Not All Resources are Visible: Exploiting Fragmented Shadow Resources in Shared-State Scheduler Architecture

Xinkai Wang[1], Hao He[1], Yuancheng Li[1], Chao Li[1], Xiaofeng Hou[1], Jing Wang[1], Quan Chen[1], Jingwen Leng[1], Minyi Guo[1], Leibo Wang[2]
[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University [2] Huawei Cloud
{unbreakablewxk, he-hao, lyc1535180405}@sjtu.edu.cn, {lichao, hou-xf}@cs.sjtu.edu.cn, jing618@sjtu.edu.cn, {chen-quan, leng-jw, guo-my}@cs.sjtu.edu.cn, wangleibo1@huawei.com

ABSTRACT

With the rapid development of cloud computing, the increasing scale of clusters and task parallelism put forward higher requirements on the scheduling capability at scale. To this end, the shared-state scheduler architecture has emerged as the popular solution for large-scale scheduling due to its high scalability and utilization. In such an architecture, a central resource state view periodically updates the global cluster status to distributed schedulers for parallel scheduling. However, the schedulers obtain broader resource views at the cost of intermittently stale states, rendering resources released invisible to schedulers until the next view update. These fleeting resource fragments are referred to as shadow resources in this paper. Current shared-state solutions overlook or fail to systematically utilize the shadow resources, leaving a void in fully exploiting these invisible resources.

In this paper, we present a thorough analysis of shadow resources through theoretic modeling and extensive experiments. In order to systematically utilize these resources, we propose Resource Miner (RMiner), a hybrid scheduling sub-system on top of the shared-state scheduler architecture. RMiner comprises three cooperative components: a shadow resource manager that efficiently manages shadow resources, an RM filter that selects suitable tasks as RM tasks, and an RM scheduler that allocates shadow resources to RM tasks. In total, our design enhances the visibility of shared-state scheduling solutions by adding manageability to invisible resources. Through extensive trace-driven evaluation, we show that RMiner greatly outperforms current shared-state schedulers in terms of resource utilization, task throughput, and job wait time with only minor costs of scheduling conflicts and overhead.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624650

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Shared-state Scheduler, Shadow Resource, Cloud Computing

ACM Reference Format:
Xinkai Wang, Hao He, Yuancheng Li, Chao Li, Xiaofeng Hou, Jing Wang, Quan Chen, Jingwen Leng, Minyi Guo, Leibo Wang. 2023. Not All Resources are Visible: Exploiting Fragmented Shadow Resources in Shared-State Scheduler Architecture. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624650

1 INTRODUCTION

With the rapid growth of the cloud computing market in recent years, clusters are expanding in scale, with some containing thousands of thousands of machines and being deployed on various cloud services, including virtual machines, containers, microservices, and function-as-a-service platforms. Moreover, the million-level concurrent submission and execution of second-level even millisecond-level tasks leads to a rising demand for low-overhead, high-utilization, and high-scalable scheduling architecture. Considering the value and challenges in large-scale scheduling, many IT companies such as Google [36], Microsoft [7], and Alibaba [47] have invested significant amounts of capital and engineering power in developing such systems.

Given the low scalability of monolithic scheduling architecture [25, 47] and low utilization of two-level schedulers [22], the shared-state scheduling architecture has become the widely-used solution for large-scale cluster scheduling due to its high scalability and capability of parallel

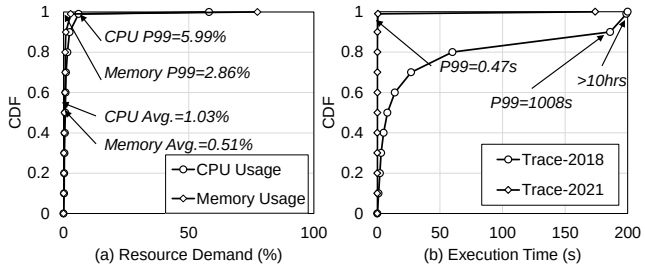


Figure 1: Distributions of resource demand and execution time (From (a) Google [37] and (b) Alibaba [4])

scheduling [32]. In such an architecture, a central resource state view periodically updates the global cluster status to distributed schedulers and the schedulers make allocation decisions in parallel for incoming tasks. Based on the original shared-state design, major cloud providers have proposed and implemented respective shared-state scheduling systems to achieve lower scheduling delay [29], lower conflicts [17], higher throughput [28], and higher scalability [7].

However, each coin has two sides, and the shared-state architecture also has its shortcomings. Apart from the commonly studied scheduling conflicts and scheduling delay issues, periodic global state update design results in intermittently stale states of distributed schedulers, which is rarely mentioned and studied before. Resources released are only visible to the central state view but invisible to all parallel schedulers until the next view update, which turns into fleeting resource fragments defined as *Shadow Resources*. The hidden shadow resources are beyond the scheduling scope of normal schedulers and cause a great waste to shared-state clusters. Nonetheless, prior works on shared-state architecture mainly focus on managing visible resources more efficiently via advanced scheduling policies and techniques [7, 17, 28, 29, 31] while ignoring the mining and utilization of invisible shadow resources.

To make matters worse, the spatial and temporal granularities are becoming more lightweight. On the one hand, the resource demand of tasks is also lower than traditional resource-hungry monolithic applications. On the other hand, the number of short-term tasks is increasing due to the emergence of cloud-native technology. From our statistical analysis of traces from large-scale clusters of Google Cloud [37] and Alibaba Cloud [4] in Figure 1, the resource demand and the execution time of tasks approximately fit exponential distributions, which is in line with the findings of prior works [1, 17]. Figure 1 (a) shows the CPU and memory demands of tasks in Google Cloud and the average resource demand per task is about 0.5% - 1.0%. Figure 1 (b) shows the execution time of tasks in Alibaba Cloud and the time granularity of tasks has become finer recently. Traditional long-lasting batch workloads gradually shift dominance to short-lived latency-sensitive tasks. The temporal and spatial trends lead to more frequent and fine-grained resource changes and exacerbate invisible resource wastes.

Shadow resources are precious while hard to exploit. Both theoretic and experimental analysis of shadow resources in real clusters show that the amount of shadow resources is affected by the duration between view updates (a.k.a., updating delay) and the average execution time of tasks, and accounts for 2-13% of overall allocated resources in the cluster, which is considerable and precious in improving resource utilization. Nonetheless, two obstacles hinder the utilization of shadow resources and call for flexible and transparent solutions. First, the fleeting and fragmented properties require an agile mining mechanism. Second, it is crucial to both utilize shadow resources and avoid intrusion into normal scheduling. In this work, we argue that shared-state architectures need to consider shadow resources carefully.

This paper takes the first step to enhance the visibility of shared-state architecture to support mining and utilizing shadow resources in the cluster. To this end, we propose Resource Miner (RMiner), a hybrid and back-compatible scheduling sub-system of shared-state architecture, which consists of three cooperative components: (1) shadow resource manager that efficiently detects and organizes shadow resources in the cluster; (2) RM filter that selects suitable tasks to match fleeting fragments; (3) RM scheduler that allocates shadow resources to RM tasks in an appropriate manner. Further, we explore more aggressive resource mining methods by exploiting shadow resources between view update and actual allocation (a.k.a., resource waiting delay). For different goals of cluster management, RMiner is flexible with two resource mining modes, SafeRM and SmartRM, to balance the maximized resource utilization and minimized conflicts.

To thoroughly evaluate RMiner, we conduct trace-driven experiments on the top of industrial cluster simulator [1] and open-source traces [4, 37]. We mimic the realistic shared-state scheduling process and use high-fidelity task execution traces as input. We show that RMiner outperforms conventional shared-state schedulers and brings up to 5.8% on resource utilization, 28% on overall throughput, and 59.9% on job wait time. More specifically, we can exploit up to 112% shadow resources with a minor overhead of less than 3% more conflicts and scheduling overhead. In summary, this paper makes the following contributions:

- We discover the invisible fragment resource opportunities in shared-state scheduling architecture and analyze them theoretically and experimentally.
- We introduce RMiner, a novel sub-system for shared-state architecture to enhance the spatial and temporal visibility of current designs. RMiner mines and exploits

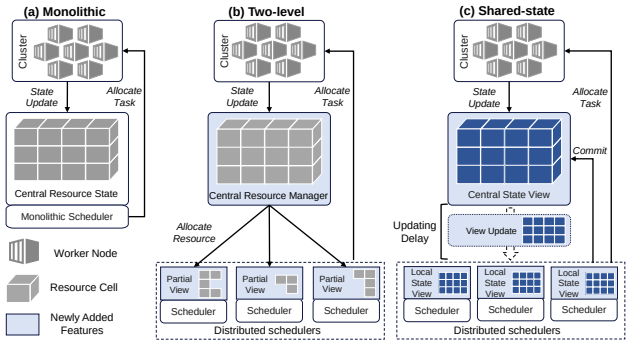


Figure 2: Three typical scheduler architectures

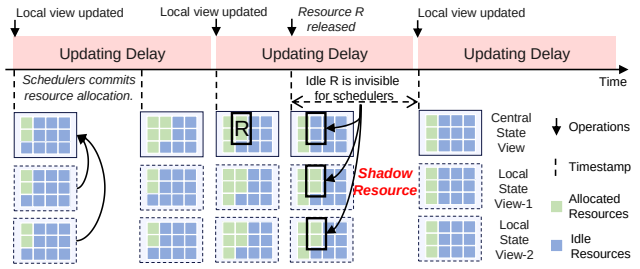


Figure 3: Shadow Resource in Shared-state Scheduler

invisible shadow resources with further enhancements for more aggressive and flexible management.

