS application. (2) SRS firstly leverages RUM to observe the number of allocated CPU cores and memory size, as well as the number of peak used CPU cores and memory size of the last execution of an application; subsequently, SRS squeezes the gap amount of resources between the allocated and the peak used resource amount *before* the application starts its next execution. Since SRS squeezes resources from a MLOS or DRAOS application before its execution, this technique is termed as a static mechanism named *GAP*. (3) DRS uses the RUM to observe the resource usage in a fine-grained time interval (e.g., one second) during the application's execution and dynamically determines whether it should squeeze resources by detecting whether the used resource amount is less than the α (it is determined in Section III-D) percentile of allocated resource amount. As DRS squeezes resources from a MLOS or DRAOS application during execution, it is a dynamic mechanism and we name it as α - *CAP*. (4) RCP is used to store the squeezed resources and offer them for other co-located Spark applications.

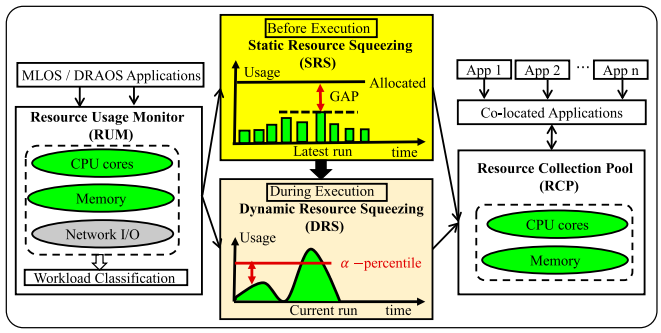

Fig. 7. An overview of PRESSER.

B. Resource Usage Monitor (RUM)

RUM monitors the following information of each MLOS or DRAOS application. First, it records the number of *allocated* CPU cores and *allocated* memory size. Second, it monitors the number of *used* CPU cores and *used* memory size in a *fine-grained* time interval (e.g., one second) manner, which is different from *coarse-fined* resource units such as the number of Spark executors in Morpheus [3]. Third, RUM calculates and records the number of peak used CPU cores, peak used memory size, α percentiles of the number of allocated CPU cores and memory size, average number of used CPU cores, and average used memory size. This data is collected by the resource usage monitoring tool of containers (e.g., docker stats [23]) and it consumes less than 0.1% CPU utilization at each time interval and uses only a few KBs to store the collected data in total.

On the other hand, MLOS or DRAOS applications run periodically with different input data sizes in production [9]. To adapt to this change, RUM constantly computes the variances of the average used CPU cores and memory sizes over the last three runs. If the variance of an application is larger than a pre-defined threshold, RUM instructs SRS to stop squeezing resources until the variance is below the threshold, guaranteeing the application's SLA.

Further, RUM has a module called *Workload Classification* to use the resource usage information to classify the monitored application into three performance bottleneck types: CPU-bound, memory-bound, and network-bound. RUM determines whether an application is bottlenecked at one resource if it achieves high utilization of that resource in a certain time window. For example, given the resource usage of a MLOS application, when its CPU utilization is above 80% and the corresponding time is longer than 50% of its total execution time, RUM classifies it as a CPU-bound one. This similar approach is used to classify memory or network-bound ones.

For network-bound queries (e.g., $Q88$), PRESSER could squeeze substantial amounts of CPU or memory resources from

Algorithm 1: Static Resource Squeezing

Input: e_n, e_c, e_m are values of executor count, core count per executor, and memory size per executor in Spark, respectively.
Input: t is CPU-bound, Memory-bound, or IO-bound.
Input: R is the fined-grained time-series of resource utilization of a MLOS or DRAOS application's latest run.
Output: s_c, s_m are the total amount of squeezed CPU cores and memory, respectively.
Output: a_c, a_m are the adjusted values of core count and memory per executor, respectively.

1: $c_a \leftarrow e_n \times e_c, m_a \leftarrow e_n \times e_m$
2: $p_c, p_m \leftarrow$ **PEAK**(R) ▷ peak CPU and memory used amounts
3: $s_c \leftarrow \max(0, c_a - p_c)$, $s_m \leftarrow \max(0, m_a - p_m)$ ▷ gap amounts
4: **if** $t \neq$ CPU-bound **then**
5: $a_c = e_c - \lfloor \frac{s_c}{e_n} \rfloor$ ▷ reduce CPU core count per executor
6: **end if**
7: **if** $s_m > e_n \times \gamma$ & $t \neq$ Memory-bound **then**
8: $a_m = e_m - \lfloor \frac{s_m - e_n \times \gamma}{e_n} \rfloor$ ▷ reduce memory size per executor
9: **end if**
10: storeSqueezedResourceInRCP(s_c, s_m)
11: **return** a_c, a_m
12: **function** **PEAK**(R)
13: $c_i \leftarrow$ time series of CPU usage extracted from R
14: $m_i \leftarrow$ time series of memory usage extracted from R
15: $p_c \leftarrow \arg\max_i c_i$
16: $p_m \leftarrow \arg\max_i m_i$
17: **return** p_c, p_m
18: **end function**

them before and during execution. For CPU-bound queries, SRS and DRS may not squeeze any CPU resources from it to protect its SLA. However, their over-allocated memory still can be squeezed. As for memory-bound queries (e.g., $Q4$), PRESSER can not squeeze memory resources from it but can squeeze CPU resources. Finally, if one query is both CPU- and memory-bound, SRS and DRS will be disabled in order to protect its performance SLA.

C. Static Resource Squeezing (SRS)

After extracting the resource usage time series of a target application (i.e., MLOS or DRAOS), as well as its bottleneck type from RUM, the GAP mechanism in SRS statically squeezes over-allocated resources from this application before its next execution, as shown in Algorithm 1. The key idea is to leverage the resource usage of the target application's latest execution to determine the amount of over-allocated resources for its next run. GAP first computes the actual allocated amount of CPU and memory resources (c_a and m_a at **Line 1**) for the application in this run. Next, given resource usages of its latest execution R, GAP checks the number of its peak used CPU cores and memory size in R **(Line 2)**. GAP in turn computes the amount of over-provisioned resources as the gap amount between the allocated and the peak used resource amount for CPU cores and memory, respectively **(Line 3)**.

