Hestia: An Attention-guided Scheduling Framework for Interference Mitigation among Cloud Services

Dingyu Yang, Jie Dai, Shiyou Qian, *Member, IEEE*, Hanwen Hu, Qin Hua, Jian Cao, *Senior Member, IEEE*, and Guangtao Xue, *Member, IEEE*

***Abstract*—To improve resource efficiency, a server usually deploys multiple latency-sensitive (LS) instances. On the other hand, this co-location may cause performance degradation due to resource contention. Numerous studies have proposed interference-aware scheduling methods or utilized resource isolation such as Linux $cpuset$ to mitigate interference. However, there is currently a lack of fine-grained binding core strategy to effectively guide production scheduling. In this study, we analyze trace data of LS services obtained from a production cluster comprising 32,408 instances deployed across 3,132 servers. We identify two interference patterns: sharing-core (SC) and sharing-socket (SS) interference. Based on our findings, we introduce Hestia[1], an attention-guided hyperthread-level interference mitigation framework to complement existing schedulers. Hestia incorporates a CPU utilization predictor for co-located LS instances, which utilizes a self-attention mechanism to consider SC and SS neighbors, as well as workloads and hardware heterogeneity. Moreover, Hestia establishes an interference scoring model to estimate the potential interference of LS instances prior to scheduling. To evaluate the effectiveness of Hestia, we deploy it in a production cluster with 19,760 cores. The results demonstrate that Hestia reduces the total cluster CPU usage by 2.27% and decreases the 95th percentile service latency by 13.9%. We also compare Hestia with five baselines based on large-scale data-driven simulations, revealing an achievement in the performance of LS services of up to 20.89%.**

***Index Terms*—cloud computing, scheduling framework, performance interference, workload characterization**

I. Introduction

To enhance resource utilization, cloud providers often employ a practice called co-location, where multiple latency-sensitive (LS) instances are hosted on a single server [32]. This strategy is widely adopted in large-scale data centers like Google [30] [32] and Microsoft [39]. However, co-locating LS instances may lead to performance interference caused by resource contention [26] [10] [9] [36] [12] [27]. To reduce this side effect, some cloud providers opt to bind each LS instance exclusively to a specific set of CPU cores, ensuring resource isolation [30] [32] [20] [32] [7] [35] [21]. Despite this approach, interference still occurs among co-located LS instances due to resource sharing, such as last-level cache (LLC) or memory bandwidth [13] [15].

Identifying and mitigating resource contention in production co-location scenarios poses significant challenges, which are reflected in three aspects: service diversity, interference mutuality, and scalability concerns. First, there are a large number of LS services, and they exhibit great diversity in resource requirements and sensitivity to interference. Second, the interference between LS instances is mutual, resulting in complex interference relationships among co-located instances. Third, scheduling long-running LS instances with different resource requirements at core granularity becomes a complex problem due to the number of cores, servers, and services involved. This problem can be formulated as the NP-hard bin packing problem [8] [14].

Researchers have invested great efforts in addressing these challenges. Existing methods for analyzing interference can be broadly categorized into two groups: interference injection-based and trace data-driven approaches. The first category involves analyzing interference by executing applications co-located with well-designed benchmarks [23] [34] [9] [22] [10]. These benchmarks enable the generation of various types and intensities of interference by adjusting the pressure on different hardware resources. The second category of methods focuses on identifying interference patterns in applications by analyzing historical trace data [15] [36].

In the realm of large-scale production co-location settings that employ core-binding techniques, current interference mitigation methods face a significant limitation. These methods schedule LS instances at the level of individual servers, thereby lacking the capability to accurately predict application performance when LS instances are bound to different processor cores. As a result, they fail to adequately quantify the interference that may occur among LS instances in such large-scale environments. Recently, some research [21], [30] has shown that deploying applications using the Linux $cpuset$ can lead to significant performance improvements. However, none of these studies explicitly investigate allocation policies that assign specific cores to each LS instance while considering interference.

