Lavender: An Efficient Resource Partitioning Framework for Large-Scale Job Colocation

WANGQI PENG, YUSEN LI*, XIAOGUANG LIU, and GANG WANG, Key Laboratory of Data and Intelligent System Security, Ministry of Education, China (DISSec) of Nankai University, China

Workload consolidation is a widely used approach to enhance resource utilization in modern data centers. However, the concurrent execution of multiple jobs on a shared server introduces contention for essential shared resources such as CPU cores, Last Level Cache, and memory bandwidth. This contention negatively impacts job performance, leading to significant degradation in throughput. To mitigate resource contention, effective resource isolation techniques at the software or hardware level can be employed to partition the shared resources among colocated jobs. However, existing solutions for resource partitioning often assume a limited number of jobs that can be colocated, making them unsuitable for scenarios with a large-scale job colocation due to several critical challenges. In this study, we propose Lavender, a framework specifically designed for addressing large-scale resource partitioning problems. Lavender incorporates several key techniques to tackle the challenges associated with large-scale resource partitioning, ensuring efficiency, adaptivity, and optimality. We conducted comprehensive evaluations of Lavender to validate its performance and analyze the reasons for its advantages. The experimental results demonstrate that Lavender significantly outperforms state-of-the-art baselines. Lavender is publicly available at https://github.com/yanxiaoqi932/OpenSourceLavender.

CCS Concepts: • **Computer systems organization → Cloud Computing**; • **Computing methodologies → Planning and scheduling**.

Additional Key Words and Phrases: Workload consolidation, job assignment, large scale resource partitioning

ACM Reference Format:
Wangqi Peng, Yusen Li, Xiaoguang Liu, and Gang Wang. 2018. Lavender: An Efficient Resource Partitioning Framework for Large-Scale Job Colocation. *J. ACM* 37, 4, Article 111 (August 2018), 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Contemporary cloud datacenters are extensively utilized for executing a multitude of throughput-centric jobs [3, 15, 37, 42], encompassing services like big data analytics, machine learning, and scientific computing. To optimize hardware expenses, these jobs are commonly consolidated within physical servers. Nevertheless, the concurrent execution of multiple jobs on a shared server introduces contention for crucial shared resources, including CPU cores, Last Level Cache (LLC), and memory bandwidth [6, 7, 24, 30, 32, 34, 40]. Consequently, this resource contention detrimentally impacts job performance, leading to substantial degradation in throughput. An efficient approach for

*Yusen Li is the corresponding author.

This work is supported by National Science Foundation of China (grant numbers 62272252, 62272253, 62293510/62293513, 62141412); Fundamental Research Funds for the Central University; NSF of Tianjin 21JCYBJC00070.
Authors' address: Wangqi Peng, pengwq@nbjl.nankai.edu.cn; Yusen Li, liyusen@nbjl.nankai.edu.cn; Xiaoguang Liu, liuxg@nbjl.nankai.edu.cn; Gang Wang, wgzwp@nbjl.nankai.edu.cn, Key Laboratory of Data and Intelligent System Security, Ministry of Education, China (DISSec) of Nankai University, 38 Tongyan Road, Jinnan District, Tianjin, China, 300350.

ACM 0004-5411/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

reducing resource contention is to use the software or hardware level resource isolation techniques (such as Intel's CAT [26] and MBA [39]) to partition the shared resources among the colocated jobs. By allocating dedicated resources to each job, performance interference can be prevented or alleviated.

The existing solutions for resource partitioning can be broadly categorized into two main approaches: performance model-based methods [13, 20, 21, 28, 31, 38, 40] and online tuning-based methods [7, 8, 11, 23, 24, 27, 45]. Performance model-based methods utilize heuristic algorithms to identify the near-optimal partitioning configuration, where the partitioning configurations are evaluated by a performance model. However, this approach is limited to handling a single resource and faces challenges when resources need to be considered due to the complexity of accurately constructing performance models. On the other hand, online tuning-based methods dynamically adjust the partitioning configuration in real-time based on feedback from the actual system. These methods are capable of handling multiple resources simultaneously, and do not rely on prior knowledge of the jobs.

However, existing solutions for resource partitioning generally assume a limited number of jobs that can be colocated (often fewer than 8) [4, 5, 30, 32, 34, 40], while modern datacenters are equipped with cloud servers that have abundant resources, capable of running tens of jobs simultaneously. With such a large number of jobs, the resource partitioning problem faces several new challenges. First, the solution space increases by exponential orders of magnitude [4, 6, 32, 34], and exploring such huge search space to find an optimal solution in real time becomes impractical. Second, some shared resources can only be coarsely partitioned due to the limitation of hardware-based isolation techniques. In order to implement a fine-grained resource allocation, such resources have to be partially shared, making the partitioning problem more complicated. Third, since more jobs are colocated together, workload changes are more frequent, which further increases the difficulty for resource partitioning. We shall show that all the existing resource partitioning solutions fail to deal with the large-scale resource partitioning problem (Section 2).

To address the aforementioned challenges, we propose Lavender, the first resource partitioning framework for large-scale job colocation. Lavender effectively tackles the problem's scale by dividing the colocated jobs into a number of disjoint groups and transforming the original problem into a series of smaller partitioning problems (*inter-group* partitioning and *intra-group* partitioning). It employs an accelerated gradient descent algorithm to solve these smaller partitioning problems individually, which is lightweight and highly efficient in terms of search efficiency. Considering the presence of numerous jobs sharing coarse-grained resources, Lavender optimizes job combinations to minimize competition for shared resources. Moreover, in order to ensure a consistent optimality, Lavender utilizes a refining process to adjust the current configuration to adapt to frequent workload changes. Experimental results demonstrate the effectiveness of Lavender, with significant improvements in system throughput ranging from 12% to 33% compared to state-of-the-art baselines.

The contributions of this paper are as follows.

- We have proposed, for the first time, the problem of large-scale resource partitioning, and identified the main challenges and gaps in existing solutions for this problem.
- We have designed Lavender, a framework specifically tailored for large-scale resource partitioning problems. Lavender incorporates several key techniques to address the challenges associated with large-scale resource partitioning, ensuring its efficiency, adaptivity, and optimality.
- We have comprehensively evaluated Lavender to validate its performance and analyze the reasons behind its advantages. The experimental results demonstrate that Lavender outperforms state-of-the-art baselines by a substantial margin.

The rest of the paper is organized as follows. Section 2 illustrates the motivation. The large-scale resource partitioning problem is formally formulated in Section 3. Section 4 presents the detailed design of Lavender. Section 5 presents the experimental platform and the configuration of baselines. The evaluations are presented in Section 6. Section 7 discusses the further expansion of the framework. Section 8 summarizes the related work. Conclusions and future work are summarized in Section 9.

2 MOTIVATION

This section illustrate the motivation of this work. We first summarize the particular challenges of large-scale resource partitioning problem. Then, we show that existing resource partitioning solutions fail to deal with the large-scale resource partitioning problem.

2.1 Challenges

First, compared to the previous resource partitioning problems, the large-scale resource partitioning problem has much larger solution space. Consider the problem of partitioning a shared resource R (e.g., CPU cores) among a set of colocated jobs. Assume resource R has K units. The total number of configurations of partitioning resource R among N jobs will be C_{K-1}^{N-1} [34]. If resource R has 20 units (e.g., 20 CPU cores) and there are 6 jobs, the total number of partitioning configurations is around 10^4. However, if resource R has 100 units (e.g., 100 CPU cores) and the number of jobs reaches to 20, the total number of partitioning configurations will be up to 10^{20}. It can be seen that the solution space of the resource partitioning problem grows exponentially as the problem size increases, and the increasing speed will be more fast when multiple resources are partitioned. Such a large search space presents a significant challenge for the resource partitioning problem.