- We build and optimize an industrial cluster simulator of RMiner and show that it could improve cluster performance greatly with minor overhead.

The rest of the paper is organized as follows. Section 2 introduces the background and further motivates our work. Section 3 illustrates the detailed design details and optimization modes of RMiner. Section 4 presents the evaluation methodology and results. Section 5 discusses related work and Section 6 concludes this paper.

2 BACKGROUND AND MOTIVATIONS

In this section, we first introduce the shared-state scheduler architecture. Then we define and analyze the shadow resources within such architecture. At last, we demonstrate the challenges of mining and utilizing shadow resources.

2.1 Shared-state Scheduler Architecture

In cloud computing clusters, there are three major scheduler architectures: 1) monolithic, 2) two-level, and 3) shared-state [32]. As shown in Figure 2 (a), a monolithic scheduler is the only scheduler in the cluster with up-to-date global resource states and applies the same algorithm for all incoming jobs. This architecture is the least scalable and flexible for large-scale clusters. In a two-level scheduler as Figure 2 (b), there is a central manager and multiple distributed schedulers [22]. The manager allocates global resources to individual schedulers, and each scheduler holds the latest partial view of resources. They can only schedule tasks on their partial resources. This architecture is more scalable compared to the prior design while suffering from low utilization due to the isolated views of parallel schedulers.

Consequently, the shared-state scheduler has become the popular architecture for large-scale clusters due to high scalability and utilization [7, 17, 28, 31, 32]. It overcomes conventional schedulers by providing all schedulers with a global view of all the resources. As shown in Figure 2 (c), there are multiple parallel schedulers managed by a master with Central State View (CSV). CSV maintains up-to-date information on cluster resources and is not responsible for scheduling tasks. CSV periodically updates the Local State View (LSV) owned by each distributed scheduler with fixed updating delays. The original shared-state design updates LSVs at each successful resource allocation action [32] but in realistic clusters, the updating delay is usually second-level to reduce the update overhead [17]. When schedulers schedule tasks, they allocate resources based on their own LSV and commit the decisions to CSV to avoid conflicts with other schedulers. It is obvious that the pursuit of low overhead makes the LSV of each parallel scheduler intermittently stale since they are invisible to the up-to-date status of released resources within the updating delays. Therefore, we define these invisible resource fragments as shadow resources in this paper.

2.2 Shadow Resources in Shared-State Scheduler Architecture

In the shared-state scheduler architecture, the resource states of distributed schedulers are stale within updating delays and it results in shadow resources. As shown in Figure 3, at the start of each updating delay, the CSV updates the local resource state of each LSV owned by distributed schedulers. The distributed schedulers commit resource allocation decisions to CSV for task scheduling and the latest cluster status is known by only the CSV. Within the updating delay, LSVs owned by each scheduler fall behind the actual cluster status but they have a relatively global view of cluster resources. Similarly, when certain resources (R) are released, the CSV updates itself immediately while distributed schedulers are still relying on stale local state views. Before the next time of local view update, each parallel scheduler would treat the idle resource R as allocated and cannot schedule new tasks to R, leading to a great waste of cluster resources. Therefore, shadow resources are those that are not visible to the distributed schedulers in their resource view when they can actually be used for allocation.

Table 1: Variables for modeling analysis

Variables	Descriptions
d_u	Duration of updating delay
r_{run}	Total allocated resources
T_s	Survival time of shadow resources
R_t	Resource demand of tasks
λ	Parameter of R_t distribution
T_t	Running time of tasks
η	Parameter of T_t distribution
N	Number of tasks running in the cluster
UT	Idealized unit of time
K	Number of tasks completed per UT
X	Instantaneous amount of shadow resources

Given the existence of precious shadow resources within shared-state scheduler architecture, there are two main differences compared with normal resources. Firstly, shadow resources are invisible to current distributed schedulers. Prior works on shared-state schedulers focus on achieving higher utilization and lower conflicts on normal resources [7, 11, 17, 28, 31, 32]. Most of works ignore the existence of shadow resources and their solutions fail to utilize them. ParSync noticed stale local state views while it partially added visibility to schedulers and focused on resolving conflicts [17]. Secondly, shadow resources are fleeting and fragmented. The granularity of shadow resources is strongly related to updating delay, which is about second-scale [17]. Further, it is not practical to alleviate shadow resources by reducing the invisibility window of shared-state architecture, because shorter invisibility windows incur higher synchronization overhead to the overall scheduling system [17]. Meantime, shadow resource fragments spread in distributed servers due to lightweight cloud task granularities. Therefore, the mining of shadow resources is valuable but challenging to current shared-state scheduling systems.

2.3 Analysis of Shadow Resources

Till now, we still have no clear understanding of them in the cluster as a whole. Therefore, we both theoretically model and experimentally analyze the quantitative amount of shadow resources in the cluster to answer the following three questions: (1) What is the total amount of shadow resources available in the cluster? (2) What are the factors affecting the amount of shadow resources? (3) How do these factors affect the amount of shadow resources?

To theoretically analyze shadow resources, we first define some constants and variables as Table 1, where upper-cases are variables and lower-cases are constants. As Figure 3, resource R is released and becomes a shadow resource, which survives until the local state view is updated. Without loss of generality, we assume that the cluster is stable with r_{run} resource allocated, and within each updating delay d_u, the resource release rate remains the same. Therefore, the survival time T_s follows a uniform distribution, i.e., $T_s \sim U(0, d_u)$.

Further, we analyze the open-source industrial traces [1, 37] and related works [17, 36] to conclude that the statistical resource demand and execution time of tasks at scale approximately fit continuous exponential distributions. Therefore, the task resource demand fits $R_t \sim EXP(\lambda)$ and the task execution time fits $T_t \sim EXP(\eta)$. λ and η are approximately the average resource demand and average execution time of tasks. The number of tasks running in the cluster N is $\frac{r_{run}}{\overline{R_t}}$, and the number of tasks completed per unit time UT is:

$$K = N/\frac{\overline{T_t}}{UT} = \frac{r_{run}}{\overline{R_t}} \cdot \frac{UT}{\overline{T_t}} \tag{1}$$

Converting the continuous analysis into realistic discrete form, we can get the number of sampled tasks running in the cluster $n = \frac{r_{run}}{\lambda}$. Also, we can transform the continuous exponential variable R_t and T_t into their corresponding discrete Γ distributions. For brevity, let $Y = \frac{1}{R_t}$ and $Z = \frac{1}{T_t}$ and we can get the expectation of tasks completed per unit time:

$$\begin{aligned}
\mathbf{E}(K) =& r_{run} \cdot UT \cdot E(Y) \cdot E(Z) \qquad (2)\\
=& r_{run} \cdot UT \cdot \int_0^\infty \frac{n}{\lambda\Gamma(n)} y^{-n} e^{-\frac{1}{y}} dy \cdot \int_0^\infty \frac{n}{\eta\Gamma(n)} z^{-n} e^{-\frac{1}{z}} dz\\
=& r_{run} \cdot UT \cdot \frac{n\Gamma(n-1)}{\lambda\Gamma(n)} \cdot \frac{n\Gamma(n-1)}{\eta\Gamma(n)}\\
=& \frac{UT \cdot r_{run}^3}{\lambda\eta \cdot (r_{run} - \lambda)^2} \qquad (3)
\end{aligned}$$