In this paper, we conduct an in-depth analysis of substantial LS services by leveraging trace data obtained from a production cluster composed of 3,132 servers (see in §IV-B). Our analysis aims to identify and comprehend two distinct levels of interference from an architectural perspective: sharing-core (SC) and sharing-socket (SS) interference. In regard to CPU utilization, our findings reveal a significant impact on the

D. Yang is with the Alibaba Group, Shanghai, China. E-mail: dingyu.ydy@alibaba-inc.com

J. Dai, S. Qian, H. Hu, Q. Hua, J. Cao and G. Xue are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. E-mail: {daijie6, qshiyou, hanwen_hu, huaqin, cao-jian, gt_xue}@sjtu.edu.cn

S. Qian is the corresponding author.

[1]In Greek mythology, Hestia is the guardian goddess of the home, dedicated to resolving conflicts and disputes among family members. Drawing an analogy, we consider each server as a household and the co-located services running on it as family members.

performance of LS instances caused by their SC and SS neighbors. This impact arises due to the varying levels of resource sharing associated with these two types of interference. **To the best of our knowledge, our work pioneers the explicit modeling of CPU core and socket level interference based on server architecture.**

Motivated by these insights, we introduce Hestia, a hyperthread-level interference-aware scheduling framework. (see in §V). Hestia comprises two components. Firstly, utilizing attention mechanisms [31], we establish a model to predict the performance of LS instances based on their co-located SC and SS neighbors, along with workloads and CPU models. We leverage attention mechanisms to capture the intricate interference relationships between co-located LS instances. Secondly, based on the prediction model, we develop an interference scoring mechanism to proactively address interference awareness during the scheduling of LS instances. Hestia demonstrates two notable features. Firstly, it boasts lightweight characteristics due to the minimal overhead involved in collecting the necessary metrics, a crucial aspect for large-scale production clusters. Secondly, Hestia is non-intrusive, allowing for seamless integration with any scheduler, including current server-level interference-aware schedulers. This capability enables Hestia to effectively complement existing work in the field.

We evaluate the effectiveness and performance of Hestia by deploying it in a production cluster comprising 19,760 cores. In addition, our evaluation encompasses comprehensive data-driven simulations. Given Hestia's ability to complement any server-level scheduler, we select five established schedulers as baselines for comparison. The results of our experiments conducted on the production cluster show that, when subjected to the same workload, Hestia exhibits significant improvements. Specifically, Hestia achieves a decrease in total cluster CPU usage by 2.27% while reducing the 95th percentile latency of LS services by 13.9%. Furthermore, through simulation studies, we observe that Hestia outperforms the five baselines in terms of total CPU usage, showcasing a reduction range from 5.49% to 6.80%.

Our main contributions can be summarized as follows:

- We recognize the interference patterns of co-located LS services from the perspective of server architecture based on the trace data of 32,408 LS instances deployed in a production cluster consisting of 3,132 servers.
- We design an interference-aware scheduling framework at the hyperthread granularity, which achieves proactive interference mitigation for large-scale LS services. Specifically, our interference scoring mechanism can be seamlessly integrated with existing schedulers.
- We evaluate the effectiveness of Hestia in a real production cluster with 19,760 cores and compare it with baselines based on large-scale data-driven simulations.

II. BACKGROUND

A. Deployment of LS Services in Clouds

LS services typically serve as user-facing applications, such as web search, webmail, and e-commerce [32] [9] [10] [30] [38]. These large-scale LS services are commonly deployed in clouds, which possess three key characteristics. Firstly, multiple LS services, organized as microservices, collaborate to handle incoming requests, resulting in a complex call stack of interdependencies. The performance degradation of a downstream service can have cascading effects on multiple upstream services [12]. Secondly, an LS service often comprises a significant number of instances, ranging from tens to thousands, which evenly process requests forwarded by the load balancer [25] [19]. Consequently, when the performance of certain LS instances deteriorates significantly, some users may experience heavy-tailed latency. Thirdly, LS instances execute within long-lived containers[2], with lifetimes ranging from hours to months [20] [13]. Ensuring low latency for these long-lived, interdependent LS services presents a significant challenge [32] [18].

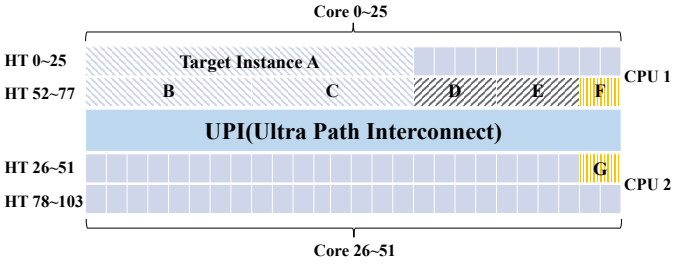


Fig. 1. The architecture of a dual-socket server and illustration of three levels of interference between co-located LS instances.