Second, limited by the hardware-based isolation techniques, some shared resources can only be coarsely partitioned. For example, the LLC can only be partitioned at cache way granularity for Intel's CPU [26]. This may lead to two issues: (1) the total amount of a resource is limited, and even assigning one unit of the resource to each job may be still insufficient; (2) the actual resource requirement for each job may not necessarily be a multiple of the smallest unit, resulting in these jobs not being allocated the correct amount of resource. Therefore, the approach of fully isolating the shared resources among jobs may be impossible or not be optimal, and some shared resources have to be partially shared among the colocated jobs. This further increases the complexity of the resource partitioning problem, because how the resources are shared by the jobs also needs to be carefully determined (an example is shown in Figure 1).

Third, workload changes are more frequent for the large-scale partitioning problem. This is because: (1) there are more jobs colocated on the server, leading to a higher probability of workload change occurring; (2) some resources have to be partially shared by multiple jobs, thus workload change of one job would impact other jobs. Since the resource partitioning configuration should be adjusted with workload changes, more frequent workload changes require a more robust resource partitioning strategy, which further increases the difficulty.

2.2 Limitations of Existing Solutions

We shall show that all the existing resource partitioning solutions do not perform well as the problem scale become large. Among the existing resource partitioning solutions, we primarily focus on the online tuning-based methods as they have the advantage of being able to partition multiple resources simultaneously. Among the various tuning-based methods, PARTIES [7], CLITE [32], and OLPart [5] stand out as the most representative solutions.

PARTIES. PARTIES is the first tuning-based approach that considers partitioning multiple resources. It continuously monitors the performance of each job and dynamically adjusts the

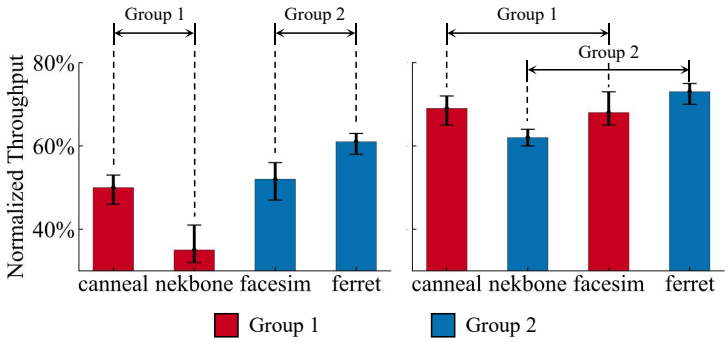

Fig. 1. There are 4 jobs running on a server, which are divided into two groups; the resources are partitioned into two equal parts, shared by the jobs within each group respectively; the two figures on the left and right respectively show the performance of each job under two different combinations; it can be seen that how the jobs are combined could significantly impact the performance. The y-axis represents the percentage of each job relative to its own performance in resource-rich scenarios.

resource allocations. PARTIES tries incrementally increasing/decreasing one resource (e.g., number of cores, number of cache ways, etc.) for one job at a time, and accesses the observed performance. This process operates until the optimization objective (e.g., system throughput) is maximized.

CLITE. CLITE is another tuning-based resource partitioning approach which adjusts multiple resources coordinately. It utilizes Bayesian Optimization to construct a cost-effective performance model by sampling a small number of points from a large search space. This performance model provides an approximate estimation of the quality of various resource partitioning configurations, enabling intelligent exploration of the search space to find near-optimal configurations.

OLPart. OLPart is a cutting-edge method for resource partitioning, which relies on the online learning techniques. This approach is developed by considering the fact that performance counters during runtime can roughly reflect the resource sensitivities of jobs. With this insight, OLPart utilizes contextual multi-armed bandit (CMAB) to incorporate performance counters for creating a partitioning solution, which can intelligently explore the search space.

Although PARTIES, CLITE and OLPart perform well on small-scale resource partitioning problems, we shall show that their performance deteriorates when the problem scale increases. As shown by the results in Figure 2 (the experimental settings are presented in Section 5), the performance of PARTIES, CLITE and OLPart drops dramatically when the number of colocated jobs exceeds 8, and gradually loses advantage over fully sharing (i.e., no partitioning) as the colocation size increases. For PARTIES, the searching efficiency is considerably constrained because it tunes solely on one resource and makes incremental adjustments for one job at a time. This drawback is amplified exponentially as the problem scale increases. For CLITE and OLPart, their performance highly relies on the accuracy of the performance models, while constructing accurate prediction models with limited sampling information become extremely challenging when the search space is large.

Summary: *The current resource partitioning solutions are not specifically designed to handle large-scale resource partitioning problems, resulting in reduced performance when confronted with large search space.*

3 PROBLEM DEFINITION

Consider a set of throughput-oriented jobs to be run on a cloud server. Our objective is to partition the server's shared resources effectively among the jobs, ensuring that the system's overall

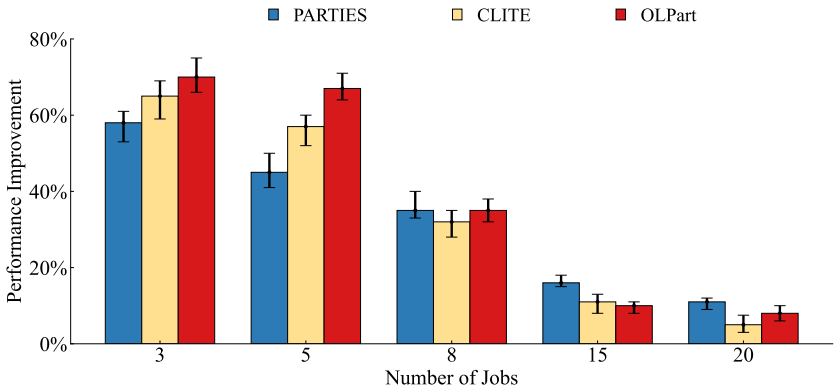

Fig. 2. The y-axis represents the performance improvement of each partitioning approach compared to fully sharing. Notably, PARTIES, CLITE, and OLPart exhibit a significant decrease once the number of colocated jobs exceeds 8.

throughput is maximized. The system throughput is defined as the total performance of all the jobs, where the performance of each job is measured by the time-averaged IPC (instructions per cycle). As the workload of the jobs fluctuates, the partitioning configuration should be continuously adjusted to maintain optimal performance consistently. Particularly, existing studies about resource partitioning usually assume a limited number of jobs that can be colocated (often fewer than 8). In this paper, we focus on a large-scale scenario which allows more jobs to be colocated.

The large-scale resource partitioning problem is formally defined as follows. Denote by $\mathcal{J}$ the given set of jobs, and N_{job} the number of jobs in $\mathcal{J}$. Let $\mathcal{R}$ denote the set of resources that can be partitioned. We divide the resources into two types: *fully-isolated* and *partially-shared*. The *fully-isolated* resources will be fully isolated among the colocated jobs, while the *partially-shared* resources will be partially shared among the colocated jobs. Let R_f and R_p denote the set of *fully-isolated* resources and *partially-shared* resources respectively.

For each job $j \in \mathcal{J}$, denote by $BASE\_IPC_j$ the average IPC of job j when it runs alone on the server. Consider a certain time point t. Let P_t denote the partitioning configuration at time t, referring to how both the *fully-isolated* and *partially-shared* resources are allocated among the coloated jobs. Let $IPC_{j,t}$ denote the IPC of job j at time t. The speedup of job j at time t is defined as $IPC_{j,t}/BASE\_IPC_j$. We define the system throughput at time t as the geometric mean of the speedups of all jobs, i.e.,

$$\left(\prod_{j=1}^{N_{job}} \frac{IPC_{j,t}}{BASE\_IPC_j}\right)^{\frac{1}{N_{job}}} \tag{1}$$

Our goal is to find the optimal partitioning configurations $\{P_t\}$ such that the system throughput is maximized at every time point t.

In the above problem definition, we ignore dynamic job arrivals/departures, and assume the jobs stay in the system for a relatively long time. This is a typical occurrence in modern data centers, where long-term services are commonly run in containers. Nevertheless, the suggested solution can be readily expanded to encompass scenarios involving dynamic job arrivals and departures. (Section 7).