Further, the total amount of shadow resources can be calculated by multiplying three elements: (1) tasks completed per unit of time K, (2) the amount of each shadow resource fragment, and (3) the corresponding survival time of each shadow resource T_s. Given that shadow resources are the released resources when corresponding tasks are complete, we assume that the amount of each shadow resource is equal to the amount of resource demand of each task, i.e., R_t. And T_s follows uniform distribution according to the analysis above. Then we divide the result by the unit of time UT to get the instantaneous amount of shadow resources X in the cluster.

$$\begin{aligned}
\mathbf{E}(X) =& \frac{1}{UT} \cdot E(R_t) \cdot E(T_s) \cdot E(K) \qquad (4)\\
=& \frac{1}{UT} \cdot \int_0^\infty \frac{r}{\lambda} e^{-\frac{r}{\lambda}} dr \cdot \int_0^{d_u} \frac{t}{d_u} dt \cdot \frac{UT \cdot r_{run}^3}{\lambda\eta \cdot (r_{run} - \lambda)^2}\\
=& \frac{d_u r_{run}^3}{2\eta \cdot (r_{run} - \lambda)^2} \qquad (5)
\end{aligned}$$

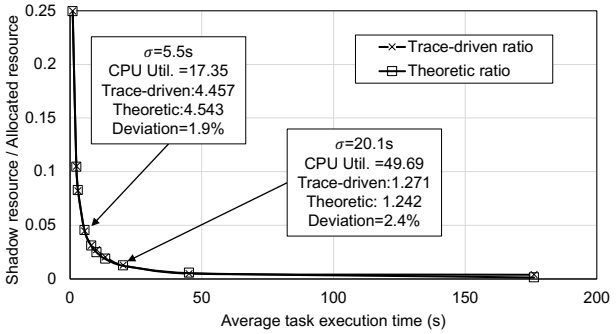


Figure 4: Comparison of the theoretic and trace-driven ratio of shadow resources

Given that the average resource demand for tasks λ is negligible compared to the total amount of allocated resources in the cluster r_{run}, the above equation can be approximated by $r_{run} - \lambda \approx r_{run}$. Therefore, the final expectation of the instantaneous amount of shadow resources is:

$$\mathrm{E}(X) = \frac{d_u \cdot r_{run}}{2\eta} \tag{6}$$

We show that the total amount of shadow resources is mainly proportional to the duration of updating delay d_u and the number of allocated resources in the cluster r_{run}, while inversely proportional to the average execution time of tasks η. The analysis above could roughly answer the last two questions about the factors affecting the amount of shadow resources. Counter-intuitively, shadow resources have little relationship with the resource demand of tasks because the granularity of execution time and resource demand is getting smaller at the same time.

To answer the first question of the total amount of shadow resources available in the cluster, we rely on trust-worthy statistics to calculate the approximate ratio of shadow resources in the cluster with $E(X)/r_{run} = d_u/2\eta$. According to industrial traces [37] and experiences [17, 36], we set the average task execution time η as $4s \sim 5s$ and the duration of updating delay d_u as $0.3s \sim 1.0s$. Then we find that shadow resources in the cluster account for about $3\% \sim 12.5\%$, which is considerable and valuable for cluster scale. With the development of lightweight cloud-native technologies, invisible resource waste becomes more and more severe.

To validate our analysis, we observe and record the shadow resources in realistic shared-state scheduling process [1] (detailed in Section 4.1). Since the ratio is strongly dependent on the trace selection, we randomly sample industrial traces [4] for 10 separate settings and report the results in Figure 4. It is obvious that the experimental trace-driven ratios of shadow resources are almost the same as the results of theoretic analysis, while the deviation becomes larger when the average task execution time increases due to the greater impact of monolithic and resource-hungry tasks.

2.4 Challenges Faced by Shadow Resources

Challenge-1: How to mine and manage shadow resources agilely and efficiently.

Given the fleeting and fragmented shadow resources, it is crucial to record and gather them for allocation quickly. Time-consuming mining strategies fail to seize such opportunities. Meantime, the fragmentation is worse with the emergence of light-weighted containerization tasks, like microservices [43] and serverless functions [46], which reduces the granularity of shadow resources but increases the quantity of them. This adds to the difficulty of management and calls for more efficient shadow resource usage designs.

Challenge-2: How to allocate and utilize shadow resources flexibly and transparently.

Even if we could manage shadow resources as wished, it is more complex to exploit them perfectly. Due to the fleeting and fragmented nature, we should carefully select suitable tasks to fulfill shadow resources as much as possible. More importantly, exploiting shadow resources cannot cause negative effects on the performance and efficiency of normal shared-state schedulers. So there is a trade-off between maximized utilization and minimized conflict. We pursue a solution that is transparent to the normal scheduling process and makes as little intrusion as possible.

Overall, we envision the hidden resource opportunities due to the limited visibility of the current shared-state design and perform a comprehensive analysis of the importance and challenges to fill the void. Next, we present our design to properly and effectively enhance current systems.

3 DESIGN OF RESOURCE MINER

Based on the characteristics and challenges of shadow resources in the current shared-state scheduler architecture. In this section, we present the detailed design, **Resource Miner** (RMiner), to mine and exploit shadow resources for a high-performance and high-visibility scheduling system.

3.1 Overview and Design Principles

RMiner is built upon current shared-state architecture (the white parts) and contains three cooperated components (the blue parts) as shown in Figure 5: (1) *Shadow Resource Manager* detects and manages up-to-date shadow resource state efficiently with a newly-designed index, (2) *RM Filter* select tasks suitable for shadow resource (RM Tasks) to the task queue, (3) *RM Scheduler* is responsible for allocating shadow resources to RM tasks flexibly. Based on the characteristics

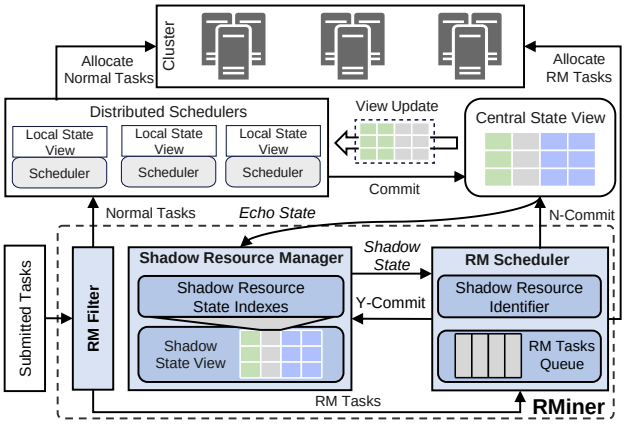


Figure 5: Overview of Resource Miner

of shadow resources and the shared-state architecture, we derive two design principles for RMiner:

Principle 1: Intrusion Avoidance

To enable transparent integration with current shared-state architectures, we should avoid intrusive modification to the original scheduling system. Preferably, the original schedulers are unaware of the usage of shadow resources and incur no conflicts with the allocation of shadow resources.

Principle 2: Balanced Performance

RMiner faces the trade-off of maximized utilization of invisible resources and minimized conflict with visible scheduling. Aggressively utilizing shadow resources can lead to abundant preemptive executions, which in turn degrade the overall performance of clusters, and vice versa. Therefore, our design should balance the performance of RMiner and the original system and pursue the optimal performance.

3.2 Shadow Resource Manager

The *Shadow Resource Manager* is responsible for detecting the generation of shadow resources, organizing their state indexes, and managing them for further scheduling. To perform these functionalities efficiently, we newly design a data structure, **shadow resource state indexes**, to manage shadow state view, as shown in Figure 5.

The shadow resource state index is a 6-tuple including three basic dimensions and three scheduling-related dimensions. In basic dimensions, *shadow resource ID* distinguishes different fragments R, *survival time* denotes the interval till the next update of all local state views, and *machine ID* maps R to a specific node in the cluster. The other three dimensions are related to RM scheduling logic, where *available resource* is set as the amount of R while *occupied resource* and *allocated tasks* are set as 0. We merge the shadow resources on each node to reduce the managing complexity.