Next, GAP squeezes the over-provisioned resource by reducing the resource requirement from the target application's configuration before submission (**Lines 4-10**). In detail, GAP decreases the number of CPU cores and memory size within each Spark executor to squeeze resources (**Lines 5 and 8**).

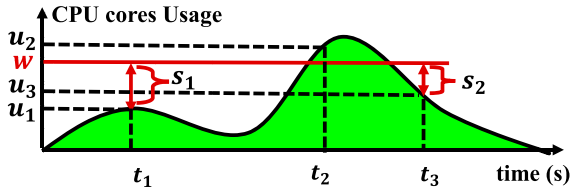

Fig. 8. How α-CAP dynamically squeezes CPU resources from a MLOS or DRAOS application per second. Y-axis denotes the number of used CPU cores. When the CPU utilization of the application is below the α percentile of allocated resource amount w, α-CAP squeezes the difference of CPU cores between w and its actual consumed CPU cores at times t_1 and t_3.

On the other hand, GAP leverages two safety strategies to guarantee the SLA of a target application. First, if this application is CPU- or memory-bound as inferred by RUM, GAP would not squeeze the corresponding resource from it (**Lines 4 and 7**). Second, as most in-memory data analytical frameworks are Java-based systems, GAP adds a safety factor γ (e.g., 500) to ensure each Spark executor has sufficient memory to avoid out-of-memory errors at the next execution. Consequently, if the amount of squeezed memory is below the guard amount of memory (e.g., $e_n \times \gamma$ MB), GAP would not squeeze any memory resource from it (**Line 7**).

D. Dynamic Resource Squeezing (DRS)

The α-CAP mechanism further dynamically squeezes the unused resources of a MLOS or DRAOS application during its execution. The resources of this application have already been squeezed by SRS and we call it a SRS squeezed application.

Fig. 8 shows how α-CAP squeezes CPU resources from a SRS squeezed application at a fine-grained time interval. At the beginning, similar to that in SRS, α-CAP would not squeeze CPU resources from the application if it is CPU-bound. Otherwise, at each interval (e.g., one second), if the number of CPU cores used by a SRS squeezed application, u, is less than the α percentile of allocated CPU core count, w, α-CAP would squeeze the gap amount of CPU cores between w and its actual CPU core count consumed from this application, which is (w - u) cores. The squeezed CPU cores are added into RCP. For example, α-CAP squeezes $(w - u_1)$ and $(w - u_3)$ of CPU cores at times t_1 and t_3, respectively, because the numbers of used CPU cores are smaller than w. If u is larger than w at time t_2, α-CAP temporarily pauses and does not squeeze any CPU cores from the application. Moreover, the RCP allocates CPU cores from the resource collection pool to this application to guarantee its SLA. A similar approach applies to dynamically squeezing more memory resources from this application.

An important question for α-CAP is how to determine the appropriate value of α without violating the SRS squeezed application's SLA. If α is too large (e.g., 100%), the application's performance would be degraded since too many resources are squeezed from it. In this case, when the amount of input data surges unexpectedly at the next interval, the SRS squeezed application is unable to allocate sufficient resources immediately to process the input data and causes performance degradation.

Algorithm 2: Determination of α in Several Executions

Input: opt denotes the optimized runtime of a MLOS or DRAOS application with SLA guarantee.
Input: jct is the runtime of its latest execution.
Input: α is initialized with 60% and $flag = False$.
Output: res is the determined value of α.

1: **function** ADJUSTPERCENTILE(jct, opt, α)
2: $\triangle loss = \frac{jct - opt}{opt}$
3: **if** $\triangle loss < 2\%$ **then**
4: $res = \alpha$
5: $r = random(0, 1)$
6: **if** $\exp(- \triangle loss) \geq r$ **then**
7: $\alpha = \alpha + 5\%$
8: **else**
9: $\alpha = \alpha + 2\%$
10: **end if**
11: $flag = True$ ▷ α has been increased
12: Run the application with updated α and record jct
13: ADJUSTPERCENTILE(jct, opt, α)
14: **else** ▷ Loss is >= 2%
15: **if** $flag = False$ **then**
16: $\alpha = \alpha - 10\%$ ▷ Decrease α to reduce loss
17: Run the application with updated α and record jct
18: ADJUSTPERCENTILE(jct, opt, α)
19: **else**
20: **return** res ▷ Determination of α
21: **end if**
22: **end if**
23: **end function**

On the other hand, a too small α yields a limited amount of squeezed CPU or memory resources.

PRESSER takes an empirical approach to determine the suitable value of α by tentatively executing a SRS squeezed application several times. Moreover, to ensure the usefulness of α value, at each execution, a CPU-bound or memory-bound application such as $Q22$ or $Q4$ is co-located with this SRS squeezed application to utilize the reclaimed resources and causes performance interference for it. Additionally, note that SRS has already squeezed some unused resources from a Spark application and in turn reduced the total amount of allocated resources. SRS can accelerate α-CAP to use less number of executions to determine suitable α values.

Algorithm 2 shows the process of adjusting α in each execution. The key idea of the algorithm is to search for an effective α value within a wide range. Before the first execution, PRESSER initializes α to 60% because the CPU or memory utilization is below 60% for more than 90% of a SRS squeezed application's execution time in our experimental cluster. For the next execution, PRESSER adjusts α based on the performance loss of its last execution (**Line 2**). If the performance loss is less than 2% (**Line 3**), it indicates that the performance of the SRS squeezed application is influenced marginally, and α grows larger to squeeze more resources. Furthermore, if the performance loss is close to 0, α increases by a big stride with high probability (e.g., 5%, **Line 7**). Otherwise, α increases by a small stride (e.g., 2%, **Line 9**). The updated α would be judged whether it is suitable or not at the next execution (**Line 13**). On the other hand, if the α does not grow and still incurs a non-negligible performance loss, α should be reduced by 10% in order to explore in a small range (**Lines 15-18**). If the performance loss is larger than 2% and α has been increased, PRESSER then sets the final value of α to the previous α of the last execution (**Line 20**) and ends the whole process.