4 DESIGN OF LAVENDER

In this paper, we propose Lavender, a resource partitioning framework for maximizing the system throughput of cloud servers with multiple jobs colocated. The key advantage of Lavender is that Lavender is able to handle much larger problem size compared to the existing resource partitioning solutions. The basic idea of Lavender is to divide the colocated jobs into a number of disjoint groups, and partition the resources in two steps: *inter-group* partitioning and *intra-group* partitioning. In the inter-group partitioning step, all the resources (including both the *fully-isolated* and the *partially-shared* resources) are partitioned among groups and the jobs within each group fully share the resources allocated to the group. In the intra-group partitioning step, the *fully-isolated* resources allocated to each group are further partitioned among the jobs in the group. Since both *inter-group* partitioning and *intra-group* partitioning have much smaller problem size, the original partitioning problem can be greatly simplified.

However, the design of Lavender is non-trivial, which faces several critical challenges: (1) how to determine the number of groups and the members of each group? (2) how to design efficient *inter-group* and *intra-group* partitioning solutions? (3) how to deal with dynamic workload changes to maintain consistently optimal partitioning configurations? We propose several techniques to address the above issues, and the details are presented in the rest of this section.

4.1 Jobs Grouping

4.1.1 How to determine the number of groups? The number of groups should be carefully determined, because it determines the total size of search space. Intuitively, a too large number of groups will lead to a large search space for the *inter-group* partitioning problem, while a too small number will lead to large search space for the *intra-group* partitioning problem (the number of jobs in each group is large). Ideally, if the number of groups is equal to the number of jobs in each group, the *inter-group* partitioning problem and the *intra-group* partitioning problem will have similar size of search space. In this case, the total size of search space will be minimized. Therefore, the optimal number of groups (denoted by N_{grp}) can be computed according to $N_{grp} = \sqrt{N_{job}}$ (rounding if N_{job} is not a perfect square).

4.1.2 How to determine the members of each group? In the *intra-group* partitioning step, we only partition the *fully-isolated* resources among the jobs within a group, while the *partially-shared* resources are fully shared by the jobs within the group. Therefore, the jobs in the same group will contend for the *partially-shared* resources. Resource contention would lead to performance degradation, while the significance of resource contention is highly dependent on the resource sensitivities of jobs. Intuitively, the jobs with similar resource sensitivities should not be assigned to the same group, because they are likely to demand for the same resource, which would lead to more serious resource contention. In order to minimize the resource contention, we first perform an offline profiling for each job to estimate its resource sensitivities, and then use a heuristic algorithm to assign the jobs to groups such that the resource contention is minimized.

For each job $j \in \mathcal{J}$ and resource $r \in \mathcal{R}_p$, the sensitivity of job j to resource r is estimated as follows. We run job j on the server for a short period, and measure the performance of job j with $H\%$ (a relatively high proportion) of resource j allocated and $L\%$ (a relatively low proportion) of resource j allocated respectively, and denote the performance of job j by T_H^j and T_L^j respectively. The sensitivity of job j to resource r is defined by

$$S_r^j = (T_H^j - T_L^j)/(H\% - L\%) \tag{2}$$

For two jobs j and j' in the same group, we define the resource contention degree between j and j' as

$$DC_{j,j'} = \sum_{r \in \mathcal{R}_p} e^{S_r^j + S_r^{j'}} \tag{3}$$

, with a higher degree indicating more significant contention. Let G_i denote the set of jobs assigned to the i-th group ($1 \le i \le N_{grp}$). The total resource contention degree for the i-th group is defined as the sum of resource contention degrees between each pair of jobs in the i-th group, i.e.,

$$DC_{G_i} = \sum_{j,j' \in G_i, j \neq j'} DC_{j,j'} \tag{4}$$

With the above definitions, our objective is to find the optimal $\{G_1, G_2, \ldots, G_{N_{grp}}\}$, such that the total resource contention degree of the system is minimized, i.e.,

$$\min \sum_{i=1}^{N_{grp}} DC_{G_i} \tag{5}$$

It is easy to see that the problem of finding the optimal $\{G_1, G_2, \ldots, G_{N_{grp}}\}$ is NP-hard. Iterating the entire search space to find the optimal solution is time consuming. Therefore, we propose a heuristic algorithm to approximate the optimal solution, which is presented in Algorithm 1.

Algorithm 1 Job Grouping Algorithm

Input the set of jobs $\mathcal{J}$, the sensitivity of each job $j \in \mathcal{J}$ to each resource $r \in \mathcal{R}_p$, the number of groups N_{grp}
Output the optimal $\{G_1, G_2, \ldots, G_{N_{grp}}\}$

1: $\mathcal{J}_u \leftarrow$ the set of unsigned jobs, initialized by $\mathcal{J}$
2: **for** each i from 1 to N_{grp} **do**
3: $j^* \leftarrow \arg_{j \in \mathcal{J}_u} \min DC_{G_i \cup \{j\}}$ //find the optimal job for group i
4: $G_i \leftarrow G_i \cup \{j^*\}$ //add job j^* to G_i
5: $\mathcal{J}_u \leftarrow \mathcal{J}_u - \{j^*\}$ //remove job j^* from $\mathcal{J}_u$
6: **end for**
7: **repeat**
8: $j_1, j_2 \leftarrow$ two jobs randomly selected from two different groups G_{i_1}, G_{i_2}
9: **if** $DC_{G_{i_1}} + DC_{G_{i_2}} > DC_{G_{i_1} - \{j_1\} \cup \{j_2\}} + DC_{G_{i_2} - \{j_2\} \cup \{j_1\}}$ **then**
10: $G_{i_1} \leftarrow G_{i_1} - \{j_1\} + \cup\{j_2\}$
11: $G_{i_2} \leftarrow G_{i_2} - \{j_2\} + \cup\{j_1\}$
12: **end if**
13: **until** the termination condition is reached

The algorithm first uses a greedy algorithm to generate an initial assignment of jobs (lines 1 to 6). Specifically, it visits group 1 to group N_{grp} in a cyclic manner and assigns one job to the group being visited each time, until all the jobs are assigned. When visiting group i, among all the unassigned jobs, the algorithm picks the job that incurs the minimum resource contention degree with the jobs already assigned to group i (line 3), and assigns the picked job to group i (line 4). After that, the algorithm conducts a local search process to iteratively tune the assignment to further improve the assignment. In each iteration, the algorithm randomly selects two jobs from two different groups, and tries to exchange the two jobs (line 8). This operation is accepted if the total contention degree of the two groups are reduced, and rejected otherwise (line 9 to 12). The

local search process terminates if an acceptable solution is found or a predefined time limit is reached.

4.2 Resource Partitioning

After the groups are formulated, we need to determine how to partition the resources among groups (i.e., *inter-group* partitioning) and how to partition the resources allocated to each group among the jobs within the group (i.e., *intra-group* partitioning). It is worth noting that *inter-group* partitioning and *intra-group* partitioning are mutually influential, which should be considered coordinately. However, that would lead to excessive complexity. Therefore, we consider all the jobs in the same group as a whole during *intra-group* partitioning, and let the jobs in the same group fully share the resources allocated to the group. In this way, *inter-group* partitioning and *intra-group* partitioning can be done separately in a sequential manner, which greatly simplifies the problem.

4.2.1 Inter-group Partitioning. *Inter-group* partitioning aims to properly partition the shared resources (including both *fully-isolated* and *partially-shared* resources) among groups, so that the system throughput is maximized. We propose a gradient-descent based resource partitioning approach, which is highly adaptive to dynamic workload changes and can quickly find a good solution in search space. The basic idea is to quickly find an initial partitioning configuration, and then dynamically tunes the configuration based on gradient-descent, until a sufficiently good configuration is found. To improve the exploring efficiency, we estimate the real-time resource sensitivities for each job based on the knowledge collected online, and use the resource sensitivities to guide the exploration of search space, which can significantly accelerate the process of exploration.