B. Server Architecture

The architectural configuration of a typical server in production environments is depicted in Fig. 1. This server comprises two sockets, each housing an Intel Xeon Platinum 8200 Series CPU based on the Cascade Lake micro-architecture [2]. The two sockets are interconnected using Ultra Path Interconnect (UPI). Each CPU supports 52 hyperthreads[3] (HTs) distributed across 26 physical cores. For instance, HTs 0∼25 and HTs 52∼77 are virtually assigned to cores 0∼25 in the first CPU. HT 0 and HT 52 share a physical core, with their own registers but a shared L1/L2 cache. Cores 0∼25 share LLC, while each core possesses its own L1/L2 cache.

C. Interference of LS Services

In essence, the interference among co-located LS instances[4] primarily stems from the sharing of low-level hardware resources, including hierarchical caches [17] and memory bandwidth [23] [34]. Within a dual-socket server, interference can manifest at three levels: core-level sharing, socket-level sharing, and server-level sharing [15]. We employ the instances depicted in Fig. 1 for illustrative purposes. Firstly, instances F and G solely share server-level resources, such as memory

[2]In this work, the terms 'container' and 'instance' are used interchangeably.

[3]In the context of Intel CPUs, the terms 'hyperthread' and 'logical core' are used interchangeably.

[4]Interference may arise from best effort (BE) jobs, such as data analysis. This situation can be mitigated by resource limitations imposed on BE jobs [36].

bandwidth, disk, and network I/O. Since there is no competition for CPU resources, the degree of CPU interference between them is negligible. Secondly, instances D and E share both server-level as well as socket-level resources, such as LLC. Thirdly, instance A shares core-level resources, such as L1/L2 cache, with instances B and C.[5] Meanwhile, they also share socket-level and server-level resources. As a result, the co-located neighbors of an LS instance can be classified into three types based on the differences in shared computation resources: instances sharing a core (**SC**), instances sharing a socket (**SS**), and instances located in the opposite socket (**OS**).

As there are a significant number of LS instances associated with various LS services, the cluster scheduler can employ diverse optimization strategies to allocate a set of LS instances on each server, with the objective of enhancing resource efficiency while minimizing interference at different levels. Therefore, in this paper, our emphasis lies on interference-aware scheduling for LS instances at the granularity of hyperthreads. For each LS instance, we propose recommendations not only for the server, but also for a specific set of HTs within the server. Therefore, our work serves as a complement to the existing schedulers operating at the server-level granularity.

D. Interference Assessment with Dynamic Workloads

The workload of LS services exhibits significant variations over time due to diurnal patterns [30] [37] [28] [20], presenting considerable challenges for interference analysis. It is necessary to account for interference under dynamic workloads to ensure the quality of service. Otherwise, a myopic scheduling decision made during a period of low workload may result in performance violations in the future. Hence, when characterizing the interference between LS instances, we also take into consideration the influence of workloads. By considering the maximum potential interference, our proposed framework, Hestia, enables predictive interference-aware scheduling for LS instances.

III. RELATED WORK

A. Interference Analysis

The primary objective of interference analysis is to predict application performance or establish scheduling guidelines for co-located applications. Current research can be categorized into two groups based on the method of data collection employed: interference injection-based and historical data-based approaches.

There are several analysis models that gather data through interference injection [23] [34] [38] [40] [29] [27] [5] [9] [10]. For instance, Bubble-up [23] focuses on memory interference, utilizing a well-designed pressure-tunable bubble. By adjusting the intensity of the bubble while an application is running, the sensitivity curve and pressure score of the application are obtained. These metrics are then employed to predict the application's performance under realistic interference. However, models based on interference injection have two inherent limitations. Firstly, the injected interference often fails to perfectly simulate real-world scenarios. Secondly, profiling applications under injected interference can be costly. To solve the latter concern, some studies leverage machine learning techniques to classify new applications based on profiles of existing applications [9] [10] [27], although these methods still rely partially on interference injection.

[5]In the exclusive binding scenario, we avoid allocating complete cores (each comprising two HTs) to a single LS instance to prevent the underutilization of computational power for instances with relatively low computational requirements.

