OLPart: Online Learning based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers

Abstract

Coloating multiple jobs on a same server has been a commonly used approach for improving resource utilization in cloud environments. However, performance interference due to the contention over shared resources makes resource partitioning an important research problem. Partitioning multiple resources coordinately is particularly challenging when multiple latency-critical (LC) jobs are colocated with best-effort (BE) jobs, since the QoS needs to be protected for all the LC jobs. So far, this problem is not well-addressed in the literature.

We propose an online learning based solution, named OLPart, for partitioning resources among multiple colocated LC jobs and BE jobs. OLPart is designed based on our observation that runtime performance counters can approximately indicate resource sensitivities of jobs. Based on this finding, OLPart leverages contextual multi-armed bandit (CMAB) to design the partitioning solution, which employs the performance counters to enable an intelligent exploration of the search space. Applying CMAB to the resource partitioning problem faces several critical challenges. OLPart proposes several techniques to overcome these challenges. OLPart does not require priori knowledge of jobs and incurs very small overhead. Evaluations demonstrate that OLPart is optimally efficient and robust, which outperforms state-of-the-art solutions with significant margins. *OLPart is publicly available at https://github.com/oksdfncsj/OLPart.*

1 Introduction

Modern data centers and cloud computing platforms commonly colocate latency-critical (LC) jobs (e.g., query processing, recommendation etc) with throughput-oriented best-effort (BE) jobs (e.g., big data, graph analytics etc) on the same commodity server for maximizing resource utilization. However, jobs sharing the same server would contend for shared resources, e.g, CPU cores, last level cache (LLC), memory bandwidth etc, leading to interference and performance degradations. Since the LC jobs usually have strict quality of service (QoS) requirement on response latency, interference may cause QoS violations of LC jobs, leading to tremendous revenue loss [36, 40].

To avoid interference, conservative solutions either entirely disallow job colocations or optionally colocate jobs whose interference is not serious, causing low server resource utilization. A more aggressive approach is to use OS-level and hardware-level isolation techniques to partition resources among colocated jobs. This approach can eliminate interference completely, which ensures QoS for the LC jobs and allow BE jobs to benefit from unused resources. Previous studies [26, 44] have summarized that a desirable resource partitioning solution should have the following properties:

- **Optimality.** A desirable solution should make optimal partitioning decisions, since suboptimal decisions would lead to either QoS violations of LC jobs or throughput sacrifice of BE jobs.
- **Efficiency & Robustness.** A desirable solution should be lightweight yet responsive, for adaptively responding to runtime changes (e.g., workload variations) which happen frequently in cloud platforms.
- **Autonomy.** A desirable solution should not make assumptions about prior knowledge of the jobs, because collecting such knowledge would be prohibitively expensive or even not feasible in many practical environments.

However, designing a resource partitioning solution achieving all the desirable properties faces several critical challenges, especially when multiple LC jobs are colocated together since the QoS of every LC job needs to be protected. First, the solution space of resource partitioning is prohibitively large when multiple resources are considered coordinately (Section 2.1). Exploring such a huge search space to find an optimal solution is not feasible as it is very time-consuming. Second, developing precise analytical performance models (which describe the relationship between job performance and resource

allocation) to guide the search process of the optimal solution is challenging. This is because: (1) building accurate performance model is difficult due to multiple impacting factors interacted with each other; (2) prior knowledge of jobs is typically required, which is not available in many practical environments such as public clouds.

To address these challenges, many resource partitioning solutions for job colocations have been proposed. Earlier works [8, 25, 34, 35, 37] usually establish analytical performance models to guide the search of an optimal partitioning configuration. However, such solutions can only handle one resource, since building accurate performance models is very challenging when multiple resources are considered. [8, 14, 32] are able to partition multiple resources simultaneously. They focus on colocating only BE jobs for optimizing fairness or throughput, which are not applicable to LC jobs that have strict QoS requirement. Resource partitioning solutions for colocating LC job with BE jobs also have been investigated [7, 20, 27, 33, 38, 39, 41, 45]. Nevertheless, they can only handle one LC job, while colocating multiple LC jobs is desirable in many cloud environments with more and more throughput-oriented jobs becoming increasingly latency critical [9, 33]. PARTIES is the first solution for colocating multiple LC jobs with BE jobs. However, it suffers extreme inefficiency when exploring a large search space (Section 2.1), because it adjusts one resource with small steps at a time. CLITE overcomes the limitation of PARTIES, which partitions multiple resources coordinately and explores the search space more efficiently. However, due to the unawareness of detailed runtime status (e.g., resource sensitivities), CLITE still falls short on optimality and efficiency (Section 2.2).

Our analysis (Section 2.3) shows that the primary reason leading to suboptimality of the existing solutions (e.g., PARTIES and CLITE) is that they are agnostic to the resource sensitivities of jobs in the searching process. Resource sensitivities can tell how a job's performance would change when the resource allocation varies, which is indispensable for designing efficient resource partitioning algorithms. Obtaining the resource sensitivities typically requires offline profiling [6, 14–16, 40, 42], which would be prohibitively expensive or not feasible. Based on a preliminary study (Section 4.2), we observe that runtime performance counters (e.g., cache miss rate etc.) can approximately indicate resource sensitivities of jobs. However, how to utilize the performance counters to design efficient resource partitioning algorithms is still an open problem and needs to be investigated.

In this paper, we address the resource partitioning problem for colocating multiple LC jobs with BE jobs, aiming to maximize the throughput of the BE jobs with the QoS guarantee for the LC jobs. To address the challenges and overcome the limitations of existing solutions, we propose OLPart, an online learning based resource partitioning solution for colocating multiple LC jobs with BE jobs. The most promising feature of OLPart is that it leverages contextual multi-armed bandit (CMAB) [10, 18, 24, 28, 43] to employ runtime performance counters of jobs to enable an intelligent exploration of the search space. Applying CMAB to the resource partitioning problem is non-trivial due to several critical challenges. OLPart proposes several new techniques to overcome these challenges.

Our evaluations show that OLPart owns all the properties that a good resource partitioning desires, which is lightweight, optimally efficient and robust. Moreover, OLPart outperforms the state-of-the-art baselines with significant margins. Specifically, OLPart is able to coloate more LC jobs and at much higher loads, with an advantage by from 40% to 80% over PARTIES and CLITE. OLPart also can improve the throughput of the BE jobs by up to 8.35x compared to PARTIES and CLITE.

2 Motivation

2.1 Challenges of Resource Partitioning

The search space to be explored for finding the optimal partitioning configuration is prohibitively large. Consider the problem of partitioning M resources among N jobs. For each resource i, let K_i denote the capacity of i (i.e., the total number of units). The total number of partitioning configurations of the M resources among the N jobs will be $\prod_{i=1}^{M} C_{K_i-1}^{N-1}$, which is up to 2×10^6 when there are five jobs, three resources and each resource has 10 units. This number could be even higher when the resources are allowed to be partially shared (e.g., cache ways can be partially shared according to Intel's CAT [31]) for a more fine-grained partitioning. Since the desired partitioning solution should be sufficiently responsive, exploring the full search space to find the optimal partitioning configuration is not feasible as it is very time-consuming.