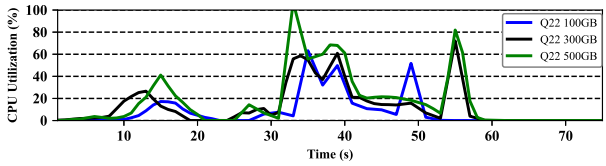

Fig. 9. Q22's CPU utilization with different input data sizes.

How to cope with different input data sizes for recurrent MLOS or DRAOS applications in DRS? In fact, α-CAP can also tolerate different input data sizes of a MLOS or DRAOS application. The reason is that the α for a smaller input data size is still likely to be useful for a larger one as the patterns of their resource usages are similar because the configuration and program codes are the same. For example, Fig. 9 plots the CPU utilization of the CPU-bound MLOS query, $Q22$, with input data sizes of 100, 300, and 500 GB, respectively. As can be seen, the CPU utilization patterns among the three datasizes are similar since the codes and configuration of $Q22$ do not change, although they have different execution times.

Further, its CPU utilization grows with the increase in input data size. Note that the time when the CPU utilization spikes up to 100% at around 32 seconds with input data size of 500 GB is very short (e.g., only 2 seconds), which has marginal impact on the overall pattern of $Q22$'s CPU utilization. The same applies to memory. These findings indicate that the α value with a small input data size can adapt to a larger one, although it might squeeze fewer resources. This is beneficial for cloud operators as data typically grows rapidly nowadays.

E. Resource Collection Pool (RCP)

Resource Collection Pool (RCP) is designed to store the squeezed resource from MLOS or DRAOS applications by the GAP and the α-CAP techniques and reprovision them for other co-located applications. As shown in Fig. 10, RCP maintains a data structure to map the application ID and information of its squeezed resources, such as the number of squeezed CPU cores (sq_cores) and memory sizes (sq_mem). RCP also monitors the running status of the squeezed applications (status). For example, if a squeezed application completes execution with status changing from Running to Exit, RCP is responsible for releasing the amount of resource squeezed from this MLOS or DRAOS application. This is done either by lowering the available amount of resources for or even canceling the execution of co-located applications (e.g., docker update [27]). Moreover, if the amount of actually used resources for the squeezed application is above the α-percentile values, RCP restores resources for it by reclaiming resources from the corresponding co-located applications to prevent its SLA violation.

In practice, DRS and RCP can be implemented using container resource control techniques (e.g., docker update [27]) to scale up or down CPU or memory resources for a MLOS or DRAOS application running within containers.

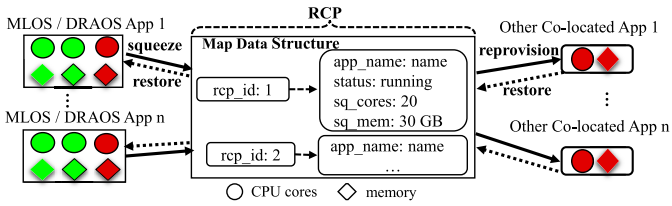

Fig. 10. An overview of RCP which stores the resources squeezed from multiple MLOS or DRAOS applications with the GAP and α-CAP mechanisms and dynamically reallocates them for co-located applications.

IV. Experimental Setup

A. Experimental Platform

Our experimental cluster contains six x86 physical servers. Each server of Ubuntu 20.04 is equipped with 48 Intel(R) Xeon(R) Silver CPU 4214 2.20 GHz cores and 128 GB DDR4 memory. We use the resource control mechanism in Docker 25.0 [27] to dynamically control the available amount of resources for an application inside a container. We deploy Apache Spark 3.4.0 atop Hadoop 3.3.6 in our cluster.

B. Representative Applications

We use all 103 TPC-DS [17] queries with 100 and 500 GB input data and Hibench [18] to evaluate PRESSER. TPC-DS stores data in a complex snowflake schema containing 24 tables, which mimics the complexity of a large retailer's sales system with different tables following different data distributions, such as Poisson and skewed distributions.

For DRAOS applications, the DRA (Dynamic Resource Allocation) mechanism is enabled in our Spark cluster, and all the aforementioned workloads are used to evaluate PRESSER. As for MLOS applications, we leverage the ML approach proposed in LOCAT [9] to make workloads as MLOS applications since LOCAT is the state-of-the-art configuration tuning study based on Bayesian optimization (BO) for performance optimization of Spark workloads. Specifically, LOCAT outperforms DAC [4] using genetic algorithms and QTune [28] based on deep reinforcement learning by factors of 2.2× and 1.9× in terms of performance speedup, respectively.

V. Results and Analysis

In this section, we first evaluate how many resources can be saved by PRESSER for DRAOS or MLOS applications with the SLA guarantee (see Section II-A). First, we evaluate the amount of squeezed resources PRESSER can save for DRAOS applications. Second, we compare PRESSER against three strategies that squeeze resources from MLOS applications. (1) **LOCAT** [9]: it is a stat-of-the-art BO-based method to optimize performance of Spark applications but does not consider any resource efficiency. (2) **Restune** [14]: it employs Constrained Bayesian Optimization (CBO) to minimize resource usage without violating the SLA constraint for cloud database systems. Concretely, we set 50 iterations for each CBO training to ensure that Restune can find the optimal resource configurations. (3) **Morpheus** [3]: it builds a resource model using integer linear programming (ILP) for a periodic job from its historical resource usage patterns. Finally, we evaluate whether PRESSER could improve resource utilization.