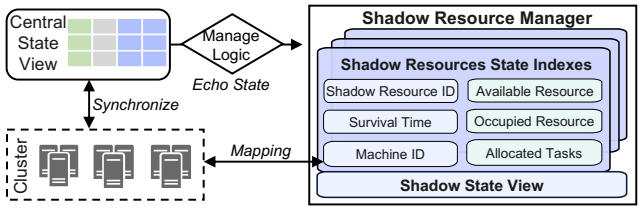


Figure 6: Workflow of shadow resources state indexes

Algorithm 1: Management of shadow state view

Input: Released shadow resource R
Output: Updated shadow state view with R_s

1 **Echo State Function:**
2 **if** *RM task released R* **then**
3 **if** *Original* R_s *exists* **then**
4 $R_s \leftarrow R_s + R$;
5 **end**
6 **else**
7 $R_s \leftarrow R,\ T_s \leftarrow t(R_s), M_s \leftarrow m(R_s)$;
8 Add a new shadow resource state index (R_s, T_s, M_s);
9 **end**
10 **View Update Function:**
11 **for** R_s *in shadow state view* **do**
12 **if** *Occupied* $R_s = 0$ **then**
13 Remove R_s index from view.
14 **end**
15 **end**

As Figure 6, when the central state view synchronizes the status of the cluster and monitors the release of resource R, the shadow state view is aware of this information immediately through *Echo State* mechanism and merges the newly discovered shadow resource into the shadow state view through the manage logic outlined in Algorithm 1. In the echo state function, we first identify the task type releasing R according to CSV (line 1). If it is the RM task releasing R, we merge it to its origin R_s if possible (line 4). We can use hash mapping to reduce the overhead of managing operations. If normal tasks release R, we use the direct $t()$ and $m()$ functions to get the corresponding survival time and machine ID and add a new shadow resource state index (lines 7-8). Moreover, in the view update function, the shadow resource manager would remove all the idle shadow resources at view update by CSV (lines 11-14) and continue to manage the occupied shadow resources until the release moment.

To follow the design principle of intrusion avoidance, scheduling RM tasks to the shadow resources should not commit to the central state view like the normal schedulers.

Otherwise, after the local views are updated, normal schedulers will see that the shadow resource is occupied, which affects the total resource of normal scheduling and is no longer invisible. As we expect the original scheduling system is not aware of the shadow resources, we design the **shadow state view** for the management of RM tasks' scheduling. With the state echo, the shadow state view is kept up to date with the central state view, and the state echoes also include state updates of the shadow resource. Therefore, the shadow state view includes both the state information in the central state view and of shadow resources, ensuring that no conflicting errors are generated in resource mining scheduling. In addition, the state update requests include the update information about the shadow resource state, while these kinds of state update requests will be ignored by the central state view in the RMiner to avoid the impact on the original scheduling system, which also follows the principle of intrusion avoidance. These state updates will still be sent to the manager with the state echoes to maintain the shadow resource states and update the shadow state view.

3.3 RM Filter

RM filter is responsible for selecting the tasks suitable for shadow resources and constructing the task queue of the RM scheduler. We filter the appropriate tasks in advance instead of stealing from the task queues of geo-distributed schedulers, since the stealing operation may span around and hinder the usage of fleeting shadow resources. In our design, the RM filter follows three principles.

Firstly, filtered tasks should match the fleeting and fragmented properties of shadow resources. This principle is straightforward and places restraints on both the resource demand R_t and execution time T_t of tasks. Fortunately, the emerging trend of lightweight containerization tasks, like microservices and serverless, enables the RM filter to select sufficient tasks matching the granularity of shadow resources. Secondly, filtered tasks should be preemptive to be killed or migrated. To avoid intrusion, we prioritize the execution of normal tasks over RM tasks and design two eviction modes to control the usage of shadow resources (detailed in Section 3.5). Selecting tasks with lower priority would both utilize shadow resources and cause less violation.

Thirdly, neither the RM filter selects too many tasks for the RM scheduler since it may become a new performance bottleneck, nor could it select too few tasks to fully utilize shadow resources. The filter threshold is of great importance and here we consider five key factors to adjust the threshold: (1) the length of the RM tasks queue that denotes the workload of the RM scheduler; (2) the current amount of shadow resources from the shadow resource manager; (3) current task submission rate denoting the workload of the cluster; (4)

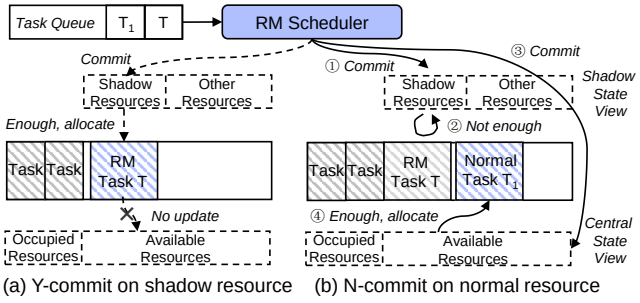


Figure 7: Execution flow of RM scheduler

the scheduling success rate of RM tasks; (5) current updating delay of the scheduling system. The threshold is relatively proportional to factors 2-5 and inversely proportional to factor 1 and RMiner uses it to filter low-priority tasks with proper resource demand and execution time. Obviously, the threshold is qualitative and we can enhance the threshold calculation function with more complex RL-based methods and more comprehensive factors, which is not our focus in this paper. In summary, the RM filter utilizes the real-time and historical status of clusters to filter just the right tasks for the RM scheduler to allocate shadow resources.

3.4 RM Scheduler

RM scheduler is responsible for allocating tasks from the RM filter to utilize shadow resources. As shown in Figure 7, the task queue contains tasks relatively suitable for shadow resources from the RM filter. The RM scheduler interacts with the shadow state view and central state view in different situations and it has two task allocation paths. On the one hand, the dotted line works when the shadow resource is enough for the task. It commits the resource allocation decision to the shadow state view to decide whether the action has enough shadow resources and conflicts. If the commit is successful, corresponding shadow resources will be allocated to RM task T. It is worth mentioning that this process is transparent to the central state view, that is, the allocation of shadow resources is not updated to CSV and CSV will set the whole shadow resources as available at next local view update. Only by doing so can we follow the principle of intrusion avoidance and avoid the available resources being mostly occupied by RM tasks in the next updating delay, which harms the performance of all distributed schedulers.

On the other hand, the solid line works when there are not enough suitable shadow resources for the task, the RM scheduler degrades to a normal parallel scheduler and allocates the task to other available resources. It (①) commits the resource allocation decision to shadow state view and (②) receives a signal of not enough shadow resources. Then it (③) commits the allocation to CSV like parallel schedulers

Algorithm 2: Scheduling algorithm of RM scheduler

Input: Task T, Resource demand R_T
Output: Updated shadow state views R_s

```
1  RM task Function:
2  Traverse nodes w/ shadow resources in descending
     order;
3  if Node N satisfy R_T then
4  |  Allocate T to N, R_s^N = R_s^N − R_T
5  end
6  Reorder nodes list;
7  Normal task Function:
8  if All nodes are not satisfied for shadow resources then
9  |  Traverse nodes w/ available resources;
10 |  if Node N satisfy R_T then
11 |  |  Allocate T to N, R_s^N ← 0;
12 |  end
13 end
```

act and (④) if the commit is successful, task T_1 is allocated to available resources. Tasks in this path are no longer marked as RM tasks and they will be treated as normal tasks when they incur shadow state view update in Algorithm 1. Strictly, not all tasks that are filtered to the RM scheduler are RM tasks, but only tasks that are allocated to shadow resources.

The execution of the RM scheduler is based on two resource state views, and the interaction is detailed in Algorithm 2. For RM task scheduling, we sort the nodes based on their shadow resources in descending order and traverse the nodes (line 2). When node N is satisfied after committing, we allocate task T to node N and update the shadow resource view (line 4). For normal task scheduling when all nodes are not satisfied for shadow resources, we use the default resource allocation function in current shared-state designs [1] and find node N to allocate task T (line 9). To give priority to shadow resource utilization, we first allocate all shadow resources on node N to task T, and the residual demand of task T is other resources. Thus we set the shadow resource on node N as none and update the view (line 11).

Another critical issue is the conflict between RM tasks and new scheduling tasks due to the invisibility of the central state view into shadow resource allocation. When CSV updates the local view, the occupied shadow resources R are marked available to distributed schedulers and they would allocate tasks to R, which definitely incurs conflict. In such cases, distributed schedulers are unaware of the conflict since the commit to CSV about resource R is successful. Therefore, we formalize and devise two optimization modes to resolve such conflicts in the following.