Developing precise analytical performance models to guide the exploration of search space is challenging. If the relationship between job performance and resource allocation can be described by some analytical performance models, the resource partitioning problem can be solved more efficiently using mathematical optimization technologies (e.g., gradient descent) or meta-heuristics (e.g., simulated annealing), instead of evaluating every possible configuration in real system. However, developing accurate analytical performance models is challenging, because the relationship between job performance and resource allocation is very complex due to multiple impacting factors.

Figure 1 shows how the performance of two LC jobs is affected by the allocation of two resources (see details of jobs and experimental platform in Section 5). We have several observations. First, the performance trends are different across the LC jobs, indicating that the LC jobs have different sensitivities to the resources. Second, the sensitivity of a LC job on one resource is impacted by the allocation of another resource (e.g., *moses* exhibits different sensitivity to LLC when

资源敏感性的具体讲说

it is allocated 1 and 2 cores). Third, resource sensitivities are different for the same LC job under different load pressures (Figure 1(b) vs. Figure 1(c)). These observations imply that the performance of jobs is jointly determined by many factors, which cannot be easily analyzed as more jobs and resources are added.

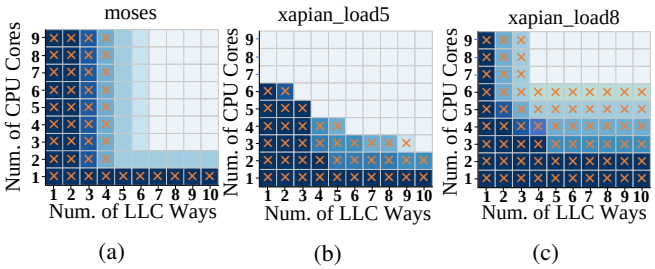


Figure 1: Tail latency of LC jobs over different resource allocations. A darker color represents a higher latency. The resource allocations not meeting QoS requirement are marked by 'x'. Figure 1(b) and Figure 1(c) are under different load pressures.

2.2 Limitations of Existing Solutions

PARTIES and CLITE are the only two approaches towards partitioning resources for colocating multiple LC jobs with BE jobs. However, we find that both of them suffer from suboptimality and inefficiency.

PARTIES PARTIES continuously monitors the latencies of the LC jobs and dynamically adjusts the resource allocations according to each LC job's latency slack (the difference between the target QoS and the observed latency). If at least one LC job's QoS is violated, PARTIES ties to continuously shift resources from the LC job with the largest latency slack to the LC job with the smallest latency slack, one resource unit at a time, until all the LC jobs satisfy the target QoS (if possible). After that, the remaining resources are allocated to the BE jobs for improving the throughput.

The core merits of PARTIES are simplicity and autonomy, but it suffers from low efficiency and suboptimality. The key reason is that PARTIES explores only one resource and tunes the resource in small steps for one LC job at a time. This significantly limits the searching efficiency, because resource sensitivities of jobs are complex, and increasing one resource may not reach the target QoS if other resources are not appropriately resized.

CLITE CLITE adjusts multiple resources coordinately. It leverages Bayesian Optimization to build low-cost performance model by sampling a small number of points in the large search space. The performance model can approximately estimate the quality of different resource partitioning configurations, which enables intelligent exploration of the search space to find near-optimal configurations.

Although CLITE outperforms PARTIES for both optimality and efficiency [33], we shall show that CLITE is still far from optimal. The core reason is that the performance model used by CLITE takes a single scalar value to indicate the quality of each sampled partitioning configuration, which lacks detailed runtime status of jobs. This would lead CLITE to make wrong decisions. For example, a value smaller than 0.5 indicates that at least one LC job's QoS is violated, but which LC jobs are not satisfying QoS is unknown. In this case, CLITE may add more resources to a LC job whose QoS is already met. Without the runtime status, CLITE is unable to learn the resource sensitivities of jobs. For example, if an LC job's latency is decreased after adding two resources simultaneously, CLITE cannot identify which resource contributes to the performance improvement, and may continuously increase the resource that the job does not require.

CLITE调节资源具有盲目性，这是由于CLITE不知道运行时具体状态（上下文）而导致的

Figure 2 shows the searching path of CLITE for a colocation of two LC jobs (*img-dnn* and *masstree*) sharing two resources (memory bandwidth and CPU cores). It can be seen that CLITE made several wrong decisions in the exploring process. For example, CLITE adds more resources to *img-dnn* in step 21, but *img-dnn* already satisfies the QoS. The wrong decisions significantly degrade the exploring efficiency, and the situation will get much worse as more LC jobs and more resources are added.

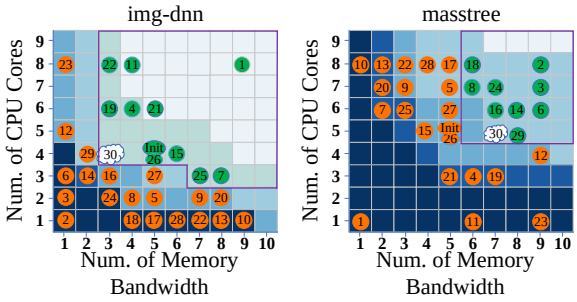


Figure 2: Search path of CLITE for partitioning two resources among two colocated LC jobs. The numbers denote the steps and the location of each number denotes the resource configuration sampled in this step. The region encircled by purple line indicates the QoS safe regions.

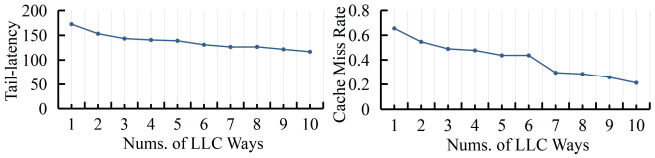
2.3 Opportunities

Our analysis on CLITE indicates that an intelligent partitioning strategy must be aware of the resource sensitivities of jobs to avoid wrong resource tuning decisions. However, learning the resource sensitivities through offline profiling or using analytical performance models are both impractical, as we have analyzed earlier. Whereas, we observe that how a job is sensitive to a specific resource can be approximately indicated by some runtime performance counters (e.g., cache miss rate

1、算法获知资源敏感性很重要；

2、利用performance count来近似获知资源敏感性，比离线方式更切合实际

etc). For example, a small LLC miss rate indicates that the LC job is not sensitive to LLC and adding more cache ways is not beneficial. Figure 3 shows how the tail latency of a LC job (*specjbb*) is correlated with LLC miss rate. It can be seen that the changing trends of tail latency and LLC miss rate are highly consistent as the resource configuration varies, which validates that LLC miss rate is able to indicate the sensitivity of the job to LLC.



(a) (b)

Figure 3: (a) Tail latency of *specjbb* over different LLC allocations; (b) Cache miss rate of *specjbb* over different LLC allocations.