TABLE II
Resource Saving Ratios Among 103 DRAOS Applications With Input Data of 100 GB

R_s_ratio	Min	Avg	P50	P75	P90	P99	Max
CPU (%)	0	25	20	35	47	62	63
Memory (%)	0	29	30	37	51	74	80

TABLE III
Resource Saving Ratios Among 103 DRAOS Applications With Input Data of 500 GB

R_s_ratio	Min	Avg	P50	P75	P90	P99	Max
CPU (%)	0	26	20	36	53	58	65
Memory (%)	3	32	23	35	52	65	66

A. Resource Savings

We first evaluate the resource savings of a DRAOS or MLOS application after applying PRESSER on it with resource saving ratio defined as $R\_s\_ratio = \frac{savedRes}{allRes} \times 100\%$ where $savedRes$ denotes the amount of saved resources by PRESSER for a DRAOS or MLOS application and $allRes$ accounts for the amount of allocated resources for the DRAOS or MLOS application. Larger R_s_ratio indicates higher resource efficiency with SLA guarantee for an application.

1) DRAOS Applications: After applying PRESSER on all 103 DRAOS queries with 100 or 500 GB of input data, all these queries complete within the SLA constraints which are discussed in Section II-A. Each DRAOS query runs five times and its average saving ratio per query is computed.

Tables II and III show the minimum, average, maximum, 50-percentile (P50), P75, P90, and P99 resource saving ratios of CPU cores and memory for all 103 DRAOS queries with the input of 100 and 500 GB, respectively. For input data of 500 GB, PRESSER is able to save up to 65% and 66% of CPU and memory resources. On average, the saving ratios for CPU and memory resources are 26% and 32%, respectively. Since the average allocated amount of resources for each of the 103 DRAOS queries are 150 CPU cores and 140 GB memory, PRESSER can thus save 39 ($150 \times 26\%$) CPU cores and 45 ($140 \times 32\%$) GB memory on average per query.

In addition, since each physical node in our cluster has 48 Intel(R) Xeon(R) Silver-4214 CPU cores and 128 GB DDR4 memory, each core costs around 16.6$ and each GB memory about 2$ according to the customer prices on Intel and Amazon websites [29], [30]. PRESSER thus saves 647.4$ (16.6×39) for CPU and 90$ (2×45) for memory costs per query on average. Concretely, PRESSER saves 498, 896, 1311, 1444, 1610, and 0$ at P50, P75, P90, P99, maximum, and minimum CPU resource saving ratios, respectively. For memory, the corresponding savings are 64, 98, 146, 182, 184, and 8 dollars.

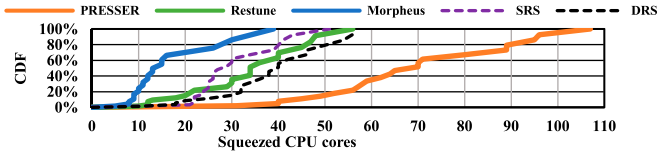

Fig. 11. Saved CPU cores CDFs of 103 MLOS applications with input data of 500 GB. Dashed lines show the effects of SRS and DRS, respectively.

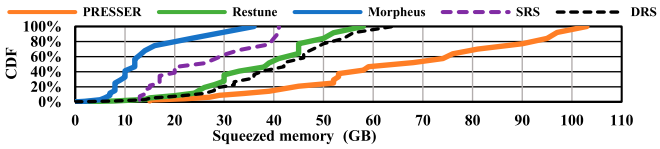

Fig. 12. Saved memory resources CDFs of 103 MLOS applications.

TABLE IV
RESOURCE SAVING RATIOS AMONG 103 MLOS APPLICATIONS

	Resource	CPU			Memory		
Datasize	R_s_ratios(%)	Min	Max	Avg	Min	Max	Avg
100G	PRESSER	0	77	23	0	79	44
	Restune	0	50	12	0	46	25
	Morpheus	0	40	9	0	49	11
500G	PRESSER	0	62	29	11	85	42
	Restune	0	47	13	0	75	21
	Morpheus	0	12	6	0	25	7

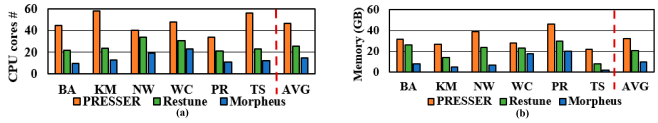

Fig. 13. Comparison for saved CPU and memory resources over six Hibench workloads. BA-Bayes, KM-Kmeans, NW-Nweight, WC-WordCount, PR-PageRank, TS-TeraSort.

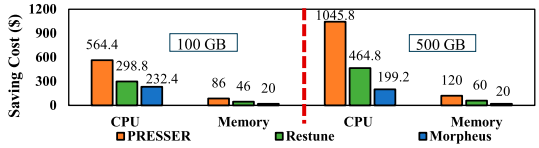

Fig. 14. Comparison of saving costs per MLOS query for PRESSER, Restune, and Morpheus with input data of 100 GB (left) and 500 GB (right).

2) MLOS Applications: For MLOS applications, PRESSER still makes their execution times satisfy their SLAs after squeezing resources. With the input data of 500 GB, Figs. 11 and 12 compare the CDFs (Cumulative Distribution Functions) of CPU and memory resources squeezed by PRESSER, Restune, and Morpheus, respectively. To sum up, the P50, P75, P90, P99, minimum, average, and maximum squeezed CPU cores for PRESSER, Restune, and Morpheus are (62, 75, 94, 104, 0, 63, 107), (30, 38, 46, 54, 0, 28, 56), and (11, 14, 24, 37, 0, 12, 39), respectively. For memory, the P50, P75, P90, P99, minimum, average, and maximum squeezed amounts for PRESSER, Restune, and Morpheus are (53, 75, 93, 101, 15, 60, 103), (30, 43, 49, 56, 0, 30, 58), and (9, 12, 16, 33, 0, 10, 36) GBs, respectively.