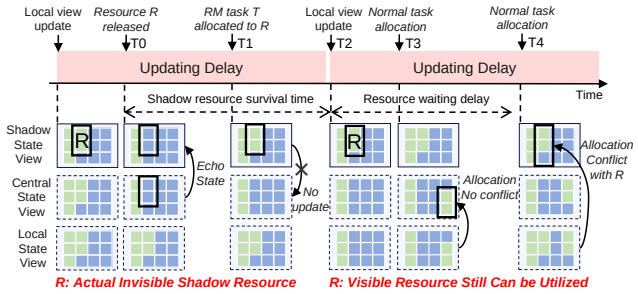


Figure 8: Resource waiting delay of shadow resources

3.5 Optimization of Resource Miner

As stated above, we seek the optimal trade-off between resource utilization and scheduling conflicts. In fact, by analyzing the state changes of shadow resources, we find that when local state views are updated, the distributed schedulers are visible to these shadow resources, but these newly observed shadow resources are not occupied immediately due to the presence of other available resources. The idle time between when the shadow resource is seen by the normal scheduler and when it is actually reallocated is called `resource waiting delay`. The resource waiting delay is the continuation of the shadow resource survival time, during which the resource is visible but can still be utilized without affecting the normal scheduling process. It is possible to optimize RMiner by exploiting the resource waiting delay.

As Figure 8, at the local view update, CSV updates the resource state of each LSV owned by distributed schedulers. At $T0$, resource R is released and becomes shadow resources invisible to LSV. CSV updates shadow state view through echo state. At $T1$, the RM scheduler commits and allocates task T to R and this allocation is not updated to CSV. At $T2$, CSV updates LSV again while ignoring the occupation of shadow resource R according to Figure 7. At $T3$, distributed schedulers commit and allocate tasks to resources in the black box without conflict with R, which means the shadow resource R can still be utilized. However, at $T4$, distributed schedulers commit and allocate tasks to resources in the black box containing R, which incurs conflict with executing RM tasks and requires further actions to eliminate the effect. Therefore, the period between $T2$ and $T4$ is the resource waiting delay and it extends the original shadow resource survival time to gain more utilization of shadow resources.

Based on the analysis above, the trade-off between higher utilization of shadow resources and lower conflicts with normal tasks is of great importance. The extended use of shadow resources in resource waiting delay is risky and we design two conservative modes of RMiner due to two reasons: (1) We cannot foresee the resource waiting delay and the timing of conflicts in advance since the task allocation is unpredictable;

Table 2: Comparison of two optimization modes

	SafeRM Mode	SmartRM Mode
Shadow resource time	U_d	$U_d + W_d$
RM filter policy	SJF	LJF
RM scheduling policy	Min Conflict	Max Utilization
Task eviction policy	Migrate - Kill	Kill - Migrate

+ U_d is shadow resource survival time and W_d is resource waiting delay.
+ SJF denotes shortest job first and LJF denotes lowest-priority job first.

(2) We follow the principle of intrusion avoidance such that the utilization of shadow resources cannot harm others' performance. Therefore, we propose two modes, *SafeRM Mode* and *SmartRM Mode*, respectively.

3.5.1 SafeRM Mode. In `SafeRM mode`, resource miner only utilizes shadow resources in shadow resource survival time instead of resource waiting delay. As shown in Table 2, in SafeRM mode, the RM filter emphasizes estimated job execution time and gives priority to jobs with the shortest time. It allows for more flexible scheduling and utilizes shadow resources with conflicts as few as possible. For SafeRM mode, we design a greedy scheduling policy to minimize conflict by mapping shorter tasks to shorter shadow resources. In the cluster, SafeRM also encounters RM tasks exceeding shadow resource survival time, which invokes task eviction. Since there is no actual conflict at the timing of the update, we first try to migrate RM tasks with the normal scheduling process, and then kill RM tasks if no successful migration. Such design can guarantee the execution of RM tasks to the greatest extent, without causing severe conflicts. Nonetheless, the purpose of SafeRM is to avoid conflict and limit aggressive resource mining. So RMiner requires a more aggressive mode that can maximize the utilization of shadow resources as much as possible.

3.5.2 SmartRM Mode. In `SmartRM mode`, the resource miner utilizes shadow resources in both the resource survival time and the resource waiting delay. As shown in Table 2, in SmartRM mode, the RM filter emphasizes task resource demand and eviction cost, and it gives priority to hungry tasks with lower priority. Moreover, RM scheduling in SmartRM aims to maximize the utilization of shadow resources and allocate RM tasks with higher resource demand to just the right shadow resources. Nonetheless, under a high task submission rate and task throughput, there is a high probability of causing conflicts. Thus RM scheduling should select shadow resources on nodes holding most idle resources to avoid conflicts with the following scheduling. To keep the intrusion avoidance principle, when conflicts happen, SmartRM kills low-priority RM tasks first to guarantee the execution of normal tasks and later tries to migrate RM tasks to the proper destination. It is obvious that SmartRM has better resource utilization than SafeRM due to the increased amount of shadow resources while definitely causes more conflicts affecting the performance of RM tasks.

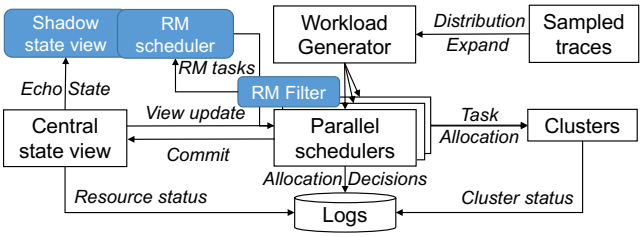


Figure 9: Modification and workflow of the industrial shared-state scheduling simulator

4 EVALUATION

In this section, we thoroughly evaluate RMiner. Specifically, we want to answer three questions:

(1) What performance improvements can RMiner bring to shared-state architecture? (Section 4.2, 4.3, 4.4)
(2) What is the cost of adopting RMiner in the current shared-state scheduler? (Section 4.5, 4.6)
(3) How do the introduced optimizations contribute to the performance improvements? (Section 4.7)

4.1 Methodology

We conduct trace-driven evaluations to thoroughly analyze RMiner on the industrial simulator. We write 2k+ LoCs of Scala to integrate the design of shadow resource manager, RM filter, and RM scheduler into the open-source Google cluster simulator [1] as Figure 9. The white components are original and the blue components are the implementation of RMiner. This simulator mimics the complete process of scheduling in large-scale clusters with three types of scheduler architectures and we mainly use the shared-state architecture of Omega [32]. It can simulate various cluster settings and various workload scenarios within a lightweight time consumption. Therefore, it is trustworthy and capable of evaluating the performance of RMiner in shared-state architecture in large-scale clusters. Without loss of generality, we mimic a cluster of 1500 homogeneous nodes with 64 CPU slots and 16 memory slots per node in the following experiments. We execute the experiments on a server with Intel (R) Xeon (R) CPU E5-2620, 32GB main memory, and Ubuntu 16.04.5 LTS installed.

4.1.1 Trace Processing. We adopt widely-used industrial cluster traces to drive the simulation [4, 37]. We adopt different processing to Google's trace and Alibaba's trace based on their information. For the original traces of Google [37],

Table 3: Processing of industrial traces

Traces	Alibaba Trace	Google Trace
Average task execution time	Sampled (4.94)	5
Average task resource demand	Sampled (1.03/64)	Sampled (0.01)
Average size of jobs	Sampled (12)	10
avgJobInterarrvialTime	1.43 (1x) - 0.7 (2x)	

we mainly follow the original method in the simulator to process the data, including generating exponential execution time, job inter-arrival time, and job size. Then we sample the resource demand of 10k jobs and expand to an input stream containing 1m jobs. For the open-source trace of Alibaba [4] containing execution time and resource demand, we sample this information of 10k jobs and expand to an input stream containing 1m jobs. Similarly, we generate exponential job inter-arrival time to mimic different task submission rates. Due to the high fidelity of the latter, we mainly use Alibaba's trace in the following experiments and use Google's trace to validate the results further.

4.1.2 Baseline Selection. In order to evaluate the improvement of RMiner over conventional shared-state scheduling architecture, we choose the typical shared-state scheduler architecture implemented in the open-source simulator, Omega [32]. The following shared-state designs have orthogonal design objectives with RMiner and have many important features (e.g., multiple scheduling algorithms and accurate task duration estimation) that are hard to reproduce in the lightweight simulator. Moreover, it is hard to create an environment in the cluster simulator for a fair comparison, if possible, with these systems. Therefore, we optimize Omega with fixed updating delay and enhanced scheduling capacities as the baseline *NoRM*. For RMiner, we use the default modes first and test various settings of SafeRM and SmartRM introduced in Section 3.5 to evaluate their performance.