We have performed studies to validate this finding for more LC jobs and resources. In order to find the performance counters that can indicate the sensitivity of an LC job j to a resource r, we generate 1000 different resource allocations for job j, where the allocation of each resource is randomly generated. For each resource allocation, we varies the allocation of resource r from the minimum units to the maximum units while keeping the allocations of other resources fixed. We measure the latency of j and the performance counters to be examined. Then, we compute the linear correlation between the tail latency and each performance counter for each resource allocation. We evaluate 6 LC jobs, 3 resources and over 100 performance counters. Table 1 summarizes the top 5 performance counters mostly correlated with each resource for each job. We have two observations: first, there are always some highly correlated performance counters for each resource and each LC job; second, the performance counters to indicate the same resource are highly overlapped for different LC jobs. The results indicate that the sensitivities of all resources can be indicated by runtime performance counters. Although this finding provides promising opportunities, how to use these performance counters to design efficient partitioning solutions is still an open problem to be investigated.

3 OLPart: An Overview

OLPart is designed following four design principles:**(1) multiple resources are explored coordinately; (2) lightweight and responsive; (3) no prior knowledge is required; (4) runtime performance counters are used to enable an intelligent exploration of the search space.** In order to achieve all the principles simultaneously, OLPart leverages *contextual multi-armed bandit* (CMAB), an online learning approach that chooses from a set of actions in a sequence of trials so as to maximize the total payoff of the chosen actions. In the rest of this section, we first give the background knowledge of CMAB. Then, we explain why CMAB is suitable for our resource partitioning problem. At last, we summarize the challenges of applying CMAB to our problem.

3.1 Contextual Multi-armed Bandit

The multi-armed bandit (MAB) problem is defined as follows: imagine a gambler is given a set of k arms; each arm a is associated with an unknown probability distribution p_a; suppose the gambler plays the game for T rounds; in round t, the gambler picks some arm j, and he will observe a reward r_t, which is a random sample from p_j; the objective is to maximize the expected total reward over T, i.e., $\max \sum_{t=1}^{T} r_t$. The MAB has been used to model many choice making problems [4, 17].

Since the expected reward of each arm is unknown, a good bandit algorithm needs to trade-off between *exploration* (take some risk to try new arms) and *exploitation* (always take the best arm observed so far) for maximizing the expected sum of rewards. A popular approach called upper confidence bound (UCB) implements this trade-off by considering both each arm's estimated reward and its uncertainty. Specifically, let $N_{a,t}$ denote the number of times that arm a has been selected so far, and $\mu_{a,t}$ denote the mean observed reward of arm a till step t. The uncertainty is estimated by $\sqrt{\frac{\ln t}{N_{a,t}}}$, where $\ln t$ is the natural logarithm of t. With the estimated reward and uncertainty, the UCB approach will choose the arm at step t with the maximum $\mu_{a,t} + c\sqrt{\frac{\ln t}{N_{a,t}}}$, where the number $c > 0$ controls the degree of exploration.

A useful generalization of the MAB is *contextual multi-armed bandit* (CMAB). At each iteration, the gambler still has to choose between arms, but they also see a d-dimensional feature vector (some runtime context information), which can be used together with the observed rewards of the arms to make better choice. The main challenge of CMAB is how to relate the feature vector and the observed rewards, so that the feature vector can be made full use of. A popular approach for solving the CMAB problems is LinUCB [1, 11, 12]. Similar to the UCB approach [3, 23], LinUCB also chooses the arm according to the estimated mean rewards and uncertainties. The difference is that LinUCB learns a linear model to estimate the rewards according to the feature vector. Let $x_{a,t}$ denote the d-dimensional feature vector associated with arm a observed at time step t. The expected reward of arm a, denoted by $\mu_{a,t}$, is estimated by $\mu_{a,t} = x_{a,t}^T \theta_a$, where θ_a is the coefficient vector of dimension $1 \times d$. To learn θ_a, for each arm a, LinUCB maintains a parameter matrix A_a of $d \times d$ dimension and a parameter vector b_a of $d \times 1$ dimension. Initially, A_a is set as d-dimensional identity matrix and b_a is set as d-dimensional zero vector for each arm a. If arm a is selected at time step t, A_a will be updated according to

$$A_a \leftarrow A_a + x_{a,t} X_{a,t}^T \quad (1)$$

Table 1: The top five performance counts mostly correlated with each resource. A darker color indicates a higher coefficient.

LC job	Resource	Counter				
masstree	MB	cycles	task-clock	ld_blocks.no_sr	cpuhz	branch-load-misses
	LLC	cache-misses	uops_retired.retire_slots	LLC-store-misses	offcore_requests.demand_data_rd	cycles
	Core	branch-loads	dTLB-loads	cycles	offcore_requests.demand_data_rd	instructions
xapian	MB	LLC-store-misses	branch-load-misses	l2_rqsts.all_demand_miss	inst_retired.total_cycles_ps	inst_retired.prec_dist
	LLC	cache-misses	offcore_requests.demand_data_rd	LLC-store-misses	uops_retired.retire_slots	branch-load-misses
	Core	cpuhz	branch-loads	inst_retired.prec_dist	ld_blocks.store_forward	cycles
img-dnn	MB	cpuhz	LLC-store-misses	cycles	ld_blocks.no_sr	branch-load-misses
	LLC	branch-misses	L1-dcache-stores	L1-dcache-loads	instructions	ld_blocks.no_sr
	Core	cpuhz	frontend_retired.latency_ge_2_bubbles_ge_3	LLC-store-misses	cache-misses	uops_retired.retire_slots
sphinx	MB	task-clock	cycles	ld_blocks.no_sr	inst_retired.prec_dist	uops_retired.retire_slots
	LLC	offcore_requests.demand_data_rd	uops_retired.retire_slots	LLC-store-misses	task-clock	cache-misses
	Core	cycles	instructions	ld_blocks.store_forward	cache-misses	LLC-store-misses
moses	MB	task-clock	inst_retired.total_cycles_ps	dTLB-load-misses	L1-dcache-stores	L1-dcache-load-misses
	LLC	LLC-store-misses	task-clock	cache-misses	uops_retired.retire_slots	offcore_requests.demand_data_rd
	Core	cpuhz	L1-dcache-load-misses	inst_retired.total_cycles_ps	cache-misses	ld_blocks.no_sr
specjbb	MB	cpuhz	cycles	L1-dcache-loads	L1-dcache-stores	dTLB-loads
	LLC	dTLB-loads	L1-dcache-loads	ld_blocks.store_forward	cache-misses	LLC-store-misses
	Core	cache-misses	uops_retired.retire_slots	cpuhz	ld_blocks.store_forward	dTLB-loads

and b_a will be updated according to

$$b_a \leftarrow b_a + r_t x_{a,t} \tag{2}$$

With the parameter matrix A_a and the parameter vector b_a, the coefficient vector θ_a is estimated by $A_a^{-1} b_a$ at time step t. So, the expected reward of arm a is estimated according to $\mu_{a,t} = x_{a,t}^T A_a^{-1} b_a$. The uncertainty of the reward is estimated by $\sqrt{x_{a,t}^T A_a^{-1} x_{t,a}}$. Let $p_{a,t}$ denote the sum of the estimated reward and uncertainty, i.e.,

$$p_{a,t} = \mu_{a,t} + \sqrt{x_{a,t}^T A_a^{-1} x_{t,a}} \tag{3}$$

LinUCB chooses the arm with the maximum $p_{a,t}$ at time step t.