Finding an initial partitioning configuration. The quality of the initial partitioning configuration is very important. If it is very far from the optimal configuration, the tuning process will be slow. In order to find a good initial configuration, we adopt Latin Hypercube Sampling approach to uniformly sample the solution space for a few times (controlled by a parameter N_{init}), and choose the sampled configuration with the best performance.

Gradient-descent based tuning. Based on the initial configuration, the gradient-descent based tuning process continues to dynamically adjust the configuration to further improve the solution. The details are presented in algorithm 2. The tuning process iterates continuously, until a sufficient good configuration is found. In each iteration, two groups are selected, one group is the *supplier* and another group is the *demander* (lines 2-3). We pick the group with the highest performance as the *supplier*, and the group with the lowest performance as the *demander*. We try to realloate some resources from the *supplier* to the *demander*. Specifically, we randomly choose a resource r' (line 6) and try to reallocate one unit of r' from the *supplier* to the *demander* if there is remaining amount of resource r' for the *supplier* (line 8); we observe the system throughput after this operation; if the system throughput is improved, we accept this operation and continue to reallocate more units of this resource; otherwise, if the system throughput is not improved, we reject this operation and change to another resource (lines 9-11). If all the resources have been tried, we end this iteration and continue to next iteration.

Resource sensitivity aware pruning. The gradient-descent tuning process is not aware of resource sensitivities of jobs, which may generate improper resource allocations and cause low search efficiency. To address this issue, we propose a resource sensitivity aware pruning strategy to further improve the search efficiency. To this end, we maintain a table to record the upper bound and lower bound of each resource for each group. Consider a group g and a resource r. Let $UP_{g,r}$ and $LW_{g,r}$ denote the upper bound and lower bound of resource r for group g. $UP_{g,r}$ indicates the maximum allocation of resource r to group g, and allocating more resource r to group g can not

Algorithm 2 Gradient-descent based Tuning

Input the initial partitioning configuration
Output the tuned partitioning configuration

repeat
  $G_s \leftarrow$ the supplier (the group with the highest performance)
  $G_d \leftarrow$ the demander (the group with the lowest performance)
  $\mathcal{R}' \leftarrow \mathcal{R}_d$ (the set of resources that have not been tried, initialized by $\mathcal{R}_d$)
  while $\mathcal{R}' \neq \emptyset$ **do**
    $r' \leftarrow$ a resource randomly selected in $\mathcal{R}'$
    while there is remaining abount of r' for G_s **do**
      shift one unit of resource r' from G_s to G_d
      if system throughput is decreased **then**
        revoke the above shift operation and break
      end if
    end while
    $\mathcal{R}' \leftarrow \mathcal{R}' - \{r'\}$
  end while
until a sufficiently good configuration is found

help to improve performance. $LW_{g,r}$ indicates the minimum allocation of resource r to group g, and allocating less resource r to group g would incur significant performance degradation.

The upper bounds and lower bounds are estimated as follows. Initially, the upper bounds of resource r for all groups are set to the capacity of resource r, while the lower bounds of resource r for all groups are set to 0. Then, the upper bounds and lower bounds are updated continuously in the gradient-descent based tuning process. Suppose a group g has m units of resource r at a certain moment. If group g's performance degradation exceeds a predefined proportion $X\%$ after reducing one unit of resource r, we update $LW_{g,r}$ to m. If group g's performance improvement is less than a predefined proportion $Y\%$ after adding one unit of resource r, we update $UP_{g,r}$ to m. With the upper bounds and lower bounds, the tuning process will prune the operations that attempt to increase the allocation of a particular resource to a group if that resource for that group has already exceeded the upper bound, and the operations that attempts to decrease the allocation of a particular resource from a group if that resource for that group has been below the lower bound. Therefore, the tuning process can be greatly accelerated.

4.2.2 Intra-group Partitioning. After *inter-group* partitioning, *intra-group* partitioning aims to further partition the *full-isolated* resources among the jobs within each group, to further improve the system throughput. It can be seen that *intra-group* partitioning is essentially the same as *inter-group* partitioning, with the only difference being that the former partitions resources among jobs while the latter partitions resources among groups. We will use similar approach to solve the *intra-group* partitioning problem.

Specifically, for each group, we first uniformly allocate the *full-isolated* resources among the jobs within the group. Then, we adopt a similar gradient-descent based tuning process as has been used in *inter-group* partitioning to dynamically adjust the partitioning configuration, where we replace "group" with "job". Note that *intra-group* partitioning is conducted by each group independently.

4.3 Adaption to Workload Changes

Fluctuations in job workload are common in data centers. Intuitively, the resource partitioning configuration should be adjusted with workload changes, since the optimal configuration would be different for different workloads. To address this issue, we continuously monitors the performance of jobs, and take corresponding refining operations according to the extent of performance degradation.

Specifically, if the throughput of a group (refers to the geometric mean of the speedups of the jobs in the group) has decreased by a relatively small margin (say 5% to 20%, which can determined according to actual needs), we believe that the impact of workload change is not significant and only the *intra-group* partitioning configuration of this group needs to be refined. In this case, we will invoke the gradient-descent tuning process as has been used in *intra-group* partitioning, but take the current configuration as starting point to refine the *intra-group* partitioning configuration for this group (Algorithm 2). Otherwise, if the average throughput of all groups has decreased by a relatively large margin (say more than 20%, which can be defined according to actual needs), we believe that the impact of workload change is significant and both of *inter-group* and *intra-group* partitioning configurations need to be refined. In this case, we will first invoke the gradient-descent tuning process as has been used in *inter-group* partitioning, but take the current *inter-group* partitioning configuration as starting point, to refine the *inter-group* partitioning configuration; and then refine the *intra-group* partitioning configuration for all groups using the methods mentioned above.

4.4 Putting It Together

The overall design of Lavender is shown in Figure 3. Given a set of jobs to be colocated, Lavender first analyzes the resource sensitivities of the jobs through a short offline profiling and divides the jobs into groups ❶. Then, Lavender starts running the jobs and partitions the resources among the colocated jobs through *inter-group* partitioning ❷ and *intra-group* partitioning ❸ respectively. After that, Lavender periodically invokes the corresponding refining process based on performance changes to adapt to dynamic workload changes ❹.

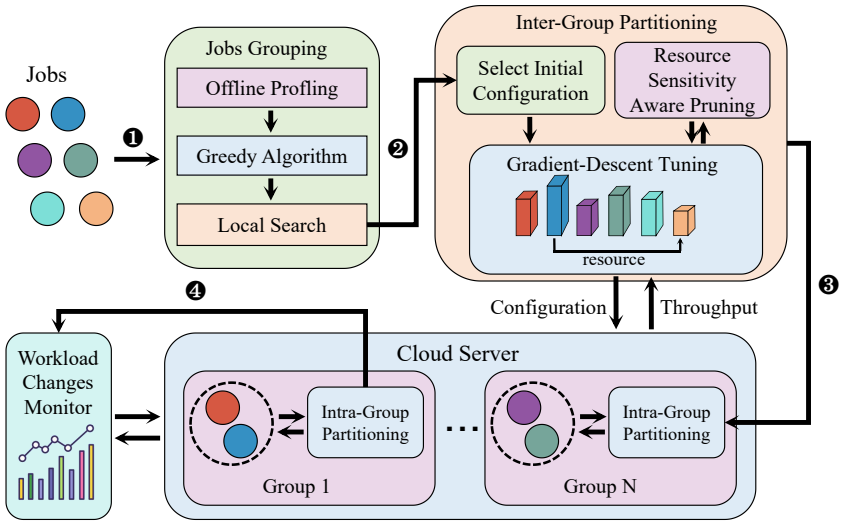

Fig. 3. Overview of Lavender, composed by the jobs grouping module, inter-group partioning module, intra-group partitioning module, and workload changes monitor module.

5 METHODOLOGY

We conduct extensive experiments to evaluate the proposed framework and compare it with the state-of-the-art baselines. This section presents the details.