4.1.3 Detailed Settings. More specifically, there are many hyper-parameters to drive the simulation experiments. The updating delay is set as 0.5s [17] and the scheduler's scheduling rate is set to 1000 tasks per second. The whole simulation execution time is set to 300 seconds. We set the number of distributed schedulers as 8 and 16. For the execution time generation of Google's trace, we fit the average time of sampled Alibaba's trace ($\eta = 4.94s$) and set the average execution time as 5s to produce exponential distribution. The size of jobs is sampled as 12 for Alibaba's trace, which means each job contains 12 tasks on average, and we set this metric of Google's trace as 10 to generate similar settings. To compare RMiner under different workload scenarios, we tune the `avgJobInterarrivalTime` from 1.43 to 0.7 to mimic different levels of pressures, where the former generates 200+ thousands of jobs in 300s to drive the experiments. For SmartRM and SafeRM, we select appropriate filter thresholds based on our design. We set the default filter threshold as updating delay and filter ratio as 50%. We further tune the thresholds and ratios to investigate the optimizations.

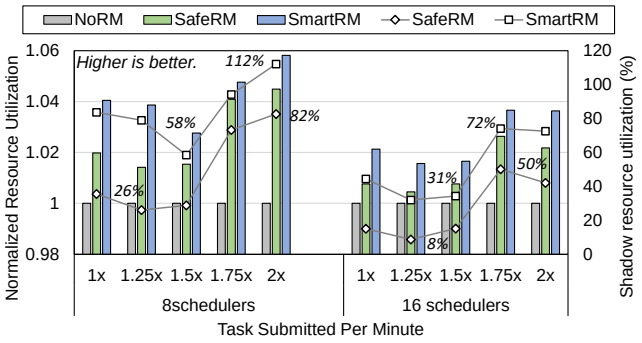
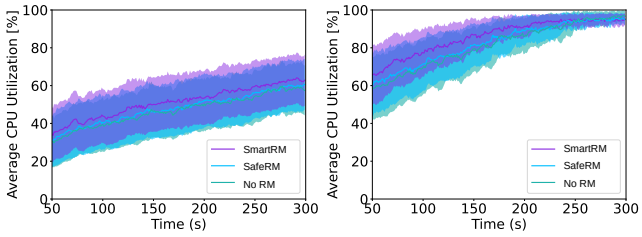


Figure 10: RMiner greatly improves resource utilization via mining shadow resources.

4.2 Resource Utilization

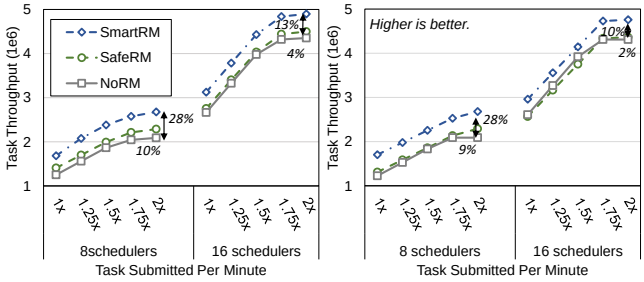
As for the first question about the performance improvements of RMiner, we record three widely used metrics to answer it. Resource utilization is one of the crucial system performance indicators of cloud computing clusters. We report the utilization-related results with Alibaba's trace in Figure 10. The bars indicate the CPU utilization improvements. Obviously, RMiner improves cluster CPU utilization via mining shadow resources. In different scenarios, shadow resources take up 1.5% to 5.0% of the cluster resource. SafeRM outperforms NoRM by 1.5% to 4% and SmartRM outperforms NoRM by 1.6% to 5.8% by utilizing resources in resource waiting delays. More specifically, RMiner works better under the 8 schedulers scenario (average utilization of 36.9%) than under 16 schedulers (average utilization of 71.8%) since fewer schedulers make it easier to find suitable RM tasks for shadow resources. Moreover, RMiner performs better under higher task submission rates (2x) since more tasks offer more released resources to be utilized.

We also report the utilization ratio of shadow resources under each setting as the marked lines. By recording the overall shadow resources and allocated shadow resources, SafeRM utilizes 26% to 82% shadow resources and SmartRM utilizes 58% to 112% shadow resources. SafeRM is more conservative by limiting tasks for shadow resource survival time only while SmartRM works more aggressively by utilizing shadow resources in resource waiting delays, which even exceeds the upper bound of invisible shadow resources. Moreover, more parallel schedulers increase the total amount of shadow resources while reducing their utilization ratio due to the reduced number of suitable RM tasks.



(a) 8 schedulers (b) 16 schedulers

Figure 11: RMiner improves the sampled CPU Utilization within the scheduling process



(a) Results on Alibaba's Trace (b) Results on Google's Trace

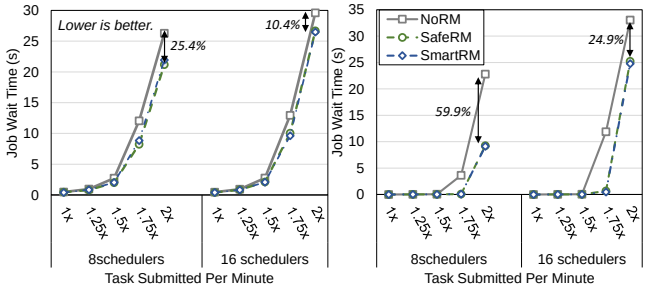
Figure 12: RMiner improves the total task throughput

Moreover, we sample the utilization of each node in the cluster within the whole scheduling process with Google's trace and report the average results in Figure 11. The line is the resource utilization and the stained part represents the standard deviation. We find that SmartRM improves utilization at all timestamps compared with SafeRM and NoRM while it has smaller fluctuations due to the aggressive scheduling policies. The average utilization under 16 schedulers is higher than that under 8 schedulers such that RMiner improves higher utilization but lower ratio compared to others. Overall, RMiner achieves considerable resource utilization improvements via the proper use of invisible resources.

4.3 Task Throughput

Besides resource utilization improvements, RMiner also improves the overall task throughput of the cluster, which is the number of scheduled tasks within the 300s period. Figure 12 reports the results on two industrial traces under various scenarios. Figure 12 (a) shows the improvements on Alibaba's trace, where SafeRM achieves up to 10% throughput improvements over NoRM and SmartRM achieves up to 28% throughput improvements. We find that under high workloads (task submission rate), RMiner performs better than lower workloads since more finished tasks produce more resource fragments, that is, shadow resources. Similarly, both SafeRM and SmartRM work better under fewer parallel schedulers (scenario of 8) like utilization comparisons. More parallel schedulers utilize more visible resources and leave less optimization space for RMiner, where SafeRM still outperforms 4% and SmartRM outperforms 13% on throughput.

More specifically, under the 8 schedulers scenario, 1.75x workload exceeds the scheduling capacity of the system and NoRM achieves the same throughput under higher workloads. But RMiner finishes more RM tasks with shadow resources and still achieves throughput improvements, which means RMiner can enlarge the scheduling capacity with enhanced resource visibility. Moreover, we compare the throughput of three schemes on Google's trace in Figure 12. It shows that SafeRM achieves 2% to 9% improvements and SmartRM achieves 10% to 28% improvements, which are similar to Alibaba's results and further validate RMiner's performance.

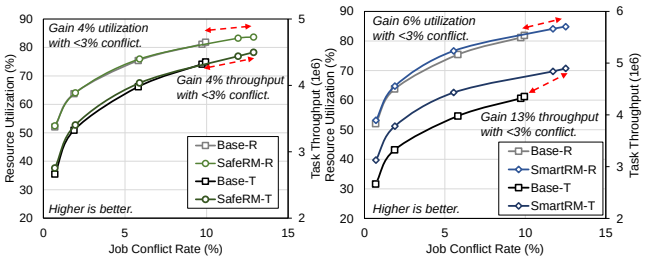


(a) Results on Alibaba's Trace (b) Results on Google's Trace

Figure 13: RMiner achieves lower job wait time

4.4 Job Wait Time

The job wait time metric reflects the waiting time between the job being submitted to the cluster and being fully scheduled [20]. It acts as an important indicator of quality of service (QoS). We present the results on job wait time in Figure 13. It shows that RMiner performs similarly to NoRM under lower workloads since tasks do not need to wait in the queue in such situations. However, under higher workloads (1.75x and 2x), more tasks are submitted in parallel, and normal schedulers have almost reached the scheduling capacity. RMiner outperforms greatly due to the utilization of more short-lived tasks to reduce the overall queuing delay. RMiner improves job wait time by up to 25.4% under 8 schedulers and up to 10.4% under 16 schedulers. More schedulers reduce the pressure of scheduling tasks concurrently but cause more scheduling conflicts. Moreover, we further validate the improvements on Google's trace and find that RMiner achieves 59.9% improvements under 8 schedulers and 24.9% improvements under 16 schedulers. Further, the