3.2 Why CMAB is Suitable for the Resource Partitioning Problem?

We use CMAB to model the resource partitioning problem, and leverages LinUCB to design OLPart. The reason is four-fold. First, the resource partitioning problem can be naturally modeled as a MAB problem, where the partitioning configurations correspond to the arms and the expected reward of an arm refers to the performance of the corresponding resource partitioning. Second, the bandit algorithm LinUCB incurs very small computational overhead (see the analysis in 5.4), which meets our objective that wants to design a lightweight and responsive partitioning solution. Third, no prior knowledge of jobs is required by LinUCB for making decisions, and all the required information (the observed rewards and the contextual feature) can be collected online. Fourth, also most importantly, the CMAB approach makes decisions by referring a contextual feature, which exactly matches our design principle that wants to use runtime information (i.e., the performance counters) to guide the exploration .

3.3 Challenges of Applying CMAB to the Resource Partitioning Problem

Despite CMAB seems a promising approach, applying it to the resource partitioning problem still faces critical challenges. First, as has been shown in Section 2.1, the total number of resource partitioning configurations to be explored is very huge. A native approach that constructs one arm for each partitioning configuration is impractical, because choosing the best arm among such a large number of arms is time-consuming. Therefore, the arms and bandits should be carefully designed. Second, designing an appropriate reward for each arm selection is not easy. This is because the reward should be able to represent multiple optimization objectives, guaranteeing the QoS for each LC job while maximizing the throughput of the BE jobs. An improper design of the reward may lead to inefficient exploration process.

4 Design of OLPart

4.1 Definition of Bandits and Arms

In order to maintain a reasonable number of arms, we use a decentralized strategy which constructs separate bandits for different resources and jobs, while the bandits make decisions collaboratively. OLPart allows a resource to be either *completely-isolated* (each unit of resource belongs to only one job) or *partially-shared* (a unit of resource may belong to multiple jobs). Let J denote the set of jobs colocated and R denote the set of resources to be partitioned. For each job $j \in J$ and each resource $r \in R$, we construct a bandit $B_{j,r}$ which represents the allocation of resource r to j. Let N_r denote the total number of allocation schemes of resource r for each job, e.g., for a server with 10 CPU cores, there are 10 allocation schemes of CPU cores for each job, from 1 core to 10 cores. The bandit $B_{j,r}$ has N_r arms, with each arm corresponding one allocation scheme. Each arm of the bandit $B_{j,r}$ maintains its own parameter matrix A and parameter vector b. If resource r is partially-shared, the bandits of different jobs will choose their arms independently at each time step. Otherwise, if re-

llc 和 mb都是可并行共享的，因此资源划分个数不需要与jobs个数相等，分配方案不需要关心对应的jobs个数，只需一个列表穷举所有分配方案即可，某次决策的不同job的资源划分方案可以重叠

cpu cores是完全独立的资源，因此它在资源划分时，对应不同jobs个数的划分方式要分开存储

source r is completely-isolated, it means that the allocations of resource r to the jobs should satisfy a hard constraint, i.e., the sum of the allocated units to all jobs should be equal to the resource capacity. To ensure this constraint, we use a greedy algorithm to choose arms for the bandits of all jobs collaboratively (Section 4.3). At each time step, after the arms are chosen for all bandits, we compute a unified reward (Section 4.4) for all the arms at this step, and update the parameter matrix A and parameter b for each arm accordingly.

By this design, we maintain a number of $|J|\sum_{r\in R}N_r$ arms, linear to the number of jobs and resources, which is far less than the total number of partitioning configurations.

4.2 Contextual Feature

OLPart employs runtime performance counters as the contextual feature for guiding the exploration of the search space. For each resource $r \in R$, we find the top k performance counters which are mostly correlated to resource r (using the approach in Section 2.3), denoted by set C^r. Let $C = \cup_{r\in R}C^r$, which represents the full set of performance counters to be used in OLPart. It it worth noting that profiling some jobs to find out the performance counters does not mean that we need prior knowledge of jobs, because the performance counters are applicable to any jobs as we have analyzed in Section 2.3.

Denote by a $|C| \times 1$ vector $C_{j,t}$ the observed values of performance counters in C associated with job j at time step t. For a specific bandit $B_{j,r}$, the contextual feature at step t for the arms of $B_{j,r}$ is defined as the vector $[C_{j,t}, \frac{1}{|J|-1}\sum_{j'=1,j'\neq j}^{|J|}C_{j',t}]$, where $\frac{1}{|J|-1}\sum_{j'=1,j'\neq j}^{|J|}C_{j',t}$ denotes the average performance counters of other jobs at step t. The intention of including the performance counters of other jobs is that the selection of arms is not independent among different resources and the sensitivity of one resource is also instructive for the allocation of other resources.

4.3 Arm Selection Strategy

Consider a specific bandit $B_{j,r}$, i.e., the bandit for job j and resource r. If resource r is partially-shared, the bandit $B_{j,r}$ chooses the arm at each step according to the standard LinUCB algorithm (the definition of reward r_t is detailed in Section 4.4). Otherwise, if resource r is completely-isolated, the bandits for resource r of all jobs, i.e., $\{B_{j,r}, j \in J\}$, choose their arms collaboratively through a beam-search based greedy algorithm.

The main idea of the greedy algorithm is to sequentially choose the arms for the jobs, always maintaining the top L partial selection results that achieve the best estimated rewards so far, and finally picks the best result among these candidates. Without loss of generality, we index the jobs from 1 to $|J|$. For job 1, it chooses the top L arms (denoted by a set A_1) with the highest estimated rewards (i.e., $p_{t,a}$ defined in Equation 3). Then, for each job j from 2 to $|J|$, the algorithm chooses the top L tuples $(a_{i_1}, a_{i_2}, \ldots, a_{i_j})$ (denoted by set A_j) with the highest total estimated reward of the arms in the tuple (i.e., $\sum_{m=1}^{j} p_{a_{i_m},t}$), such that $(a_{i_1}, a_{i_2}, \ldots, a_{i_{j-1}}) \in A_{j-1}$ and $\sum_{m=1}^{j}\Gamma(a_{i_m}) \leq K_r$, with $\Gamma(a_{i_m})$ denoting the amount of resource r allocated to job m and K_r denoting the capacity of resource r. After $A_{|J|}$ is generated, the algorithm chooses the tuple $(a_{i_1}, a_{i_2}, \ldots, a_{i_{|J|}})$ with the highest estimated reward (i.e., $\sum_{m=1}^{|J|} p_{a_{i_m},t}$) as the final solution, where a_{i_m} represents the arm selected for job m.

4.4 Design of Reward

Since the bandits determine the resource partitioning configuration altogether, we define a unified reward shared by all the bandits. The reward should be properly designed to evaluate the quality of resource partitioning configurations to guide OLPart in the right direction in the large search space. To this end, we classify the system into two situations: QoS-violating (some LC jobs violate the QoS) and QoS-satisfying (all the LC jobs satisfy the QoS). We define separate reward for each of situation. When the system is under QoS-violating situation, our primary goal is to meet the QoS for all the LC jobs as soon as possible. Therefore, we use a negative value -1 as the reward, which would suggest the bandits escape from the current partitioning configuration quickly. Once the system becomes QoS-satisfying, our goal turns to maximizing the throughput of the BE jobs. In this case, we use the total instructions per cycle (IPC) of the BE jobs as the reward, which is a commonly used metric to evaluate the throughput (a larger IPC indicates a higher throughput).