In particular, Figs. 11 and 12 also show the amounts of squeezed resources for static resource squeezing (SRS) and dynamic resource squeezing (DRS). We can see that DRS could squeeze more CPU and memory resources than SRS because the amount of used resources is significantly less than its α-percentile value for most time during execution. In summary, DRS contributes 57% and 60% to the total CPU and memory resource savings for an application on average, respectively. In addition, the amount of squeezed resource by SRS is larger than that of Morpheus but smaller than that of Restune while DRS outperforms both of them. This indicates that merely leveraging SRS can achieve limited amounts of squeezed resources since the gap between the allocated amount and peak used amount for a MLOS application might be small. We need to combine SRS and DRS together to improve resource efficiency significantly for the target application.

On average, PRESSER, Restune, and Morpheus save 63, 28, and 12 CPU cores, and 60, 30, and 10 GB of memory, respectively. Correspondingly, as summarized in Table IV, the CPU saving ratios made by PRESSER, Restune, and Morpheus are 29%, 13%, and 6%, and memory saving ratios of them are 42%, 21%, and 7%, respectively. This implies that PRESSER squeezes 2.2×, 4.8× more CPU cores and 2×, 6× more memory than Restune and Morpheus, respectively. For monetary savings, PRESSER saves 1045.8$ for CPU and 120$ for memory costs per query on average. In contrast, Fig. 14 shows that the CPU and memory cost savings per query achieved by Restune and Morpheus are only (464.8$, 60$) and (199.2$, 20$), which are prominently lower than that of PRESSER. In summary, the statistics of resource saving ratios are listed in Table IV and monetary savings in Fig. 14.

Moreover, we use six representative workloads from HiBench to validate PRESSER's generalizability. These six MLOS workloads are Bayes, Kmeans, Nweight, WordCount, PageRank, and Terasort, covering micro, machine learning, graph, and websearch categories. Fig. 13(a) shows that PRESSER, Restune, and Morpheus harvest 47, 26, and 15 CPU cores on average, resulting in that PRESSER saves 1.8× and 3.2× more CPU resources than baselines. Besides, Fig. 13(b) plots that PRESSER squeezes 1.5× and 3.2× more memory resources than Restune and Morepheus, respectively.

Why does PRESSER outperform LOCAT, Restune, and Morpheus significantly in saving CPU and memory resources? First, LOCAT only optimizes performance instead of resource efficiency for a Spark application. However, different amounts of resources may lead to identical performance (Section II-C1). Therefore, LOCAT may leverage more resources to achieve optimal performance, rather than using fewer resources. Second, Morpheus only adopts the coarse-grained resource unit (i.e., of Spark executors) instead of further reducing the over-allocated resources within a Spark executor. Hence it achieves limited resource savings. Third, Restune uses the CBO method

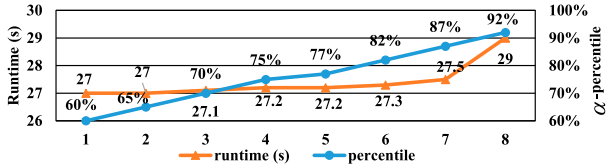

Fig. 15. Process of adaptively adjusting the value of α percentile for Q69 of DRS. The x-axis denotes the indexes of its 8 executions.

(a black-box algorithm) to tune three resource configurations of MLOS applications without SLA violations. However, since the value ranges of CPU cores and memory size can be very large (e.g., 1 to 600 CPU cores) in big data systems, it is hard for CBO to find the minimum amount of resources within 50 iterations. In contrast, PRESSER is a white-box mechanism capturing the over-provisioned amount of resources accurately just in a few iterations. Finally, both Restune and Morpheus focus on reducing over-allocated resources before execution for an application, while the α-CAP of PRESSER can further dynamically squeeze more resources from it at runtime.

B. Overhead for Determining α

How to dynamically determine a suitable value for α is essential for PRESSER's α-CAP mechanism. We first provide a case study of this determination and then discuss its overhead.

Fig. 15 shows how the α percentile of DRS is adjusted for CPU resource in eight repeated executions for $Q69$ which we made it a MLOS application and have performed PRESSER's static resource squeezing on it. The performance SLA of $Q69$ is 27 seconds (see Fig. 2). Initially, the percentile α is set with a small value (e.g., 60%) to guarantee its SLA. When the first execution terminates, the execution time of $Q69$ is 27 seconds. According to Algorithm 2, at the second execution, α is increased by 5% to become 65% such that more resources are allowed to be squeezed. The performance losses of $Q69$'s second, third, fourth, fifth, sixth, and seventh executions are still less than the threshold 2%, so that α continues to increase by 5% or 2% to allow a larger amount of resources to be squeezed from $Q69$. However, the performance loss at the eighth execution is larger than 2%. In this case, PRESSER resets the α value to 87% in the seventh iteration to avoid larger performance loss and guarantee its SLA. Applying the similar approach of $Q69$ on other MLOS and all 103 DRAOS applications, α-CAP can obtain their individually appropriate α values. Table V shows the maximum, minimum, and average α values for them.

In addition, the overhead caused by α-CAP is 5 iterations on average to converge for both DRAOS and MLOS applications. Compared to history-based offline (e.g., Morpheus) or ML approaches (e.g., Restune) requiring at least tens (e.g., 60) of executions, PRESSER takes significantly less time to squeeze the over-provisioned resources. Note that SRS has reduced the tunable range values of CPU and memory resources, helping α-CAP find suitable α values quickly. Our experiments show that α-CAP needs $2\times$ more iterations to determine the suitable α values without SRS.