(a) 8 schedulers (b) 16 schedulers

Figure 14: RMiner trade minor scheduling conflicts for performance improvements

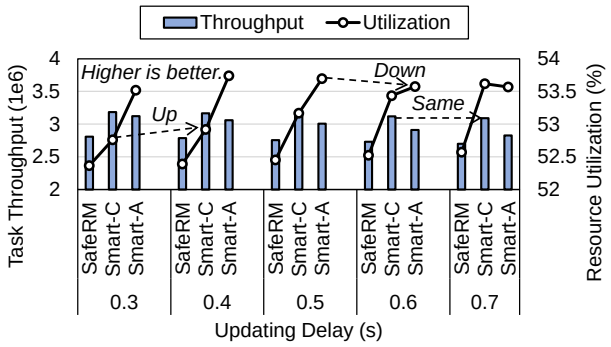


Figure 15: Comparisons on throughput and utilization of different settings of RMiner

important tail latency metric in datacenters could be inferred from job wait time. Although our evaluation environment cannot support the measure of tail latency due to the fixed task execution time, the reduction in job wait time results in better overall tail latency since the execution latency is composed of shorter job wait time and fixed execution time.

4.5 Job Conflict

In order to answer the second question about the cost of adopting RMiner in current shared-state schedulers, we first illustrate the conflict cost of RMiner. The main problem faced by shared-state scheduler architecture is the scheduling conflict of parallel schedulers and RMiner should cause as little impact on the original scheduling system as possible. More schedulers cause more scheduling conflicts and we record the conflict under different workload levels of 16 schedulers settings. Figure 14 reports the relations between performance improvements and induced conflicts. Figure 14 (a) shows the comparison between baseline and SafeRM, which shows that SafeRM causes less than 3% conflict increase in the worst case to improve resource utilization and task throughput by 4%. On average, SafeRM causes 0.5% more conflicts compared with current shared-state schedulers, which is acceptable compared to the performance earnings.

Moreover, we report the results of SmartRM in Figure 14 (b). Similarly, SmartRM causes a 3% conflict increase in the worst case for 6% resource utilization improvements and 13% throughput improvements. The average conflict cost of SmartRM is 0.73%, which is a little higher than SafeRM due to more aggressive mining strategies. We also find that conflict costs are more severe under higher workloads since more concurrent task submission makes SmartRM easier to conflict with normal schedulers. In summary, from the perspective of job conflict, the cost is negligible compared with the performance improvements.

4.6 Overhead Analysis

Besides observed conflict costs, the extra overhead of RMiner is also an important aspect to answer question 3. Unfortunately, the industrial simulator does not model overhead for scheduling so we conduct a comprehensive theoretical analysis. The overhead of RMiner contains shadow resource managing overhead and RM scheduling overhead. The shadow resource manager occupies additional memory space to store and update shadow resource state indexes, which is about 3%-12.5% to the space of CSV. The frequency of the managing algorithm is frequent under higher workloads but the complexity of the action is $O(1)$ via hash mapping, leading to acceptable computing overhead. Overall the management of shadow resources incurs negligible overhead.

As for additional scheduling overhead, current shared-state scheduler designs [11, 32] are equipped with tens of parallel distributed schedulers with global state views. RMiner adds one more RM scheduler to enhance the visibility of the current designs greatly, and the scheduling cost of the RM scheduler is lower than one conventional parallel scheduler since both the scheduling scopes and entities are smaller than before. Thus, the scheduling overhead of RMiner is roughly a single-digit increase from current shared-state designs. Overall, the costs of adopting RMiner in current shared-state schedulers are negligible compared to the considerable cluster performance improvements

4.7 Comparison of RM Modes

To answer the third question about the detailed optimizations of RMiner, we conduct extensive experiments by fine-tuning the parameters of RMiner. SafeRM remains the filter threshold of updating delay to guarantee minimized conflict. The default threshold of SmartRM is updating delay as well and we tune the filter threshold to 2x updating delay and 4x updating delay to compare the performance. The former has a lower likelihood of conflict and is defined as conservative

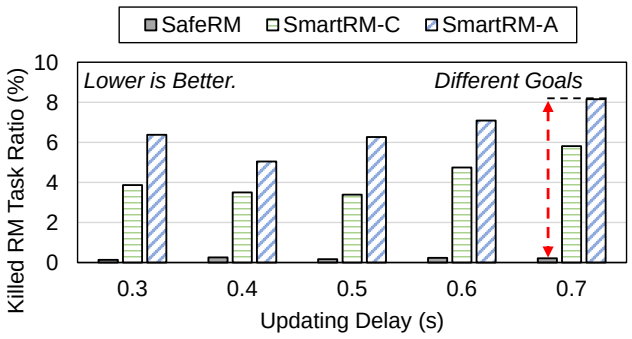


Figure 16: Scheduling conflicts comparison of different settings of RMiner

SmartRM (SmartRM-C). Conversely, the latter is defined as aggressive SmartRM (SmartRM-A). Moreover, we vary the default updating delay of 0.5s to investigate the impact on results. The experiments are conducted under 1x workload level and 16 parallel schedulers.

We firstly report the performance comparisons in Figure 15. Counter-intuitively, filtering more tasks to RMiner (SmartRM-A) increases resource utilization while decreasing task throughput. It is because this setting leaves fewer short-lived tasks to normal schedulers, which tend to schedule heavy-weight tasks to the cluster, which occupies more resources concurrently while finishing fewer tasks in total. Therefore, we need to control the filter principles carefully to avoid both over-filter and under-filter tasks in RMiner. The updating delay influences the performance of RMiner as well. The larger the updating delay is, the more the resource waste is, leading to lower task throughput and resource utilization. Under higher updating delay, SmartRM-A performs worse on utilization than in lower situations since the improvements of RMiner are masked by the degradation of normal schedulers, but it performs almost the same on utilization since more heavy-weight tasks are executed under such scenarios.

Further, we report the details of conflicts in Figure 16. We record the killed tasks due to conflict and normalize the number to RM tasks. The lower value of this metric means fewer RM tasks cause conflict with normal scheduling. It is obvious that SafeRM rarely causes conflicts with normal scheduling. As for SmartRM, the higher the filter threshold is, the higher the killed RM task ratio is, leading to more conflict with parallel schedulers. There is a trade-off between performance improvements and conflict costs and the two sides of the trade-off represent different design objectives of RMiner: highest performance improvements or lowest intrusion into the normal system. Overall, different RMiners achieve considerable performance improvements with acceptable costs and they can be flexibly configured for different goals.

5 RELATED WORK AND DISCUSSION

In this section, we summarize the advances in scheduler architectures and policies. Then we demonstrate the differences and uniqueness of shadow resources and our designs.

5.1 Scheduler Architectures and Policies

As stated in Section 2.1, there are three major scheduler architectures in current data centers. The monolithic schedulers work in high-performance computing [2, 3] and large-scale cluster computing [20, 40, 47]. The only scheduler, like Quasar[13] and Paragon[12], runs in a monolithic machine and processes all jobs with the same logic and the scheduler is obviously the bottleneck of scheduling. However, the visibility of such architecture is global and up-to-date due to the one and only design. The two-level architecture contains a central resource manager and multiple distributed schedulers [22, 39, 45]. The manager statically/dynamically partitions global resources to different schedulers and each scheduler could implement application-specific policies. This design is more scalable while suffering from low utilization due to partial view [32]. More recently, the shared-state scheduler architecture was proposed by Google [32] in 2013. It inherits the application-specific scheduling advantage of the two-level architecture and solves its problem of partial visibility of resources [7, 17, 31]. This work focuses on the stale shortcomings of shared-state architecture, which few works have studied before.

Apart from scheduler architecture designs, many scheduling policies have been proposed for different architectures. These works are variants of traditional scheduling algorithms [15, 16, 33] and focus on various goals like fairness [19, 26, 45], resource utilization [8, 23, 24, 27], and job completion time [21, 34, 38]. There are also some works targeting emerging resources like far memory [5, 41, 42] and opportunistic resources [35], which improve scheduling efficiency from remote and intermittent perspectives. Moreover, some works improve resource utilization by predicting dynamic utilization in real time [6, 9, 30], which is difficult in long-term and large-scale. Also, a bunch of works harvest idle resources from executing services to improve resource utilization [18, 44]. Our work is orthogonal and complementary to these works focusing on utilizing shadow resources within shared-state schedulers. Our scheduling resources are invisible to the above methods while the scheduling algorithms of RMiner can be enhanced with state-of-the-art complex policies and goals.