4.5 Maintaining Multi-Versioned Bandits

In the practical environment, both LC jobs and BE jobs have dynamic workload changes. Although OLPart can learn an updated model through continuous interactions with a changeable environment, such update may not be timely enough to achieve consistently optimal performance. To address this issue, we construct a new version of bandits at regular intervals. The new version of bandits are trained by the recently observed rewards, which are more optimal for the current environment compared to the old versions. Moreover, in order to make the partitioning decision more robust, we also maintain a set of old versions of bandits constructed previously. At each step, among the partitioning decisions made by all the versions of bandits, we choose the partitioning decision achieving the best estimated reward.

4.6 Reusing the Bandits of Previous Jobs

The LinUCB algorithm learns the bandit parameters (i.e., the parameter matrix A and the parameter vector b) from scratch

when a bandit is newly constructed. This "cold-start" can be optimized in our context. This is because in practical cloud environments, a job in the current colocation may have appeared in the previous colocations, thus the bandits of the job previously constructed can be directly reused by the current colocation for a quick start. To do this, OLPart records the parameters of all the previous jobs' bandits. When a job arrives, OLPart first uses a classifier to check (according to some identifier, e.g., the job's name) whether the job has been seen before. If the job has appeared before, the bandits associated with this job are initialized using the recorded bandit parameters of this job. Otherwise, if the job has not appeared before or it is infeasible to identify whether a job has appeared or not, we initialize the bandits associated with this job using the averaged bandit parameters of all the recorded jobs. The intention is that even the job is different from the recorded jobs, the bandit parameters of the recorded jobs are also helpful, because some knowledge learnt by the recorded bandits are also applicable to the job, e.g., a small LLC miss rate indicates that the job needs no more LLC.

4.7 Putting It Together

The overall design of OLPart is shown in Figure 4. OLPart works in an online manner. When a new colocation arrives, the resources are first partitioned according to some default strategy, e.g., equally partitioned. After OLPart constructs the bandits, it periodically produces and updates partitioning decisions. At each step, OLPart first collects the required performance counters associated with the jobs. Then, it chooses the arms, generates the partitioning decisions according to the selected arms, and applies the generated partitioning decision to real system. After that, OLPart observes the performance of the jobs and computes the reward accordingly. With the reward, OLPart updates the bandit parameters accordingly. When the allowed search time is reached (sets 60s as other methods), OLPart terminates its search process. OLPart divides the search space by reducing the number of arms and keeping the low length of contextual features for each bandit of each jobs to trade off between performance and overhead.

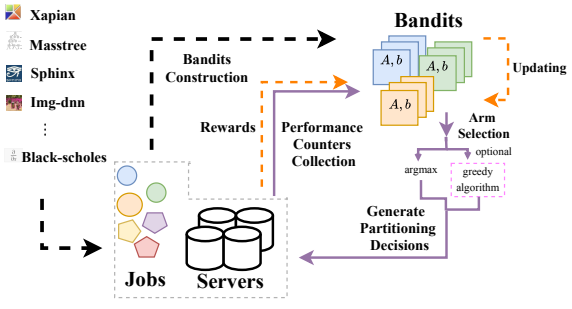


Figure 4: The Overall Design of OLPart

5 Evaluations

5.1 Experimental Settings

Table 2: Experimental testbed configuration.

Component	Specification
Processor	Intel(R) Xeon(R) Silver 4114
Processor Speed	2.20GHz
Logical Processor Cores	20 Cores(10 physical cores)
Private L1 & L2 Cache Size	32 KB and 1024 KB per core
Shared L3 Cache Size	13.75MB
Memory Capacity	128GB
Operating System	CentOS7.8 with linux4.10.0
SSD Capacity	1TB

Platform. We implement and evaluate OLPart on a platform with detailed configurations listed on Table 2. For all the experiments, the Hyper Threading and Turbo Boost features of the CPU are disabled. To reduce unnecessary distractions, we leave one CPU core to run the OS and the other CPU cores to run colocations.

Resources. We particularly consider the partitioning of three most important resources: CPU cores, LLC and memory bandwidth. We use *taskset*, Intel CAT [31] and Intel MBA [2] to implement the partitioning of CPU cores, LLC and memory bandwidth respectively. The CPU cores are partitioned in completely-isolated manner. For a server with 10 CPU cores, there would be 10 allocation schemes for a job, from 1 core to 10 cores. LLC is partitioned in partially-shared manner. Intel's CAT organizes the LLC as a set of sequentially indexed cache ways and requires the cache ways allocated to a job to be continuous (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. The memory bandwidth is also partitioned in partially-shared manner. Intel's MBA can specify the maximum proportion from the entire bandwidth that an application can use, where the maximum proportion can be one of ten levels: $\{10\%, 20\%, \ldots, 100\%\}$.

Table 3: LC and BE jobs used for evaluation.

Latency-Critical (LC) Job Workloads			
Job	Domain	Target QoS	Max Load
masstree(MS)	Key-value store	100 msec	330
xapian(XP)	Online search	5 msec	6000
img-dnn(IM)	Image recognition	2 msec	4500
sphinx(SP)	Speech recognition	1500 msec	1
moses(MS)	machine translation	500 msec	300
specjbb(SJ)	Java middleware	1 msec	8000
Best-Effect(BE) Job Workloads			
blackscholes(BS)	Option pricing with Black-Scholes Partial Differential Equation (PDE)		
canneal(CN)	Simulated cache-aware annealing to optimize routing cost of a chip design		
fluidanimate(FM)	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method		
freqmine(FQ)	Frequent itemset mining		
streamcluster(SC)	Online clustering of an input stream		
swaptions(SW)	Pricing of a portfolio of swaptions		

Jobs. Table 3 lists all the LC and BE jobs used for evaluation. The loads for LC jobs are obtained from the Tailbench benchmark suite [22], which is specifically developed to capture characteristics of LC workloads. The BE loads are obtained from the PARSEC benchmark suite [5], which is widely used for evaluating throughput-oriented jobs. To determine the QoS requirements of the LC workloads, we run them alone on the server at different loads (measured by queries-per-second (QPS)). Similar to the method used by CLITE, we measure their 95^{th} percentile tail-latencies at these different loads. We define the QoS tail-latency of the LC jobs as the knee of these curves and the corresponding QPS is the maximum load for that job.

Configurations of OLPart. We test a large variety of parameter settings for OLPart, and find that the following configurations achieve sufficiently good performance while keeping acceptable overhead: OLPart makes partitioning decisions in every 3 seconds; 5 performance counters are selected for each resource; up to 3 versions of bandits are maintained; the beam-search algorithm for arm selection chooses top 5 arms for each bandit; OLPart terminates until the given search time limit is reached.

For each colocation, we allow each resource partitioning approach to explore the search space for a period of 60 seconds (same as in CLITE), and compare the final partitioning configuration produced by each approach. For fair comparison, we let the first partitioning decision be the same for all the partitioning approaches, where each resource is equally partitioned among the colocated jobs.