5.1 Experimental Settings

Hardware Platform. We implement and evaluate the proposed framework on a server with the configuration shown in Table 1. In this paper, we particularly consider the partitioning of three most important shared resources: CPU cores, LLC and memory bandwidth, which are partitioned using taskset [22], Intel's CAT [26] and MBA [39] respectively. Among the three resources, CPU cores are *fully-isolated*, LLC and memory bandwidth are *partially-shared* among the colocated jobs. The CPU is partitioned in core granularity, and at least one CPU core will be allocated to each job. Intel's CAT arranges the LLC as a series of cache ways with sequential indexing. It mandates that the cache ways assigned to a particular job must be contiguous (e.g., {3,4,5} or {1,2}). Thus, we use (llc_l, llc_r) to denote the allocation of LLC, where llc_l and llc_r represent the left and right boundaries of the cache ways. Intel's MBA can specify the maximum proportion from the entire bandwidth that a job can use, where the maximum proportion can be one of ten levels: $\{10\%, 20\%, \ldots, 100\%\}$. The IPC of jobs is collected using the perf tool [1].

Table 1. Experimental testbed configuration.

Component	Specification
Processor	Intel(R) Xeon(R) Gold 6330, 2.00GHZ, 120 cores (60 physical cores)
Private L1 & L2 Cache Size	64 KB and 1024 KB per core
Shared L3 Cache Size	80MB (12 cache ways)
Memory Capacity	256GB
Operating System	Ubuntu 20.04.4 with Linux 5.4.0

Workloads. Table 2 presents the jobs utilized in our experiments. We use a series of throughput-sensitive jobs contained in PARSEC 3.0 [42], ECP [33], and SPLASH-3 [35] benchmarks to carry out our experiments.

Parameters setting. We have tried a lot of settings for Lavender and find the following settings work well for our testbed: in order to measure the sensitivity of a job j to resource r, we allocate 40% (i.e., the parameter $H\%$) and 10% (i.e., the parameter $L\%$) of resource r to job j respectively, and run job j for a period of 2.5 seconds for each allocation to measure its performance; for the job grouping algorithm (Algorithm 1), the local search process terminates if the quality of the solution does not improve for 30 consecutive iterations; for finding the initial partitioning configuration, we uniformly sample the solution space for 30 times (i.e., parameter N_{init}); the gradient-descent based tuning process terminates if it fails to find a better configuration with 15 seconds; for resource sensitivity aware pruning, the parameter X% (i.e., the threshold of performance degradation that triggers the update of the resource sensitivity after the *supplier* reduces resources) and the parameter Y% (i.e., the threshold of performance improvement that triggers the update of resource sensitivity after the *demander* increases resources) are set to 20% and 15% respectively; the *intra-group* partitioning configuration is refined if the throughput decline of a group is between 5% and 20%, while both *intra-group* partitioning and *inter-group* partitioning configurations need to be refined if the average throughput decline of all groups is larger than 20%.

Table 2. The benchmark suites used in the evaluations.

Benchmark Suites	Jobs
PARSEC 3.0 [42]	blackscholes, canneal, dedup, facesim, fluidanimate, freqmine, ferret, raytrace, streamcluster, vips, swaptions, x264
ECP [33]	AMG, CoMD, Laghos, miniAMR, miniFE, miniTri, NEKbone, SW4lite, SWFFT, XSBench
SPLASH-3 [35]	Cholesky, Barnes, FFT, FMM, LU_CB, Ocean_CP, Radiosity, Radix

5.2 Baselines

We compare Orchid with a wide variety of state-of-the-art baseline solutions for resource partitioning, as listed below:

Random. In this approach, the resource partitioning configuration is randomly selected among all possbile partitioning configurations.

PARTIES. PARTIES [7] is the first tuning-based approach that considers partitioning multiple resources. It continuously monitors the performance of each job and dynamically adjusts the resource allocations. PARTIES tries incrementally increasing/decreasing one resource for one job at a time.

CLITE. CLITE [32] leverages Bayesian Optimization to build approximate performance model online and uses the performance model to guide an intelligent search for the near optimal resource partitioning configuration.

OLPart. OLPart [5] is a cutting-edge method for resource partitioning, which relies on the online learning techniques. OLPart utilizes contextual multi-armed bandit (CMAB) to incorporate performance counters for creating a partitioning solution, which can intelligently explore the search space.

Brute-Force Search (ORACLE). ORACLE stands for the maximum throughput that can be achieved theoretically, which is obtained through a brute-force sampling. Note that it is not feasible to traverse the entire solution space to find the best configuration. Instead, we run PARTIES for 24 hours (which is sufficiently long to ensure a good configuration) and use the obtained configuration to approximate ORACLE.

Note that PARTIES, CLITE and OLPart all assume that the number of jobs is smaller than the total units of each resource, and all the resources are *fully-isolated* in their context. However, this assumption does not hold in our scenario since there are more jobs colocated. Therefore, directly applying PARTIES, CLITE and OLPart to our scenario is not feasible, since some resources (e.g., LLC and memory bandwidth) have to be partially shared. To address this issue, in the implementation of PARTIES, CLITE and OLPart, we divide the jobs into groups (the number of groups is equal to that of Lavender, each group has equal number of jobs where the jobs are randomly selected) and partition the *partially-shared* resources among groups. The *partially-shared* resources allocated to each group are shared by the jobs within the group. Moreover, PARTIES, CLITE and OLPart are originally designed to optimize two objectives, maximizing the throughput of throughput oriented jobs while guaranteeing the QoS of latency critical jobs. In our implementation, since there are only throughput oriented jobs, we have changed the objective to optimize only throughput according to the requirements.

In order to make a fair comparison, each approach makes partitioning configuration in every 0.5 seconds (except for ORACLE), and the resources are equally partitioned among the jobs in the initial partitioning configuration. By default, each colocation runs for a period of 90 seconds, and

we compute the average performance of the last 5 seconds (when all the baselines get a stable performance) for each approach; we partition 60 CPU cores, 10 LLC cache ways and the entire memory bandwidth among the colocated jobs.

6 EVALUATIONS

6.1 Overall Performance

In this experiment, we evaluate the overall performance of Lavender compared to the baselines. We randomly generate 10 colocations using the jobs in the benchmark suite, with each colocation has 20 jobs. We measure the throughput (as a percentage relative to ORACLE) of each colocation for each approach, and compute the average throughput of all the colocations for each approach. Figure 4 shows the results. We have several observations.

First, Random exhibits the poorest performance which is only half that of Oracle. It indicates that the partitioning configuration could greatly impact system performance. Therefore, careful and accurate resource allocation is essential to maximize performance.

Second, the performances of CLITE and OLPart are similar, which are both slightly better than that of Random (with 6% and 7% advantages respectively). However, CLITE and OLPart are still far from optimal, with gaps of 35% and 34% respectively compared to Oracle. This is because they both rely on the performance models to guide the exploration of search space, while building accurate performance models with limited sampling is very difficult when the search space is huge.

Third, PARTIES also deviates significantly from optimal, with a gap of 31% compared to ORACLE. This is because PARTIES tunes one resource and one job at a time, its search efficiency is very low, especially when dealing with large search spaces. However, the performance of PARTIES is slightly better than that of CLITE and OLPart, with advantages of 4% and 3% respectively. This result contradicts the conclusion drawn from small-scale resource partitioning problem [5] (i.e., PARTIES performs worse than CLITE and OLPart when the problem scale is small). It implies that the increase in problem size has a greater impact on CLITE and OLPart than on PARTIES. This is because despite PARTIES has low search efficiency, as it is based on gradient descent, the quality of the solution has been consistently improving; in contrast, CLITE and OLPart rely on performance models, which may generate poor-quality solutions during the exploration process if the model is inaccurate.

Last, Lavender outperforms the baselines significantly, with advantages of 27%, 26% and 23% over CLITE, OLPart and PARTIES respectively. The performance gap of Lavender is only 8% compared to ORACLE, which demonstrates the efficiency of Lavender. We attribute this to that: (1) by decomposing the original problem into two stages (*inter-group* partitioning and *intra-group* partitioning), Lavender reduces the complexity of the problem and significantly narrows down the solution space; (2) by designing a pruning-based gradient descent search algorithm, the search efficiency has been greatly improved.