TABLE I
COMPARISON OF INTERFERENCE-AWARE SCHEDULERS

Method	Interference resource	Core granularity	SC interference	SS interference	Interference-injection free	Performance prediction
Bubble-up [23]	Memory	-	-	-	✗	✓
Kam [15]	CPU	✓	✓	✗	✓	✗
CPI^2 [36]	CPU	✗	✗	✗	✓	✗
Paragon [9]	Multi-resource	✗	✗	✗	✓*	✗
DeepDive [26]	Multi-resource	✗	✗	✗	✓	✓
Mage [27]	Multi-resource	✓	✗	✗	✓*	✓
Hestia (ours)	CPU	✓	✓	✓	✓	✓

✓*indicates interference injection-based methods that classify new instances based on profiled instances.

Another category of methods relies exclusively on historical trace data for interference analysis [15] [26] [36] [12]. For instance, Kambudar et al. [15] extract qualitative affinity and anti-affinity information for two applications sharing CPU cores from Google's trace data. However, their work lacks quantitative interference analysis and overlooks socket-sharing interference. CPI^2 [36] identifies abnormal samples in CPI distribution as victims but does not provide interference prediction. Similarly, DeepDive [26] employs clustering techniques to detect performance interference among virtual machines.

B. Interference-aware Scheduling

Recent studies have explored various approaches for mitigating interference. Some methods aim to prevent the co-location of applications that generate substantial interference with each other [9] [5] [27] [29] [15] [23] [26] [10]. Conversely, other methods primarily focus on resource partitioning [11] [16] [17] [22] [6]. For instance, Bubble-up [23] and Bubble-flux [34] address memory interference by preventing the co-location of memory-intensive applications. Similarly, Paragon [9] employs classification techniques to reduce profiling time, while Quasar [10] additionally considers the amount of allocated resources. Both approaches use a greedy algorithm based on classification results to strike a balance between performance and resource utilization during scheduling. Regarding resource partitioning, Lo et al. [22] utilize existing hardware and software isolation techniques to ensure the quality of LS services co-located with BE jobs.

C. Discussion

Table I compares Hestia with several state-of-the-art models. Firstly, Hestia serves as a valuable extension to CPU interference modeling at the granularity of hyperthreads. While Mage [27] addresses interference at the core level, it fails to incorporate interference relationships from both the SC and SS resource competitions. Secondly, Hestia offers a completely interference-injection-free approach. To the best

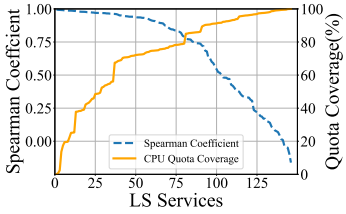


Fig. 2. Ranked Spearman coefficients and the corresponding CPU quota coverage of the 146 LS services.

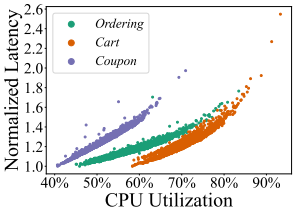


Fig. 3. Correlation between CPU utilization and latency of the top three services with most instances.

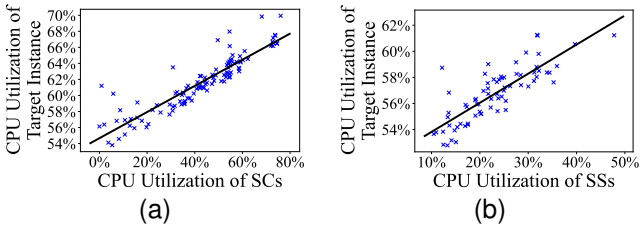


Fig. 4. The interference relationships between $Ordering$ instances CPU utilization and that of their SC and SS neighbors.

of our knowledge, only a limited number of studies have adopted a fully data-driven approach to achieve interference-aware scheduling. Furthermore, Hestia effectively mitigates interference by proactively predicting performance prior to scheduling, aiming to ensure the quality of LS services.

IV. OBSERVATIONS AND MOTIVATION

This section presents the insights from our analysis conducted on a production cluster, accompanied by the motivation behind this research endeavor.

A. Production Cluster

The cluster comprises 3,132 servers, each equipped with CPUs configured as illustrated in Fig. 1. Every server in the cluster possesses 384GB of memory. These servers collectively host a total of 35,738 instances belonging to 464 LS services. To analyze the performance of these services, we gather trace data during peak load testing. From this dataset, we identify 146 key LS services, each of which has more than 30 instances, to ensure an ample sample size representative of the workloads. These selected services encompass a total of 32,408 instances, constituting 93.47% of the overall CPU quota. Although our observations are based on our production cluster, we explore their underlying causes to investigate their broader applicability.