TABLE V
MINIMUM, MAXIMUM, AND AVERAGE α PERCENTILE VALUES FOR 103 DRAOS AND MLOS APPLICATIONS WITH INPUT DATA OF 100 AND 500 GB. C - CPU, M - MEMORY

Type	DRAOS				MLOS			
Datasize	100GB		500GB		100GB		500GB	
Resource	C	M	C	M	C	M	C	M
Max(%)	90	75	83	87	81	75	90	72
Min(%)	0	0	0	52	0	0	0	48
Avg(%)	72	71	75	74	68	65	72	67

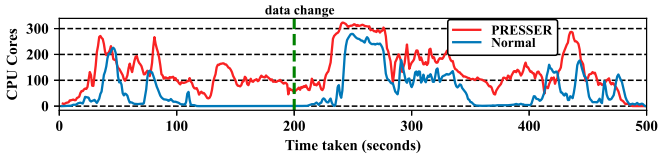

Fig. 16. CPU resource utilization comparison between PRESSER and normal with input data size changing from 100 to 500 GB at 200 seconds.

C. Cluster Resource Utilization

We next evaluate whether PRESSER could reprovision the squeezed resources to other co-located applications and improve cluster resource utilization, which consists of two phases. Taking MLOS applications for example, in the first phase, we co-run the 103 MLOS applications on our 6-node physical cluster and we call this cluster the **Normal** one. When 103 MLOS applications complete on the **Normal** cluster, we deploy PRESSER on the cluster and rerun these MLOS applications, which is the second phase. In this phase, we call our cluster the **PRESSER** one. Both the squeezed resources by GAP and by α-CAP are stored in RCP. PRESSER then leverages the squeezed resources in RCP to co-locate 103 other Spark applications simultaneously. Moreover, we also evaluate whether PRESSER can adapt to varying input data sizes by dividing the evaluation period (500 seconds) into two stages.

During the first stage from the beginning to 200 seconds, MLOS queries are submitted with the input data size of 100 GB. In the second stage (200-500 seconds), we increase the input data size from 100 to 500 GB for them, as shown by the green dashed lines in Figs. 16 and 17. However, PRESSER does not know the input data size changes. Therefore, the DRS (Dynamical Resource Squeezing, see Section III-D) of PRESSER still maintains the α values with input data of 100 GB rather than 500 GB for these submitted MLOS queries in the second stage. Because of the same configuration and same program codes for the two input data sizes, the resource usage patterns are similar, and in turn the α value can be the same, although the execution times are significantly different.

Figs. 16 and 17 plot the comparison of CPU or memory utilization of the **PRESSER** and the **Normal** clusters with SLA guarantee. First, we see that the CPU or memory utilization of the **PRESSER** cluster is significantly higher than that of the **Normal** cluster for most time in both the first and second stages. Especially, both higher CPU and memory utilization of the **PRESSER** cluster in the second stage implies that PRESSER's

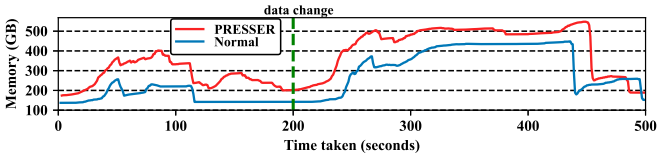

Fig. 17. Memory resource utilization comparison between PRESSER and normal with input data size varying from 100 to 500 GB at 200 seconds.

DRS is still able to squeeze CPU and memory resources to co-run more Spark applications and double the cluster throughput, although it retains the previous α values of 100 GB input data when the input data size grows to 500 GB. This confirms that the PRESSER's α-CAP technique can adapt to the input data size changes (see Section III-D).

Second, the **PRESSER** cluster uses 134 CPU cores and 363 GB memory on average per second while the **Normal** cluster only uses 47 CPU cores and 246 GB memory. Divided by the total resource capacities in our cluster, it indicates that the **PRESSER** cluster can improve 31% of CPU utilization and 20% of memory utilization per second over the **Normal** cluster. As for DRAOS applications, PRESSER can improve the average CPU and memory utilization per second by 27% and 18%, respectively, with input data of 500 GB while still protecting their SLAs.

D. Comparing With Performance-Resource Co-Optimizations

Two recent studies (i.e., UDAO [15] and GeneralBO [11]) propose to co-optimize resource and performance of Spark applications by abstracting the problem as a multi-object optimization issue. Concretely, UDAO [15] tries to find a Pareto optimal set of configurations to co-optimize both the execution time and CPU core cost of an application. GeneralBO adopts Bayesian Optimization to tune configuration parameters by formulating the product of resource consumption and execution time as the optimization objective.

Both GeneralBO and UDAO are two black-box methods based on ML modeling, requiring a number of execution trials. In contrast, PRESSER is a white-box approach. We further explore whether resource efficiency can be improved further by performing PRESSER on top of these two methods. Fig. 18 shows that PRESSER saves 22 (1.8×) and 20 (1.7×) more CPU cores than GeneralBO and UDAO on average, respectively. Moreover, Fig. 19 shows that PRESSER squeezes 17 (1.5×) and 20 (1.7×) more memory GBs than GeneralBO and UDAO on average, respectively. This indicates that PRESSER's white-box methods can further squeeze more unused resources from applications with performance-resource co-optimized by black-box methods thanks to its hybrid static and dynamic squeezing mechanism.

E. Discussion

One may think that setting the DOP (degree of parallelism) of a SparkSQL query to 3 times the allocated CPU core count before making it a MLOS application can prevent the CPU cores from being over-allocated. We conduct experiments to explore

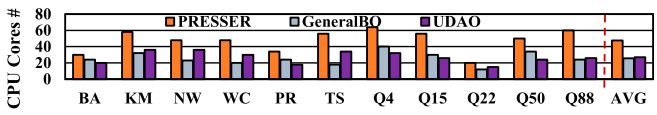

Fig. 18. CPU resource savings over TPC-DS queries and Hibench workloads for PRESSER, GeneralBO, and UDAO.