Figure 5 provides a detailed performance breakdown of all approaches on each colocation. It can be seen that there are variations in the performances across different colocations. This is because the resource sensitivities of jobs are different for different colocations, while the impact of resource partitioning is more significant for the colocations with more intense resource contention. Among all the colocations, Lavender alway achieves the best performance, with an advantage ranging from 12% to 33% over the baselines. The results demonstrate that the effectiveness of Lavender is consistent across all types of colocations.

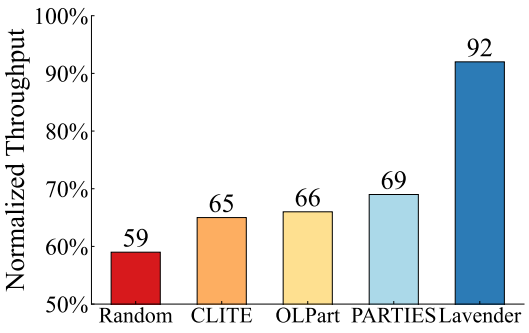

Fig. 4. The average performance of each approach over all colocations (normalized to Oracle).

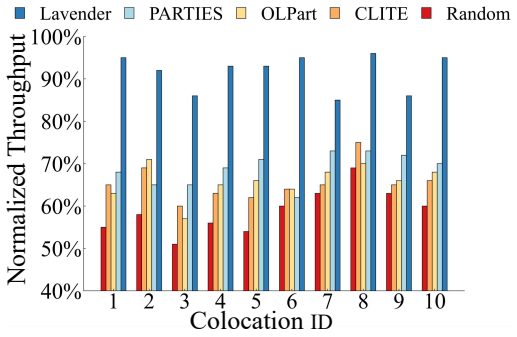

Fig. 5. Performance breakdown of all approaches on each colocation.

6.2 Scalability

In this experiment, we evaluate the scalability of Lavender to different problem scales. We create different problem scales by changing the colocation size or the amount of resources to be partitioned. The left of Figure 6 shows the average performance of each approach when we fix colocation size at 20 (each colocation has 20 jobs) and partition 40, 60 and 80 CPU cores (hyper-threading is enabled to generate 80 cores) respectively. The right of Figure 6 shows the average performance of each approach when we fix CPU cores at 60 and vary colocation sizes in 10, 20 and 30. All the colocations used in each experiment are randomly generated.

We have two observations from the results. First, as the problem scale increases (either more CPU cores or more jobs), the performance of all approaches shows a downward trend. This is because resource partitioning becomes more challenging as the problem scale increases. Second, the decrease in performance of Lavender is much smaller compared to other baselines. For example, the performance decrease of Lavender is 5% for 80 CPU cores compared to 40 CPU cores, which is much smaller than that of PARTIES (18%, from 81% to 63%). This indicates that Lavender has much better scalability compared to other baselines across different problem scales.

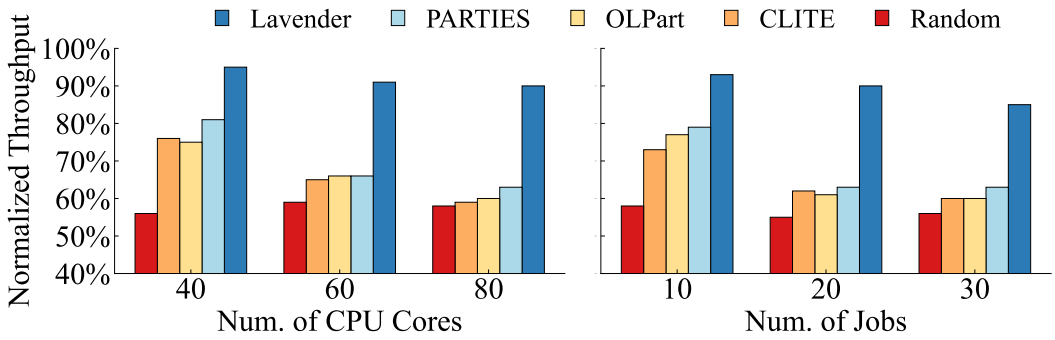

Fig. 6. The left part shows the performance of each approach when partitioning different numbers of CPU cores; the right part shows the performance of each algorithm for different colocation sizes.

6.3 Efficiency

In this experiment, we evaluate the search efficiency of Lavender compared to the baselines. Figure 7 shows the performance trend of each approach during the search process for a specific colocation of 20 jobs (randomly selected). We have several observations. Firstly, the performance of Random exhibits violent fluctuations throughout the entire process, and cannot converge to a stable value. This is expected because Random selects a configuration randomly each time, which cannot guarantee the quality of the configuration. Second, the performance of CLITE and OLPart also exhibit significant fluctuations, but they show a converging trend towards the end, although the convergence speed is relatively slow. On average, their performance is only slightly better than Random. This is because both CLITE and OLPart rely on performance models, and it is challenging to establish an accurate performance model with limited sampling attempts in such a large search space. Third, compared to CLITE and OLPart, PARTIES exhibits much smaller performance fluctuations, especially in the later stages, where it can converge to a relatively stable value. This is because PARTIES makes small adjustments each time, resulting in relatively continuous performance changes. However, the convergence speed of PARTIES is still relatively slow and only slightly better than CLITE and OLPart. This again confirms that the search efficiency of PARTIES is rather low for large problem scale. Last, the performance convergence speed of Lavender is much higher than that of other baselines. For example, Lavender reaches the performance level of PARTIES around halfway through the time (45 seconds), and its final performance is about 22% higher than PARTIES. This result indicates that Lavender has a very high search efficiency.

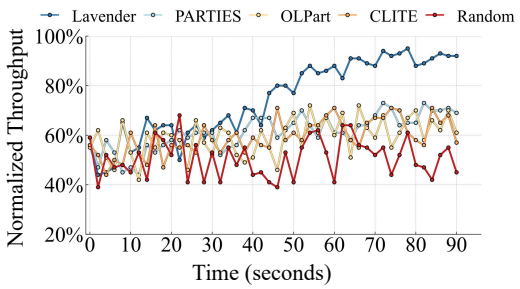

Fig. 7. Performance change of each approach during the search process.

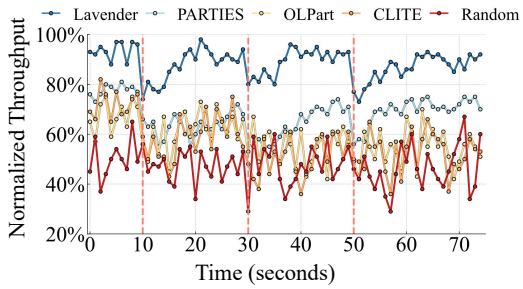

Fig. 8. Performance change of each approach in the face of pronounced workload changes.

6.4 Adaption to Dynamic Changes

In this experiment, we evaluate the adaptability of Lavender to workload changes. It is worth noting that the workload of jobs is constantly changing. Therefore, in the earlier experimental results, the outstanding performance of Lavender has already demonstrated its excellent adaptability to workload changes. In this experiment, we particularly evaluate the performance of Lavender when the workload changes are more pronounced. The pronounced workload changes are created manually by modifying the parameters of jobs during the execution. Figure 8 shows the performance variations of each approach when they encounter pronounced workload changes (marked with dashed lines). We have several observations.