B. Observations

Observation 1: There exists a positive correlation between the CPU utilization of LS instances and the latency they experience.

The CPU utilization of an LS instance is defined by the percentage of CPU time consumed relative to the elapsed time. In a multi-core system, the elapsed time is multiplied by the number of requested HTs, denoted as $requests.ht$:

$$CPU\_utilization = \frac{CPU\_time}{(requests.ht * time\_real\_passed)}$$

It is important to note that not all CPU time is dedicated to executing instructions. When a cache miss occurs, the CPU must wait for data or instructions. Furthermore, instances executing simultaneously on two HTs of a core contend for physical core resources, such as the floating-point unit, which can potentially result in increased CPU utilization. This increased utilization, in turn, suggests a less efficient utilization of the CPU when processing the same workload.

Initially, we compute the Spearman correlation coefficients between CPU utilization and latency for each LS service. Fig. 2 illustrates these coefficients alongside their CPU quota coverage, with LS services sorted in descending order based on their coefficients. Remarkably, there are 78 services with Spearman coefficients exceeding 0.8, accounting for 78.69% of the total CPU quota. For instance, the top three services in terms of CPU quota, namely $Ordering$, $Cart$, and $Coupon$, comprise 4,276, 1,943, and 1,911 instances respectively. As shown in Fig. 3, the latency of these LS instances, normalized by the minimum latency, increases with CPU utilization. This relationship between latency and CPU utilization in our cluster corroborates the theoretical deduction that higher CPU utilization signifies a reduction in CPU efficiency.

Implication: Based on this observation, it can be argued that CPU utilization can effectively serve as an effective metric to reflect the latency experienced by users during their interaction with LS services.

Observation 2: A noticeable correlation exists between the CPU utilization of LS instances and the CPU utilization of their SC and SS neighbors.

Our investigation probes into the relationship between the CPU utilization of an LS instance and those of its SC and SS neighbors respectively. We take the $Ordering$ service as an exemplary case, accounting for 11.70% of the total CPU quota. Specifically, we select 114 instances from the $Ordering$ service whose SC neighbors exhibit varying CPU utilization while their SS neighbors maintain nearly consistent CPU utilization levels. As depicted in Fig. 4a, the CPU utilization of these instances showcases a nearly linear correlation with that of their SC neighbors[6]. Additionally, a linear correlation is also discernible between the CPU utilization of $Ordering$ instances and that of their SS neighbors, as shown in Fig. 4b. However, we do not find analogous evidence for the OS neighbors, since the competition stemming from SC and SS neighbors is more pronounced than that from OS neighbors.

Informed by the above analysis, we employ linear regression to model the relationship between the CPU utilization of an LS instance (the dependent variable) and that of its SC and SS neighbors (the independent variables). The mean of the R^2 values (the coefficient of determination) is approximately 0.83, with over 80% of services' R^2 values exceeding 0.75. These findings underscore the universality of the linear relationship.

Implication: The CPU utilization of an LS instance can also

[6]In the implementation, the CPU utilization of SC neighbors represents the average CPU utilization of the opposite HTs of the target instance, whereas the CPU utilization of SS neighbors denotes the average CPU utilization of all HTs in the socket, excluding those of the target instance.

act as an indicator, reflecting the interference originating from its neighboring SC and SS instances.

C. Motivation

Observation 2 elucidates a significant correlation between the level of interference experienced by an LS instance and the activities of its neighboring SC and SS instances. Simultaneously, Observation 1 suggests that CPU utilization can serve as a viable metric for assessing the extent of interference endured by an LS instance. Motivated by these insights, we delve into the optimization of LS instance orchestration at the granularity of CPU HTs. In terms of CPU utilization, we introduce Hestia, a novel scheduling framework that is acutely aware of interference for the management of LS services in production clusters.

V. Design of Hestia

A. Overview

The design of Hestia presents three challenges: service diversity, interference mutuality, and scalability concerns. The first challenge arises from the wide range of diversity observed among LS services in production clusters. These services exhibit significant variations in their sensitivity to CPU resources and the quantity of CPU resources they require. Moreover, LS services may be deployed on heterogeneous servers, and their workloads are known for their dynamic nature over time. Consequently, it is crucial to have a universally applicable metric for measuring interference. The second challenge relates to the mutual interference between LS instances. Adopting the approach that only aims to minimize interference for a new instance may inadvertently cause significant performance degradation for existing instances within the same server. The third challenge concerns the scalability of the scheduling mechanism. Scheduling at the granularity of hyperthreads, as opposed to server granularity, can introduce higher overheads that may potentially hinder the scalability of the scheduler.