5.2 Baselines

We compare OLPart with several state-of-the-art baselines:

PARTIES. PARTIES [9] is the first work towards partitioning resources among colocated LC jobs and BE jobs. It makes incremental adjustments in one resource at a time until QoS is met for all LC jobs, and then allocates the leftover resources to the BG jobs.

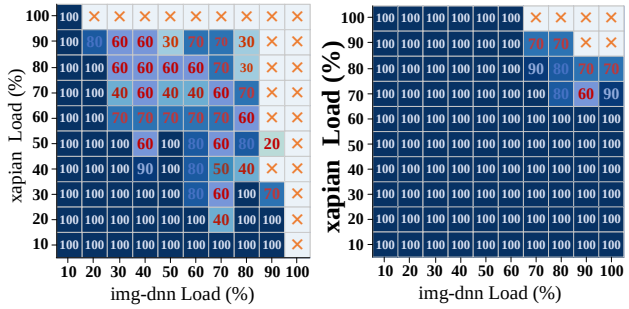
CLITE. CLITE [33] is the first work considers adjusting multiple resources coordinately. It leverages Bayesian Optimization to build approximate performance model, and uses the performance model to guide the exploration of the search space.

Brute-Force Search (ORACLE). ORACLE attempts to sample all feasible resource allocations offline and selects the one has the best performance as the result. However, this is unacceptable for our experimental settings since there are too many partitioning configurations. To address this issue, we search a smaller space by assuming all the three resources are completely-isolated. In this manner, ORACLE achieves a near-optimal performance.

5.3 Results and Analysis

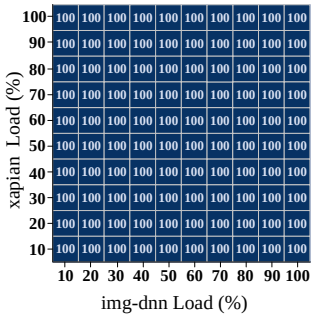
5.3.1 Optimality

OLPart can colocate more LC jobs and at much higher loads. To be intuitive, we consider one colocation of three LC jobs (*xapian*, *img-dnn* and *moses*), without any BE jobs. In each experiment, we fix the load of *xapian* and *img-dnn* at some loads, and gradually increase the load pressure of *moses* to find the highest tolerable load for guaranteeing the QoS for all the three jobs. Figure 5 summarizes the results achieved by OLPart compared to the baselines.

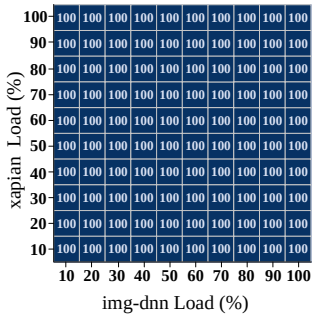


(a) PARTIES

(b) CLITE



(c) OLPart



(d) ORACLE

Figure 5: The x- and y- axis show the loads of *img-dnn* and *xapian*, respectively, and the value in the cell shows the highest tolerable load of *moses*. 'x' means that *moses* cannot be colocated at any load.

We have several observations. First, PARTIES performs worst among all the approaches. It cannot colocate *moses* at all for many load settings and coolocates *moses* at much lower load in many cases (e.g., when img-dnn load is 30% and xapian load is 70%). This is expected, because PARTIES tunes one resource at a time, which significantly limits the searching efficiency. This limitation is aggravated when the feasible solution space become smaller, i.e., when img-dnn and xapian have higher load. Second, CLITE performs much better than PARTIES. For example, CLITE can colocate *moses* when xapian load is 100% and img-dnn load is 20% to 60%, where PARTIES cannot. The improvement of CLITE over PARTIES confirms that coordinately exploring multiple resources is effective. However, CLITE is still not sufficiently intelligent, because there are still some cases that it cannot find feasible

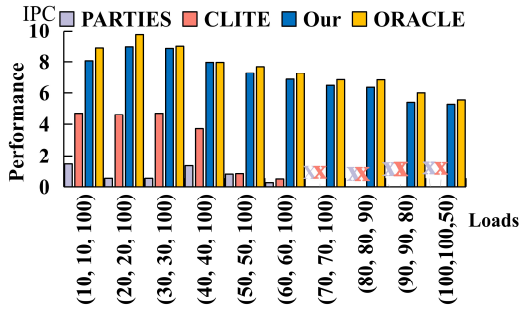


Figure 6: OLPart can support higher loads and better performance for batch jobs than PARTIES and CLITE.

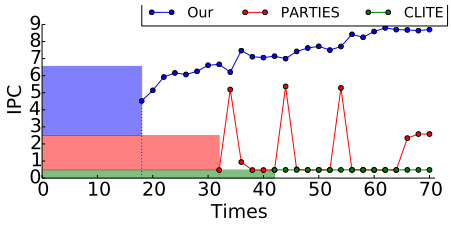


Figure 7: Real-time throughput of the BE job over time. The colored region means that no feasible configuration is found.

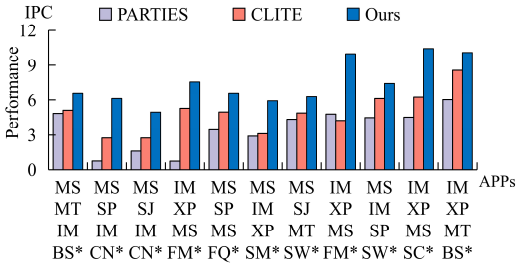


Figure 8: Performance of OLPart for more colocations. The x-axis denotes the colocated jobs, and the y-axis denotes the throughput of the BE job (marked by *).

configurations. Third, OLPart achieves similar performance to ORACLE, which outperforms CLITE and PARTIES significantly. OLPart is always able to colocate *moses* for vary load settings, which enables OLPart to colocate more LC jobs for many load settings (e.g., when xapian load is 100% and img-dnn load is 70% to 100%). Moreover, even when PARTIES and CLITE are able to colocate *moses*, OLPart is always able to colocate *moses* at the same or higher load (the advantage is up to 80% for PARTIES and 40% for CLITE). The results confirms that OLPart can achieve near-optimal partitioning configurations.

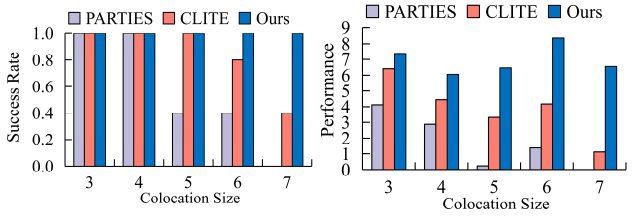
OLPart can significantly improve the throughput of BE jobs. In this experiment, we colocate three LC jobs (*xapian*, *img-dnn* and *moses*) with one BG job (*blackscholes*), and compare the throughput of the BG job achieved by each approach over different load settings. Figure 6 summarizes the results. It be seen that PARTIES achieves the lowest throughput among all the approaches. This is because once a feasible partitioning configuration (satisfying the QoS for all LC jobs) is found, PARTIES stops the searching process without trying to further optimize the throughput. CLITE performs much better than PARTIES, because it continuously optimizes the throughput even it has found a feasible configuration. OLPart achieves similar performance with ORACLE, which outperforms PARTIES and CLITE significantly, by 1.7x to 12x. We attribute this to that OLPart is able to explore the search space more efficiently. To be intuitive, we plot the real-time throughput of the BE job over time for one colocation in Figure 7. It can be seen that OLPart takes the shortest time to find a feasible configuration, while PARTIES and CLITE spend much longer time. Moreover, OLPart always achieves higher throughput than PARTIES and CLITE at the same time point. The results confirm that OLPart explores the search space more efficiently than PARTIES and CLITE.