First, all the approaches (except for Random) suffer significant performance degradation when they encounter pronounced workload changes. This is because the current searched configuration is only suitable for the previous workload characteristics and becomes outdated when the workload changes. Second, the recovery process after performance degradation is relatively slow

for CLITE and OLPart. This is because they both rely on performance models to guide the search, but the previous performance models become inaccurate after the workload changes. Third, the performance of PARTIES recovers much faster than CLITE and OLPart. This is because PARTIES makes small adjustments based on the current configuration. Despite the workload changes, the current configuration remains relatively good (compared to any randomly generated configurations). Therefore, making adjustments based on it can quickly lead to a good configuration. Last, the performance of Lavender recovers very fast after the workload changes, which only takes about 10 seconds to restore to the level before workload change. This is because the pronounced workload changes have triggered the *intra-group* partitioning configuration refining process (the performance degradation is between 5% and 20%); the refining process is based on gradient descent with pruning, which can quickly find a good configuration based on the current state. The results validate the effectiveness of the *intra-group* partitioning configuration refining process, and also indicate that Lavender has strong adaptability to dynamic workload changes.

6.5 Effectiveness of Key Technologies

In this experiment, we evaluate the effectiveness of each key technology in the design of Lavender. To this end, we implement three modified versions of Lavender: Lavender without *inter-group* partitioning (Lavender-No-InterGp), Lavender without *intra-group* partitioning (Lavender-No-IntraGp) and Lavender without pruning (Lavender-No-Prune). For Lavender-No-InterGp, all resources are evenly distributed among groups. For Lavender-No-IntraGp, the resources allocated to each group are fully shared by the jobs within the group. For Lavender-No-Prune, there is no pruning in the gradient-descent based tuning. Figure 9 shows the performance of each modified version compared to the original Lavender over 10 randomly generated colocations of 20 jobs.

It can be seen that all the three modified versions suffer from a performance degradation compared to the original Lavender, with a ratio of 10.2%, 16.1% and 9.3% respectively. The reason for Lavender-No-InterGp is that the resources are not properly allocated among groups, which would result in increased resource competition among jobs within the groups that have not been allocated sufficient resources. The reason for Lavender-No-IntraGp is that the resources are not partitioned within each group, which would lead to performance degradation due to resource contention among the jobs in the same group. The reason for Lavender-No-Prune is that unnecessary configurations were not pruned, leading to a decrease in search efficiency.

6.6 Impact of Parameter Settings

In this experiment, we study the impact of key parameters in Lavender, including N_{init} (the number of sampling times for finding the initial partitioning configuration before the tuning process), $X\%$ (the threshold of performance degradation that triggers the update of the resource sensitivity after the *supplier* reduces resources in the tuning process) and $Y\%$ (the threshold of performance improvement that triggers the update of resource sensitivity after the *demander* increases resources in the tuning process). We randomly generate 50 colocations of 20 jobs, and run Lavender for the colocations with different parameter settings.

Figure 10 shows the performance change of Lavender as the parameter N_{init} varies from 10 to 40. It can be seen that the performance of Lavender increases as N_{init} grows, but becomes stable after N_{init} is larger than 30. This indicates that if the number of sampling times is less than 30, it is difficult to ensure that a sufficiently good configuration is included in the samples obtained. It also indicates that as long as the initial configuration reaches a certain level of quality, the performance of Lavender will be good, and a better initial configuration does not significantly improve the performance of Lavender. Therefore, we set N_{init} at 30 in the experiments.

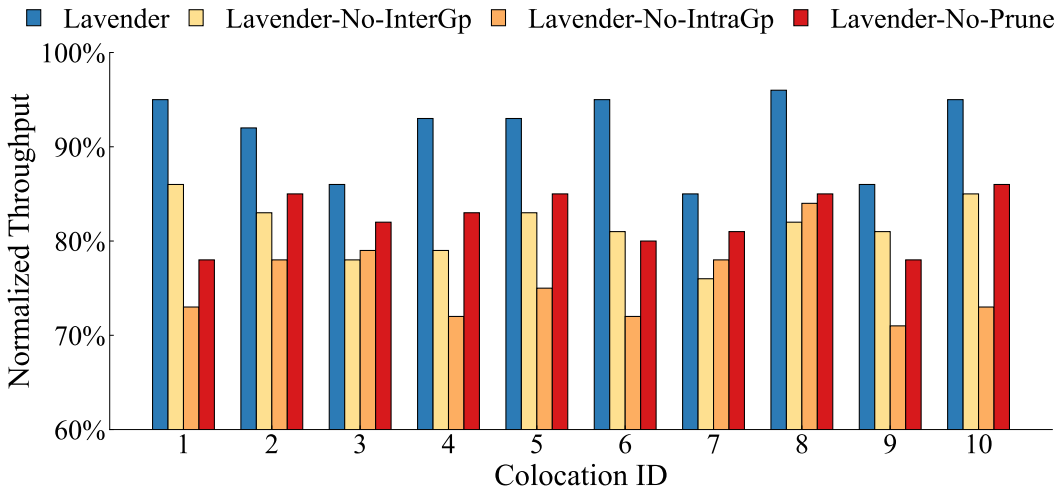

Fig. 9. The performance of each modified version compared to the original Lavender.

Figure 11 shows the performance change of Lavender as the parameter $X\%$ varies from 5% to 40%. It can be seen that Lavender achieves the best performance when $X\%$ is equal to 20%, and worse performance for both small and large $X\%$. This implies that the lower bounds of resource allocations for jobs can be precisely estimated when $X\%$ is set at 20%. Too large or too small values of $X\%$ could lead to inaccurate estimations of the lower bounds, resulting in incorrect resource allocations to jobs. Therefore, we set $X\%$ at 20% in the experiments.

Figure 12 shows the performance change of Lavender as the parameter $Y\%$ varies from 5% to 40%. It can be seen that Lavender achieves the best performance when $Y\%$ is equal to 15%, and worse performance for both small and large $Y\%$. This indicates that the upper bounds of resource allocations for jobs can be precisely estimated when $Y\%$ is set at 15%. Too large or too small values of $Y\%$ could lead to inaccurate estimations of the upper bounds, leading to incorrect resource allocations to jobs. Therefore, we set $Y\%$ at 15% in the experiments.

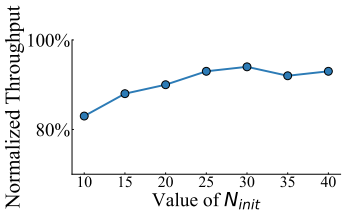

Fig. 10. Impact of parameter N_{init}.

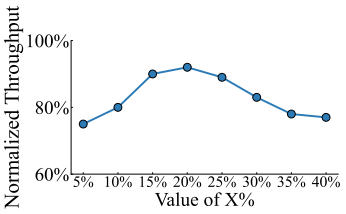

Fig. 11. Impact of parameter X%.

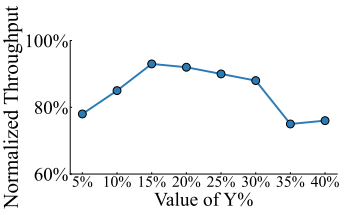

Fig. 12. Impact of parameter Y%.

6.7 Overhead

Lavender can be divided into offline stage and online stage. In the offline stage, we profile the sensitivity of jobs and divide the jobs into groups. For each job, the offline profiling takes around 15 seconds. Since profiling each job only occupies a portion of server resources, multiple jobs can be profiled simultaneously. Note that Job grouping (i.e., Algorithm 1) can be completed within 1 second. Such a time cost is acceptable because the offline stage is only performed once, and it is

Table 3. Time cost for each algorithm to complete a decision

Algorithm	10 jobs	20 jobs	30 jobs
CLITE	1200ms	1500ms	2300ms
OLPart	180ms	190ms	220ms
PARTIES	0.08ms	0.11ms	0.15ms
Lavender	0.18ms	0.20ms	0.23ms

done before the task execution. In the online stage, Lavender periodically tunes the partitioning configuration. For each tuning, it only requires a few simple computational operations to make decision, incurring very low time complexity. Table 3 summarizes the time cost for generating one partitioning configuration of each approach. It can be seen that Lavender and PARTIES are both computationally efficient (because they are both gradient-descent based), which takes less than 1ms to generate a partitioning decision. In contrast, CLITE and OLPart takes much longer time to make a partitioning decision, because they need to update the performance model each time.