Based on our data analysis, we choose to employ CPU utilization as the metric for quantifying the interference experienced by LS services. Firstly, as supported by Observation 2, CPU utilization effectively captures the extent of interference. Secondly, CPU utilization is readily available and cost-effective to measure. This is particularly important in the context of large-scale clusters where cost considerations play a crucial role. Thirdly, as highlighted in Observation 1, CPU utilization demonstrates a correlation with latency, which is a vital factor in ensuring the quality of LS services. Therefore, our objective is to accurately quantify and mitigate the level of CPU interference among co-located LS instances.

To tackle the aforementioned challenges, we propose Hestia, a hyperthread-level scheduling framework that is designed to address interference effectively. Firstly, Hestia incorporates an interference model that strategically utilizes the self-attention mechanism [31]. This model is specifically tailored to predict the CPU performance of LS instances, taking into account the intricate interference relationships among SS and SC neighbors. Moreover, it considers factors such as CPU heterogeneity and dynamic workloads, thereby ensuring a comprehensive understanding of the interference landscape. Secondly, Hestia introduces a scoring mechanism to quantify the mutual interference experienced by co-located LS instances. The primary objective of this mechanism is to optimize the overall performance of LS services and improve resource efficiency.

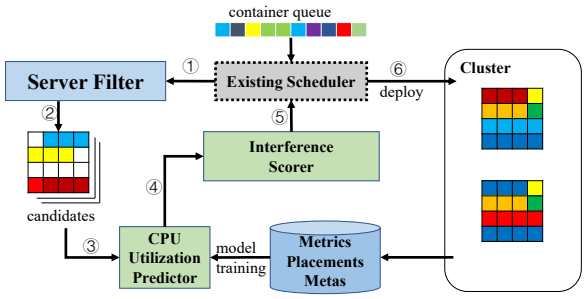


Fig. 5. The system architecture of Hestia.

B. System Architecture

The architecture of Hestia is depicted in Fig. 5, encompassing three components: the server filter, the CPU utilization predictor, and the interference scorer. 1) The server filter (§V-C) is responsible for selecting a set of candidate servers and candidate HT sets on each server that satisfy specific constraints [20], such as resource requirements. In contrast to the server filters implemented in existing schedulers, such as kube-scheduler [3], our server filter identifies both candidate servers and HT sets, rather than solely operating at the server granularity. 2) The CPU utilization predictor (§V-D) is devised to estimate the CPU utilization of LS instances on a candidate server prior to scheduling. This estimation assumes that the new instance will be scheduled to the potential server and HT sets. 3) The interference scorer (§V-E) is responsible for quantifying the comprehensive interference of each candidate server, thereby facilitating the selection of the optimal server and HT set. Furthermore, the existing scheduler module in Fig. 5 illustrates that Hestia can be easily integrated into existing server-level interference.

Specifically, for each new LS instance τ to be scheduled, Hestia follows a workflow, as depicted in Fig. 5. Initially, leveraging the metadata associated with τ, such as the requested resource quantity and specific scheduling rules, the server filter identifies sets of candidate HTs on a group of candidate servers. Subsequently, to evaluate the interference experienced by τ as well as the interference caused by τ on each candidate server, the CPU utilization predictor estimates the CPU utilization of all co-located instances assuming that τ is placed on the server. Then, based on the predicted CPU utilization, the interference scorer quantifies the degree of interference for each candidate server. Finally, τ is scheduled to the candidate server with the lowest interference score.

C. Server Filter

The operational process of the server filter consists of two sequential steps. Firstly, the filter can employ any scheduling policy to select a group of candidate servers capable of accommodating the new LS instance. Examples of such

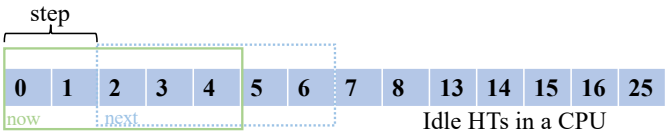


Fig. 6. Sliding window for selecting HTs.

policies include the $spread$[7] and $stack$[8] policies. Hestia can seamlessly integrate into any scheduler to enhance interference awareness. Subsequently, for each candidate server, the filter identifies multiple sets of idle HTs. This secondary selection step ensures that the server filter considers various potential HT options within the candidate server.