Figure 8 shows the performance trend for more colocations. In each colocation, three LC jobs (randomly generated) are colocated with one BE job. The load of each LC job is randomly generated in the range from 50% to 100%. We observe that OLPart always beats PARTIES and CLITE with significant margin, by 1.17x to 8.35x. It confirms that the advantage of OLPart is consistent for various colocations.

5.3.2 Scalability

We randomly generate 20 colocations for each colocation size ranging from 2-jobs to 6-jobs. Each colocation has only one BE job and the others are LC jobs. The load of each job is randomly generated from 10% to 60%. Figure 9 (a) plots the success rate that each approach is able to find a feasible configuration for different colocation sizes. We observe that the success rate decreases for all the approaches as colocation size grows. This is reasonable because as more jobs competes for the resources, finding a partitioning configuration satisfying the QoS for all LC jobs becomes more difficult. PARTIES and CLITE cannot even find a feasible configuration when colotion sizes are larger than 6. For PARTIES, this is because as the search space grows too fast, adjusting one resource with small steps at a time gets extremely inefficient. For CLITE, this is because Bayesian Optimization uses Gaussian Processes to fit the performance model, which will lose effectiveness in higher dimensions due to large computational overhead. In contrast, OLPart always can find feasible configurations for different colocation sizes, and has the highest success rate for all colocation sizes compared to PARTIES and CLITE. The results again confirm its searching efficiency. Figure 9 (b) shows the averaged throughput of the BE job achieved by each approach for different colocation sizes. It can be seen that the throughput decreases as the colocation size grows. This is because as the number of jobs grows, the resources allocated to the BE job is reduced. The results

demonstrate that OLPart has a good scalability to varying colocation sizes.

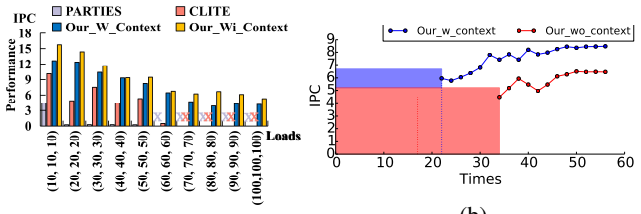


(a) (b)

Figure 9: Scalability of OLPart to varying colocation sizes. (a) The success rate of finding a feasible configuration; (b) The averaged throughput of the BE job achieved by each approach.

5.3.3 Effectiveness of Contextual Feature

In this experiment, we compare the performance of OLPart with/without considering the contextual feature. Figure 10 (a) shows the performance trend (in terms of the throughput of the BE job) for a colocation with three LC jobs (*img-dnn*, *xapian*, *moses*) and one BE job (*blackscholes*) over different load settings. It can be seen that the OLPart without contextual feature achieves much lower throughput for most of the load settings, or even cannot find feasible configuration for some load settings. Figure 10 (b) plots the real-time throughput over time for a specific load setting. We observe that the OLPart with contextual feature can find a feasible configuration much faster, and always achieves higher throughput at the same time point compared to that without contextual feature. The results confirm that the contextual feature enables a more efficient exploration of the search space.



(a) (b)

Figure 10: Performance of OLPart with/without contextual feature. (a) The throughput of the BE job for different load settings; (b) Real-time throughput for a specific load setting. The x-axis denotes the loads of the three LC jobs. 'x' denotes that no feasible solution is found.

5.3.4 Effectiveness of Bandits Reusing

One promising feature of OLPart is that it can reuse the bandits of previous jobs to solve the "cold-start" issue for newly arrived jobs. To demonstrate the effectiveness of this mechanism, we run two colocations consecutively (with each colocation running 60 seconds) under three different scenarios. The details are summarized in Table 4.

Table 4: Performance of OLPart with bandits reusing under different scenarios. 'X' means that no feasible configuration is found.

Scenario I: first colocation ({*specjbb*, *moses*, *sphinx*,*fluidanimate*), second colocation (*img-dnn*, *xapian* and *masstree*,*blackscholes*)										
Times	10	20	30	40	50	60	70	80	90	100
IPC	X	X	X	X	4.15	4.78	5.02	5.12	6.12	6.28
Scenario II: first colocation ({*img-dnn*, *moses*, *sphinx*,*blackscholes*), second colocation (*img-dnn*, *xapian* and *masstree*,*blackscholes*)										
Times	10	20	30	40	50	60	70	80	90	100
IPC	X	X	X	5.12	5.35	6.12	6.28	6.83	6.98	7.03
Scenario III: first colocation (*img-dnn*, *xapian* and *masstree*,*blackscholes*), second colocation (*img-dnn*, *xapian* and *masstree*,*blackscholes*)										
Times	10	20	30	40	50	60	70	80	90	100
IPC	6.26	5.96	6.35	6.86	6.89	6.95	7.03	6.98	7.02	7.03

Table 4 summarizes the real-time performance (refers to the throughput of the BE job in the second colocation) of OLPart over time for different scenarios. We observe that OLPart achieves the worst performance for Scenario I and the best performance for Scenario III. We attribute this to the reusing of the previous jobs' bandits. In Scenario I, no LC job in the second colocation has appeared in the first colocation, thus OLPart needs to learn the bandit parameters from scratch. In contrast, all the LC jobs in the second colocation have appeared for Scenario III, thus all the bandits for the LC jobs can be reused. The results demonstrate that bandit reusing can significantly improve the searching efficiency. Note that both PARTIES and CLITE have no such mechanism. They need to explore the search space from scratch for every colocation, no matter whether the jobs have appeared before or not.

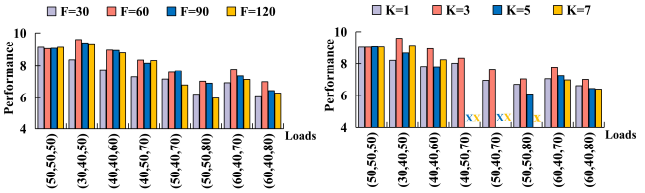
5.3.5 Effectiveness of Using Multi-Versioned Bandits

In this experiment, we examine the effectiveness of maintaining multiple versions of bandits. To this end, we run a colocation (consisting of three LC jobs and one BE job) for a long time period of 480 seconds. During the running period, we change the load of each LC job in every 40 seconds, either increasing or decreasing 10% of its load, randomly determined. We use the same load change trace to repeat the experiment for multiple times.

We first fix the number of versions of bandits that we maintain (denoted by K), and vary the frequency of constructing a new version of bandits (denoted by F). Figure 11(a) shows the performance of OLPart with varying F. We observe that OLPart basically achieves the best performance for some intermediate value of F (e.g., 60 when load is (60, 40, 70)), while gets worse performance for both small and large F. This is reasonable because constructing new versions of bandits too often cannot guarantee the bandit parameters of each version are well-learnt, while a too long time interval between two versions would not be timely enough for adapting the load

changes.