7 DISCUSSION

The current Lavender ignores dynamic job arrivals and departures, and assume the jobs stay in the system for a relatively long time. However, it can easily be extended to support dynamic job arrivals and departures. For example, when a new job arrives, it can be simply assigned to the group that causes the least resource contention after assignment (the resource contention can be estimated according to formula (4)); after the assignment, we can simply determine an initial resource allocation to the new job (e.g., transfer some resources from the resource-rich jobs), and then invoke the *intra-group* refining process to improve the partitioning configuration; if there are too many jobs within a group due to frequent job arrivals, the group can be divided into two groups; when a job departs, we can reallocate the resources possessed by this job to the job with the worst performance, and then invoke the *intra-group* refining process to improve the partitioning configuration; if there are too few jobs within certain groups due to frequent departures, such groups can be merged.

8 RELATED WORK

The literature extensively discusses the study of workload consolidation to enhance resource utilization. Previous research primarily focused on conservative solutions, neglecting resource partitioning [2, 9, 12, 17, 41, 44]. Traditionally, the analysis of resource sensitivities for jobs is conducted through offline profiling. Based on this analysis, only jobs that do not compete for the same resources are colocated to ensure Quality of Service (QoS). However, these solutions have limitations as they rely on prior knowledge of jobs and are inefficient due to the limited types and number of jobs that can be colocated.

In recent studies, more aggressive approaches to workload consolidation have been adopted [10, 16, 19, 23, 25, 31, 36, 43]. These approaches involve partitioning the available resources among the colocated jobs. Existing resource partitioning solutions can be divided into two categories. The first category is model-based, where a performance model is utilized to guide the exploration of the search space [13, 20, 21, 28, 31, 38, 40]. Nonetheless, these approaches cannot be effectively applied to partition multiple resources. This is because constructing a performance model becomes impractical when dealing with complex contention behaviors involving multiple resources. The second category is online tuning-based, which dynamically adjusts the partitioning configuration in

real-time based on feedback from the actual system. Earlier works in this category adopt gradient-based tuning method [7, 8, 11, 23, 24, 27, 45], which has low search efficiency when there are multiple dimensions of resources. Recent works in this category leverage AI techniques to enable more intelligent exploration of the search space [5, 6, 32, 34]. These methods generally require constructing an approximate performance model through online sampling and utilize this model to guide the search process. However, these methods perform poorly when the problem scale is large, since it is difficult to create an accurate performance model.

There are also numerous studies that investigate specific scenarios of resource partitioning. Scavenger [18], PARTIES [7], CLITE [32] and OLPart [5] consider the scenario where latency critical jobs and throughput oriented jobs are colocated, and aim to maximize the performance of throughput oriented jobs with the QoS guarantee for latency critical jobs. SATORI [34] and Orchid [4] strives to improve both fairness and throughput of the colocated jobs. Shenango [29], Caladan [14] and Racksched [46] aims to achieve resource partitioning at a microsecond timescale, enabling rapid response to sudden phase changes that occur within very short time intervals. These approaches complement Lavender and can be seamlessly integrated into Lavender itself.

9 CONCLUSION

In this paper, we propose Lavender, a resource partitioning framework for maximizing the system throughput of cloud servers with a large number of jobs colocated. Lavender effectively addresses the scalability of the problem by dividing colocated jobs into multiple distinct groups. This approach transforms the original problem into a series of smaller partitioning problems, namely inter-group partitioning and intra-group partitioning. To solve these smaller partitioning problems, Lavender utilizes an accelerated gradient descent algorithm. This algorithm is lightweight and highly efficient in terms of search efficiency. We also have proposed several technologies to ensure the efficiency, adaption and optimality of Lavender. Experimental results have validated the effectiveness of Lavender.

There are many directions that can be considered for further work. Firstly, modern servers often consist of multiple CPUs and adopt NUMA architecture, which incurs additional overhead when accessing memory across nodes. However, this paper only considers single CPU structures and does not account for the impact of NUMA architecture and cross-node memory access. Second, the performance of jobs is not only dependent on resource allocation but also on their own parameter settings. Moreover, there is a mutual influence between resource allocation and parameter settings, and both need to be jointly considered to achieve the best performance.

REFERENCES

[1] 2020. perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org/index.php/.
[2] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. 2015. Multi-objective job placement in clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[3] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for {Cloud-Scale} Computing. In *OSDI '14*. 285–300.
[4] Ruobing Chen, Wangqi Peng, Yusen Li, Xiaoguang Liu, and Gang Wang. 2023. Orchid: An Online Learning Based Resource Partitioning Framework for Job Colocation With Multiple Objectives. *IEEE Trans. Comput.* 72, 12 (2023), 3443 – 3457.
[5] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. 2023. OLPart: Online Learning based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers. In *EuroSys '23*.
[6] Ruobing Chen, Jinping Wu, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. 2020. DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers. In *ACM HPDC '20*. 175–188.

[7] Shuang Chen, Christina Delimitrou, and F. José Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ASPLOS '19*. 107–120.

[8] Cristina Chiva, Roger Olivella, Eva Borras, Guadalupe Espadas, Olga Pastor, Amanda Sole, and Eduard Sabido. 2018. QCloud: A cloud-based quality control system for mass spectrometry-based proteomics laboratories. *PloS one* 13, 1 (2018), e0189209.

[9] Christina Delimitrou and Christos Kozyrakis. 2013. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 23–33.

[10] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems* 31, 4 (2013), 1–34.

[11] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[12] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 97–110.

[13] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. 2018. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 104–117.

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *OSDI '20*. 281–297.

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS '19*. 19–33.

[16] Derek R Hower, Harold W Cain, and Carl A Waldspurger. 2017. Pabst: Proportionally allocated bandwidth at the source and target. In *HPCA '17*. IEEE, 505–516.

[17] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 261–276.

[18] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *SoCC '19*. 272–285.

[19] Seungmin Kang, Shin gyu Kim, Hyeonsang Eom, and Heon Young Yeom. 2012. Towards workload-aware virtual machine consolidation on cloud platforms. In *ICUIMC '12*.

[20] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. 598–610.

[21] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. *ACM SIGPLAN Notices* 49, 4 (2014), 729–742.

[22] Michael Kerrisk. 2021. taskset(1) — Linux manual page. https://man7.org/linux/man-pages/man1/taskset.1.html.

[23] Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. 2011. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *J. Parallel and Distrib. Comput.* 71, 5 (2011), 700–713.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ISCA '15*. 450–462.

[25] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. 2017. Improving Spark Application Throughput via Memory Aware Task Co-Location: A Mixture of Experts Approach. In *Middleware '17*. 95–108.

[26] Khang T Nguyen. 2019. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology/.

[27] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In *Proceedings of the 48th International Conference on Parallel Processing*. 15.

[28] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. {DeepDive}: Transparently identifying and managing performance interference in virtualized environments. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 219–230.

[29] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI '19*. 361–378.

[30] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *EuroSys '19*. 1–10.

[31] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. 2018. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *PACT '18*. 1–14.

[32] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *HPCA '20*. IEEE, 193–206.

[33] David F Richards, Omar Aaziz, Jeanine Cook, Hal Finkel, Brian Homerding, Peter McCorquodale, Tiffany Mintz, Shirley Moore, Abhinacv Bhatele, and Robert Pavel. 2018. *Fy18 proxy app suite release. milestone report for the ecp proxy app project*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[34] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *ISCA '21*. IEEE, 292–305.

[35] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 101–111.

[36] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[37] Johannes Thönes. 2015. Microservices. *IEEE Software* 32, 1 (2015), 116–116.

[38] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: dynamic cache allocation with partial sharing. In *EuroSys '18*. 13.

[39] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2019. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *ICPP '19*. 16.

[40] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dCat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *EuroSys '18*. 14.

[41] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 607–618.

[42] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3. 0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5 (2017), 1–16.

[43] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *EuroSys '13*. 379–391.

[44] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 2014. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 406–418.

[45] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*. 33–47.

[46] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *OSDI '20*. 1225–1240.