To minimize overhead, the selection of HT sets adheres to two principles. Firstly, instances are deployed within a single socket to minimize inter-socket communication. Secondly, Hestia allocates continuous idle HTs to each LS instance. As illustrated in Fig. 6, Hestia uses a sliding window approach to acquire a specific number of HT sets from the pool of idle HTs. The size of the sliding window is equivalent to the number of requested HTs, which, in conjunction with step size, determines the number of candidate HT sets.

D. Attention-guided CPU Utilization Prediction Model

Given a new instance and a set of recommended HTs within each candidate server, the primary objective is to assess the mutual interference that may occur between the new instance and its potential SC and SS neighbors. This evaluation takes into account factors such as CPU heterogeneity and dynamic workloads. To evaluate this interference accurately, we propose the establishment of a model that predicts the CPU utilization of both the new LS instance and the existing LS instances if it is deployed on a set of HTs within the candidate server. By predicting CPU utilization which serves as a reflection of interference, we can effectively quantify the potential impact of co-located LS instances on each other.

1) Design Idea: Accurately predicting the CPU performance of each LS instance co-located on a server is crucial for estimating mutual interference. Based on observations presented in Section IV-B, it is evident that the CPU utilization of LS instances is influenced by their neighboring instances. Moreover, previous research has demonstrated that factors such as workload measured in terms of RPS [28] and CPU model [9], [27] also impact the CPU utilization of LS instances.

While the load balancing in production clusters typically ensures an equitable distribution of workload among different instances associated with a service, Hestia explicitly takes into account the RPS of each individual instance. This consideration assumes significance due to the following reasons: 1) Despite load balancing efforts, slight variation in the RPS may persist across different instances belonging to the same service. Hestia acknowledges these differences, as they have the potential to impact the overall performance. 2) RPS demonstrates substantial temporal variations, as evidenced by diurnal patterns observed in previous studies [30]. By incorporating RPS into its decision-making process, Hestia effectively adapts to these temporal fluctuations. 3) Hestia exhibits a generic capability to function effectively in clusters that lack a load balancer, thereby enhancing its versatility and applicability.

Therefore, our objective is to design a comprehensive model that considers interference relationships, workloads, and CPU model to predict CPU utilization for LS instances. In production environments, it is indeed challenging to establish a theoretical model that accurately characterizes the complex interference relationships, primarily due to several factors. For instance, in our production cluster, a server's single socket frequently accommodates five or more instances belonging to different services. Furthermore, these instances are bound to 48 or 52 HTs, implying that an instance may share physical cores with multiple other instances. Moreover, they also share socket-level resources with other instances, further complicating the modeling of these interference relationships.

To tackle this challenge, we adopt a data-driven approach. Although previous models, such as those in FECBench [4] and Paragon [9], leverage machine learning techniques to forecast the performance degradation resulting from interference, they neglect the aspects of HT binding. Specifically, they do not account for which instances share a physical core or a socket. Consequently, they fall short in accurately predicting performance in scenarios involving HT binding. In our predictor, we utilize the self-attention mechanism [31] as the central component of the encoder. This mechanism enables relationships among input items to be effectively captured, making it suitable for modeling complex interference relationships. However, the self-attention mechanism does not incorporate the layout information of LS instances, limiting its ability to model multiple types of interference relationships. To overcome this limitation, we propose two dedicated encoders based on the self-attention mechanism. These encoders are specifically designed to independently model SC interference and SS interference.

The architecture of the CPU utilization predictor is depicted in Fig. 7, comprising four components: the input layer, sharing core encoder, sharing socket encoder, and loss function.

2) Input Layer: The input layer is an LS embedding layer, which facilitates the mapping of each LS service to a corresponding vector. This LS embedding technique is similar to word embedding techniques commonly applied in the field of natural language processing [24].

To construct the input feature I for each socket, we employ a concatenation approach that combines the LS embedding, RPS, and CPU model. This can be formalized as:

$$I = [X_{(h,n)}W_{(n,e)}, rps, cpu\_model] \quad (1)$$

Here, W represents the embedding matrix, which consists of a set of learnable parameters. The one-hot encoding representation of LS services, denoted by $X_{(h,n)}$, plays a crucial role in the construction process. In this encoding, h denotes the

[7]The $spread$ policy, which prioritizes servers with the highest quantity of residual resources, such as available CPU cores, is widely utilized in practical schedulers, such as Kubernetes [3] and Docker Swarm [1].