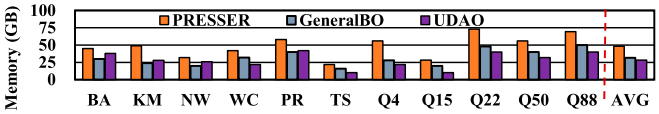

Fig. 19. Memory resource savings.

this thought by using $Q4$. We first set the parallelism to 270 which is 3 times its total count of CPU cores (90 cores) allocated to $Q4$. Note that the default parallelism in SparkSQL is 200 [6]. We then use ML to optimize the performance of $Q4$, making it an MLOS application. We find the DOP for optimal performance made by ML is 120 seconds and the optimal CPU core count chosen by ML is 80. Subsequently, we apply PRESSER to this MLOS application, $Q4$. It can further reduce the CPU count to 58 without degrading performance. This indicates that using 3 times higher DOP than CPU core count does not necessarily prevent CPU core over-allocation. Why? There are two reasons. First, some SparkSQL applications are memory-bound or IO-bound, higher DOP does not help it fully utilize all the CPU cores. Second, as we show in Fig. 4, a SparkSQL application does not use all the CPU cores at each moment during its execution duration. So, from the time-line view, there is always CPU core over-allocation for many applications. Thanks to the fine-grained dynamic squeezing mechanism, PRESSER can save substantial CPU resources from the time-line view for many MLOS applications.

VI. Related Work

A. Performance Optimization of Spark Applications

A large body of approaches has been proposed to improve the performance of Spark programs to meet the SLA. In general, they can be divided into five categories: 1) Rule-based approaches (e.g., Spark DRA) [6]. 2) ML-based approaches [4], [10], [11], [31] collect a large number of training samples with different configuration settings and build performance models using machine learning algorithms. 3) Analytical model-driven approaches [7], [32], [33] model the performance of Spark workloads as a function of a number of variables, such as the number of partitioning. 4)Simulation-based approaches leverage simulation results to build the performance models [34]. 5) Dynamically resource reprovisioning [35], [36] or adaptively tuning parameters [37] for a Spark workload at runtime is useful. However, these works require to modify a large portion of Spark source codes. 6) Query-plan based methods [38], [39], [40], [41], [42] have applied deep learning or reinforcement learning to achieve high-quality execution plans and estimations of a query's execution time. For example, Sun et al.[40] develop

an end-to-end cost estimator based on deep learning to estimate both execution cost and cardinality together for DBMS queries. Stage [42] proposes a hierarchical execution time predictor to address the challenges of cold-start prediction and reliable estimation for unseen DBMS queries.

All these methods only predict or optimize the performance of data analytics. In contrast, PRESSER primarily improves resource efficiency with performance SLA guarantee.

B. Resource Efficiency Optimization

Recent studies of resource efficiency similar to PRESSER are Restune [14], GeneralBO [11], UDAO [15], and modified Spark core modules [13], [24], [43]. Restune and GeneralBO are ML-based and black-box methods to co-optimize performance and resource efficiency while PRESSER is a white-box approach. UDAO [15] aims to find a set of configuration parameters for spark data analytics through multi-objective optimization methods to optimize performance and CPU core cost together. Intelligent pooling [43] improves the cold-start process of Spark clusters rather than the job-level resource efficiency. Wu et al. [13] propose to change the interval structure of Spark workflow to improve resource efficiency, introducing ad-hoc new parameters. Lyu et al. [24] propose to recommend instance-level resource provisioning to trade-off execution time and resource cost for Spark workloads. However, their predictor adopts a deep neural network and requires a great amount of historical traces to train.

Other studies [3], [11] mainly employ the *coarse-grained* resource usage metric (e.g., Spark executor #) to reduce the over-allocated resources. Conversely, PRESSER captures the *fine-grained* resource usages, which can save more resources.

VII. Conclusion

In this paper, we propose PRESSER, a hybrid static dynamic mechanism to squeeze resources from Spark applications with optimized performance while guaranteeing SLA. Experimental results show that PRESSER improves CPU and memory cluster utilization by 31% and 20% on average, respectively.

References

[1] M. Zaharia et al., "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, 2017, pp. 469–482.

[3] S. A. Jyothi et al., "Morpheus: Towards automated SLOs for enterprise clusters," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2016, pp. 117–134.

[4] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 564–577.

[5] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-Spark: Transient computing for big data analytics," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 484–496.

[6] "Tuning - Spark 3.4.0 documentation," 2025. [Online]. Available: https://spark.apache.org/docs/latest/tuning

[7] Y. Chen, J. Lu, C. Chen, M. Hoque, and S. Tarkoma, "Cost-effective resource provisioning for spark workloads," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.*, 2019, pp. 2477–2480.

[8] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1891–1904, Jul. 2017.

[9] J. Xin, K. Hwang, and Z. Yu, "LOCAT: Low-overhead online configuration auto-tuning of spark SQL applications," in *Proc. ACM SIGMOD/PODS Int. Conf. Manage. Data*, 2022, pp. 674–684.

[10] Á. B. Hernández, M. S. Perez, S. Gupta, and V. Muntés-Mulero, "Using machine learning to optimize parallelism in big data applications," *Future Gener. Comput. Syst.*, vol. 86, pp. 1076–1092, Sep. 2018.

[11] Y. Li et al., "Towards general and efficient online tuning for spark," *Proc. VLDB Endowment*, vol. 16, no. 12, pp. 3570–3583, Aug. 2023. [Online]. Available: http://dx.doi.org/10.14778/3611540.3611548

[12] R. Sen, A. Roy, and A. Jindal, "Predictive price-performance optimization for serverless query processing," 2021, *arXiv:2112.08572*.

[13] Y. Wu et al., "Towards resource efficiency: Practical insights into large-scale spark workloads at bytedance," *Proc. VLDB Endowment*, vol. 17, no. 12, pp. 3759–3771, Aug. 2024, doi: 10.14778/3685800.3685804.

[14] X. Zhang et al., "ResTune: Resource oriented tuning boosted by meta-learning for cloud databases," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2102–2114.

[15] F. Song, et al., "Spark-based cloud data analytics using multi-objective optimization," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, 2021, pp. 396–407.