Then, we fix F at 60 and varies K among 1, 3, 5 and 7. Figure 11(b) shows the results. Similarly, OLPart achieves the best performance for some intermediate value of K (e.g., 3 when load is (40, 40, 60)). The reason is that a larger K would cause too many old versions of bandits, leading to sub-optimal decision for the current load; a smaller K leads to a higher probability of using more recent versions of bandits, but would increase the risk of generating bad decisions since new versions of bandits lack history knowledge. We also observe that the performance of OLPart for $K = 3$ is always better than that when $K = 1$, which demonstrates the effectiveness of using multiple versions of bandits.



(a) (b)

Figure 11: (a) Performance of OLPart for different F when $K = 3$; (b) Performance of OLPart for different K when $F = 60$.

5.4 Overhead

As has been analyzed in Section 4.1, there are in total of $|J|\sum_{r\in R} N_r$ arms maintained by OLPart for each version of bandits, where N_r refers to the number of allocation schemes for resource r. Estimating the reward of each arm incurs a complexity of $O(d^3)$ (matrix inversion usually can be solved in $O(d^3)$), where d is the dimension of contextual feature. It is easy to analyze that the complexity of the arm selection algorithm is $O(L|J|\sum_{r\in R} N_r)$ (the algorithm maintains top L arms with the highest estimated reward). Therefore, the overall complexity of OLPart for making a partitioning decision is $O(L|J|\sum_{r\in R} N_r + d^3|J|\sum_{r\in R} N_r)$. Note that the values of the parameters are very small, implying that OLPart is computationally efficient.

Table 5 summarizes the time spent by OLPart for making a partitioning decision for different colocation sizes. It can be seen that OLPart can always make decisions within 180 ms, and the time spent only increases a little bit as the colocation size grows. The results confirm that OLPart is very efficient with respect to computational overhead.

Table 5: Running time of OLPart over different colocation sizes.

colocation size	3	4	5	6	7
Time	55ms	110ms	130ms	150ms	180ms

6 Related Work

Workload consolidation has been extensively studied in the literature for improving resource utilization. Earlier works have used conservative solutions without considering resource partitioning [9, 13, 29, 30, 33, 40, 42] . They usually analyze the resource sensitivities of jobs through offline profiling, and then colocate only the jobs that do not contend for the same resources for guaranteeing the QoS. However, such solutions require priori knowledge of jobs and suffer from inefficiency because the types and number of jobs that can be colocated are very limited.

早期

Recent works have adopted more aggressive workload consolidation solutions which partition the resources among colocated jobs. Earlier resource partitioning solutions [8,34,35,37] usually establish dedicated analytical performance models to guide the search for an optimal partitioning configuration. However, such solutions typically rely on extensive domain knowledge, which can only handle one resource. The problem of partitioning multiple resources simultaneously also have been studied [8, 14]. However, they are towards consolidating BE jobs for optimizing fairness or system throughput, which thus cannot be applied to colocating LC jobs who have strict QoS requirement. There are also many resource works focused on colocating LC job with BE jobs [7, 19–21, 27, 33, 45]. However, they mainly focus on partitioning to satisfy QoS for only one LC job, and do not improve the performance of the BE jobs.

近期

The most relevant works to OLPart are PARTIES and CLITE, both of which aim to partition multiple resources among colocated LC jobs and BE jobs, for maximizing the performance of the BE jobs with the QoS guarantee for multiple LC jobs. However, PARTIES adjusts one resource with small steps at a time, which significantly limits its efficiency. As we have shown in Section 5.3.1, PARTIES cannot find a feasible solution in many cases. CLITE uses a more intelligent approach which builds approximate performance model to guide a more intelligent exploration of the search space. However, CLITE is not able to perceive the resource sensitivities of jobs, which significantly degrades the exploring efficiency. Moreover, CLITE cannot handle large coloation sizes due to the high computational complexity of BO.

7 Conclusion

In this paper, we propose an online resource partitioning solution, named OLPart, for colocating multiple LC jobs with BE jobs. OLPart leverages CMAB to employ runtime performance counters to enable an intelligent exploration of the search space. Evaluation results show that OLPart outperforms state-of-the-art baselines significantly. In the future, we would like to integrate OLPart into real systems.

References

[1] Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. *Advances in neural information processing systems*, 24:2312–2320, 2011.

[2] H. Andrew, Khawar M Abbasi, and C. Marcel. Introduction to memory bandwidth allocation. https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation, 2019.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[4] Donald A Berry and Bert Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5(71-87):7–7, 1985.

[5] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, volume 2011, page 37, 2009.

[6] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. Multi-objective job placement in clusters. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[7] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 272–283, 2019.

[8] Ruobing Chen, Jinping Wu, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. Drlpart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 175–188, 2020.

[9] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[10] Yifang Chen, Alex Cuellar, Haipeng Luo, Jignesh Modi, Heramb Nemlekar, and Stefanos Nikolaidis. Fair contextual multi-armed bandits: Theory and experiments. In *Conference on Uncertainty in Artificial Intelligence*, pages 181–190. PMLR, 2020.

[11] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 208–214. JMLR Workshop and Conference Proceedings, 2011.

[12] Radu Ciucanu, Anatole Delabrouille, Pascal Lafourcade, and Marta Soare. Secure cumulative reward maximization in linear stochastic bandits. In *International Conference on Provable Security*, pages 257–277. Springer, 2020.

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

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

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

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

[17] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[18] Nicolas Gutowski, Tassadit Amghar, Olivier Camp, and Fabien Chhel. Context enhancement for linear contextual multi-armed bandits. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1048–1055. IEEE, 2018.

[19] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 519–532, 2018.

[20] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610. IEEE, 2015.

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

[22] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[23] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[24] John Langford and Tong Zhang. The epoch-greedy algorithm for contextual multi-armed bandits. *Advances in neural information processing systems*, 20(1):96–1, 2007.

[25] Daniel James Lizotte. *Practical bayesian optimization*. University of Alberta, 2008.

[26] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312. IEEE, 2014.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[28] Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. JMLR Workshop and Conference Proceedings, 2010.

[29] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[30] Amirhossein Mirhosseini and Thomas F Wenisch. The queuing-first approach for tail management of interactive services. *IEEE Micro*, 39(4):55–64, 2019.

[31] Khang T Nguyen. 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/, 2019.

[32] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[33] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[34] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 57–68, 2011.

[35] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[36] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[37] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[38] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132. IEEE, 2017.

[39] Carole-Jean Wu and Margaret Martonosi. A comparison of capacity management schemes for shared cmp caches. In *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, volume 15, pages 50–52. Citeseer, 2008.

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

[41] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 58–68, 2019.

[42] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 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*, pages 406–418, 2014.

[43] Li Zhou. A survey on contextual multi-armed bandits. *arXiv preprint arXiv:1508.03326*, 2015.

[44] Haishan Zhu and Mattan Erez. 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*, pages 33–47, 2016.

[45] Haishan Zhu and Mattan Erez. 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*, pages 33–47, 2016.