5.2 Shared-state Scheduler Deep Dive

There are many works following the shared-state design [7, 14, 17, 28, 31] to optimize various aspects as shown in Table 4. Apollo [7] of Microsoft uses the wait-time matrix to estimate jobs' wait time and infers future resource availability in the

Table 4: Comparison of related work on scheduler architectures

Categories	Schemes	Goal	Resource choice	Spatial V	Temporal V
Monolithic	Quasar[13], Paragon[12]	Resource Utilization	All available	Global	Up-to-date
Two-level	Mesos[22], Yarn[39]	Scalability	Subset available	Partial	Up-to-date
Shared-state	Omega[32], ParSync[17]	Schedule quality	All visible	Par-global	Stale
	Yaq-d[31], Tarcil[14]	Job complete time	All visible	Par-global	Stale
	RMiner (Ours)	Scheduling visibility	All available	Global	Up-to-date

+ V denotes resource visibility.

cluster. Yaq-d [31] optimizes queue sizing and reordering strategies of distributed schedulers. Tarcil [14] dynamically adjusts the sampling to reduce conflict rates and job completion time. Other frameworks [10, 11, 28] utilize work stealing, federated scheduling, and more advanced techniques to optimize. However, the gap between the update of schedulers' view and actual resource release causes the intermittently stale states of distributed schedulers (detailed in Section 2.3). Neither of the above works noticed such problems and failed to utilize the following invisible shadow resources.

To the best of our knowledge, ParSync [17] mentions the stale state problem within shared-state architecture, and it proposes a partial solution from the schedulers' perspective. ParSync partitions the cluster state into N parts and each scheduler maintains $1/N$ up-to-date resource view as prioritized scheduling destinations. It could eliminate shadow resources when specific schedulers have just the right jobs for the $1/N$ partition. However, it incurs great overhead to keep distributed schedulers up-to-date and cannot manage overall invisible resources. Therefore, our work is the first systematic analysis of shadow resources in shared-state scheduler architecture and enhances the spatial and temporal visibility of current designs by fully utilizing the invisible yet precious resources from a central perspective.

5.3 Discussion and Future Work

In this paper, we enhance the visibility of shared-state design by exploiting invisible shadow resources. Through experiments on industrial traces, we find that shadow resources are highly related to the overall workloads in the cluster. In traditional heavy-weight and long-lasting traces, the shadow resources are fewer because the fleeting resource fragments are negligible to normal occupation. However, the increase in the proportion of short tasks significantly increases the ratio of shadow resources. We keep the original distribution of industrial traces in our experiments and observe performance improvements with a wide range of fluctuations. Looking forward, lightweight and short-lived workloads provide more resource fragments to be utilized and the enhanced visibility definitely results in greater profits.

Meanwhile, there is no practical open-source shared-state scheduler at present, which hinders the realistic evaluation of the subsequent optimizations to the shared-state architecture. Given the only open-source industrial cluster simulator, we try our best to adjust the industrial traces and enhance the original scheduling process to evaluate RMiner fairly. However, the evaluation methodology and results could be better and more valid on top of a commercial shared-state scheduler in the cluster.

In the future, we will continuously optimize RMiner in two ways. Firstly, we will actively integrate RMiner into industrial shared-state scheduling systems, which could further validate the effectiveness of RMiner in realistic environments and the enhancement of current schedulers. Then, the current filter principles and scheduling algorithms in this paper are simple yet effective and they can be enhanced with more complex and intelligent techniques like machine learning techniques, which lead to more practical and efficient shared-state scheduler architecture.

6 CONCLUSION

In this paper, we take the first step to enhance the visibility of shared-state scheduler architecture to support utilizing shadow resources in the cluster. We conduct thorough theoretic and experimental analysis about the invisible shadow resource fragments and propose RMiner to both agilely mine shadow resources and transparently utilize them. Through industry-grade simulation, we show that RMiner can boost the overall performance of server clusters with minor overhead and conflicts. We expect that our design can help improve the resource efficiency of big data centers and various distributed micro data centers near the edge.

7 ACKONWLEDGEMENTS

We sincerely thank our shepherd, Yannis Chronis, and other anonymous reviewers for their valuable comments that helped us to improve the paper. This work is supported by the National Key R&D Program of China (No. 2022YFB4501702), and the National Natural Science Foundation of China (No. 62122053). Xinkai Wang, Hao He, and Yuancheng Li have contributed equally. The corresponding author is Chao Li.

REFERENCES

[1] 2014. Google cluster scheduler simulator. https://github.com/google/cluster-scheduler-simulator.

[2] 2014. SLURM. https://github.com/chaos/slurm.git.

[3] 2020. TORQUE. https://github.com/adaptivecomputing/torque.git.

[4] 2022. Alibaba Cluster Trace Program. https://github.com/alibaba/clusterdata.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[6] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *European Conference on Computer Systems*.

[7] 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 *Operating Systems Design and Implementation*.

[8] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Symposium on Operating Systems Principles*.

[10] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni Matteo Fumarola, Botong Huang, Kishore Chaliparambil, A. Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. 2019. Hydra: a federated resource manager for data-center scale analytics. In *Networked Systems Design and Implementation*.

[11] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association, 499–510.

[12] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

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

[14] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Symposium on Cloud Computing*.

[15] Dror G Feitelson. 1996. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS'96 Workshop Honolulu, Hawaii, April 16, 1996 Proceedings 2*. Springer, 89–110.

[16] Dror G Feitelson and Ahuva Mu'alem Weil. 1998. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE, 542–546.

[17] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. 2021. Scaling Large Production Clusters with Partitioned Synchronization.. In *USENIX Annual Technical Conference*. 81–97.

[18] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 583–594.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types.. In *Nsdi*, Vol. 11. 24–24.

[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[21] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters.. In *OSDI*, Vol. 16. 65–78.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[23] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Yang Hu, and Minyi Guo. 2020. ANT-Man: Towards agile power management in the microservice era. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[24] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Shaolei Ren, Jingwen Leng, Quan Chen, and Minyi Guo. 2021. AlphaR: Learning-powered resource management for irregular, dynamic microservice graph. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 797–806.

[25] Xiaofeng Hou, Jiacheng Liu, Chao Li, and Minyi Guo. 2019. Unleashing the scalability potential of power-constrained data center in the microservice era. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[26] 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.

[27] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[28] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*.

[29] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Symposium on Operating Systems Principles*.

[30] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *Symposium on Cloud Computing*.

[31] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *European Conference on Computer Systems*.

[32] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 351–364.

[33] Uwe Schwiegelshohn and Ramin Yahyapour. 1998. Analysis of first-come-first-serve parallel job scheduling. In *SODA*, Vol. 98. 629–638.

[34] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. 2002. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings. International Conference on Parallel Processing Workshop*. IEEE, 514–519.

[35] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Symposium on Cloud Computing*.

[36] Muhammad Tirmazi, Adam Barker, Nan Deng, E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *European Conference on Computer Systems*.

[37] Muhammad Tirmazi, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steve Hand, and Adam Barker. 2019. Google cluster-usage 2019 trace. The clusterdata-2019 trace dataset provides information about eight different Borg cells for the month of May 2019. https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[38] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[39] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[40] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric S. Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems*.

[41] Jing Wang, Chao Li, Junyi Mei, Hao He, Taolei Wang, Pengyu Wang, Lu Zhang, Minyi Guo, Hanqing Wu, Dongbai Chen, et al. 2022. HyFarM: Task Orchestration on Hybrid Far Memory for High Performance Per Bit. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 33–41.

[42] Jing Wang, Chao Li, Taolei Wang, Lu Zhang, Pengyu Wang, Junyi Mei, and Minyi Guo. 2022. Excavating the potenFtial of graph workload on rdma-based far memory architecture. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1029–1039.

[43] Xinkai Wang, Chao Li, Lu Zhang, Xiaofeng Hou, Quan Chen, and Minyi Guo. 2022. Exploring efficient microservice level parallelism. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 223–233.

[44] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 1–16.

[45] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. 265–278.

[46] Lu Zhang, Chao Li, Xinkai Wang, Weiqi Feng, Zheng Yu, Quan Chen, Jingwen Leng, Minyi Guo, Pu Yang, and Shang Yue. 2023. FIRST: Exploiting the Multi-Dimensional Attributes of Functions for Power-Aware Serverless Computing. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[47] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, Vol. 7. 1393–1404.