[16] K. Zaouk, F. Song, C. Lyu, A. Sinha, Y. Diao, and P. Shenoy, "UDAO: A next-generation unified data analytics optimizer," *Proc. VLDB Endowment (PVLDB)*, vol. 12, no. 12, pp. 1934–1937, 2019.

[17] "TPC-DS Benchmark V3," TPC, 2025. [Online]. Available: http://www.tpc.org/tpcds

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 41–51.

[19] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein, "SLAOrchestrator: Reducing the cost of performance SLAs for cloud data analytics," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 547–560.

[20] "TPC-DS full disclosure report," TPC, 2025. [Online]. Available: https://www.tpc.org/results/fdr/tpcds/databricks~tpcds~100000~databricks_SQL_8.3~fdr~2021-11-02~v01.pdf

[21] "Spark dynamic resource allocation," Apache, 2025. [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

[22] A. Roy et al., "SparkCruise: Workload optimization in managed spark clusters at Microsoft," *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 3122–3134, 2021.

[23] Docker Website, Aug. 2024. [Online]. Available: https://www.docker.com/

[24] C. Lyu et al., "Fine-grained modeling and optimization for intelligent resource management in big data processing," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 3098–3111, Jul. 2022, doi: 10.14778/3551793.3551855.

[25] L. Liu and H. Xu, "Elasecutor: Elastic executor scheduling in data analytics systems," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 107–120.

[26] M. Poess, T. Rabl, and H.-A. Jacobsen, "Analysis of TPC-DS: The first standard benchmark for SQL-based big data systems," in *Proc. Symp. Cloud Comput.*, 2017, pp. 573–585.

[27] "Update configuration of one or more containers." Docker, May 2024. [Online]. Available: https://docs.docker.com/engine/reference/commandline/update/

[28] G. Li, X. Zhou, S. Li, and B. Gao, "QTune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2118–2130, 2019.

[29] "Intel Xeon Silver 4214 Processor." Intel. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/193385/intel-xeon-silver-4214-processor-16-5m-cache-2-20-ghz.html

[30] "Memory RAM DDR4." Amazon, May 2024. [Online]. Available: https://www.amazon.com/Memory-Ram-DDR4/s?k=Memory+Ram+DDR4

[31] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *Proc. IEEE 38th Int. Conf. Data Eng. (ICDE)*, 2022, pp. 1995–2007.

[32] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2016, pp. 363–378.

[33] M. Kunjir and S. Babu, "Black or white? How to develop an Auto-Tuner for memory-based analytics," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1667–1683.
[34] D. Ardagna et al., "Predicting the performance of big data applications on the cloud," *J. Supercomput.*, vol. 77, no. 2, pp. 1321–1353, 2021.
[35] D. Cheng, Y. Wang, and D. Dai, "Dynamic resource provisioning for iterative workloads on Apache spark," *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 639–652, Jan.–Mar. 2023.
[36] K. Wang, M. M. H. Khan, and N. Nguyen, "A dynamic resource allocation framework for Apache spark applications," in *Proc. IEEE 44th Annu. Comput., Softw., IEEE Int. Symp. Spread Spectr. Tech. Appl. Conf. (COMPSAC)*, 2020, pp. 997–1004.
[37] C. Lyu, Q. Fan, P. Guyard, and Y. Diao, "A spark optimizer for adaptive, fine-grained parameter tuning," *Proc. VLDB Endowment*, vol. 17, no. 11, pp. 3565–3579, Jul. 2024, doi: 10.14778/3681954.3682021.
[38] "Neo: A learned query optimizer," *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, 2019.
[39] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 2361–2374, Jul. 2022, doi: 10.14778/3551793.3551799.
[40] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 307–319, Nov. 2019, doi: 10.14778/3368289.3368296.
[41] Y. Zhao, G. Cong, J. Shi, and C. Miao, "QueryFormer: A tree transformer model for query plan representation," *Proc. VLDB Endowment*, vol. 15, no. 8, pp. 1658–1670, Apr. 2022, doi: 10.14778/3529337.3529349.
[42] Z. Wu et al., "Stage: Query execution time prediction in amazon redshift," in *Proc. SIGMOD/PODS*, New York, NY, USA: Association for Computing Machinery, 2024, pp. 280–294, doi: 10.1145/3626246.3653391.
[43] D. Ravikumar et al., "Intelligent pooling: Proactive resource provisioning in large-scale cloud service," *Proc. VLDB Endowment*, vol. 17, no. 7, pp. 1618–1627, May 2024, doi: 10.14778/3654621.3654629.

Le-Le Li received the B.S. degree in software engineering from Beihang University, Beijing, in 2016, and the M.S. degree in computer applied technology from the University of Chinese Academy of Sciences, Beijing, in 2019. He is currently working toward the Ph.D. degree with the School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen. His research interests include cloud computing and big data analytics.

Wei-Chung Hsu (Member, IEEE) received the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1987. Before becoming a Professor with The Chinese University of Hong Kong, Shenzhen, in 2023, he worked with Intel, USA. From 2013 to 2021, he was a Professor with the Department of Computer Science and Information Engineering, National Taiwan University. His research interests include parallel processing, dynamic binary translation, and optimization techniques.

Yeh-Ching Chung (Senior Member, IEEE) received the Ph.D. degree in computer and information science from Syracuse University, USA, in 1992. He is a Full Professor with The Chinese University of Hong Kong, Shenzhen. His research interests include parallel and distributed processing, cloud computing, and embedded systems.

Zhi-Bin Yu (Member, IEEE) received the Ph.D. degree in computer science from Huazhong University of Science and Technology (HUST) in 2008. He is currently a Professor with Shenzhen Institute of Advanced Technology (SIAT), Chinese Academy of Sciences (CAS), China. His research interests include workload characterization and generation, performance evaluation, big data processing and so forth. He is a member of ACM. He serves for PACT2016, ICS2018, 2019, 2024, and ASPLOS2025.