[8]The $stack$ policy prioritizes servers with the least residual resources to minimize resource fragmentation [14].

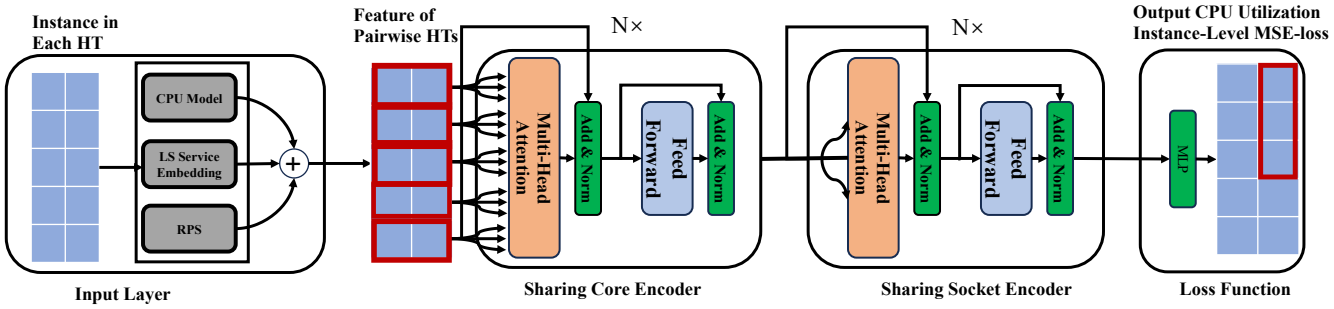


Fig. 7. The model architecture of CPU utilization predictor.

number of HTs in a socket, while n represents the number of services within the cluster, and e refers to the embedding dimension [24]. Each row of $X_{(h,n)}$ corresponds to a one-hot encoding of an LS instance, indicating its association with a specific set of HTs. It is important to note that certain HTs may be idle and unassigned to any instances. To address this, we allocate an "empty service" to these idle HTs, characterized by a workload and CPU utilization of zero.

3) Sharing-core Encoder: Interference can occur among sharing core instances due to the shared utilization of resources such as L1/L2 cache. To effectively capture this interference relationship, we introduce the "sharing core encoder", as illustrated in Fig. 7. This module facilitates the characterization of the extent of interference between instances that share a physical core. More specifically, for each pair of instances that share a physical core, we employ a self-attention-based encoder. In scenarios where the cores are homogeneous within a CPU, each physical core is assigned an equal weight within the encoder. However, if the CPU architecture involves heterogeneous cores, the shared weights within the self-attention module can be adjusted accordingly to accommodate such variations.

The formalization of the sharing core encoder E_j for each physical core j can be expressed as:

$$E_j = Attention(I_{(j_1,j_2)}) \tag{2}$$

Here, $I_{(j_1,j_2)}$ represents the input of two instances accommodated on core j, which corresponds to two rows of I in Eq. (1). $Attention(X)$ refers to the self-attention mechanism proposed in [31], which can be formally defined as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where $Q = W_Q X$, $K = W_K X$, and $V = W_V X$. To form SC_E as the input of feed-forward networks (FFNs), we concatenate $E_1 \sim E_l$, where l denotes the number of physical cores within a socket:

$$SC\_E = [E_1, E_2, ..., E_l] \tag{3}$$

$$SC\_H = FFN(SC\_E) \tag{4}$$

As defined in Eq. (4), upon the completion of the FFNs, the computation of the sharing core encoder concludes, and its output SC_H is subsequently passed to the sharing socket encoder.

4) Sharing-socket Encoder: Instances residing within the same socket often share resources, such as LLC. To capture this interference relationship, we employ an additional self-attention-based encoder, as illustrated in Fig. 7. Regarding SS interference, the specific layout of each instance within the socket holds limited significance. Hence, we directly utilize all the outputs from the sharing core encoder as the input for the sharing socket encoder. This can be formulated as:

$$SS\_E = Attention(SC\_H) \tag{5}$$

where SC_H denotes the output of the sharing core encoder. The FFNs employed in the sharing socket encoder are similar to the ones in the sharing core encoder.

5) Loss Function: The instance-level mean squared error (MSE) of CPU utilization serves as the loss function. The MSE loss is calculated in three steps. Firstly, we obtain the CPU utilization of each HT by employing a multilayer perception block. Then we compute the predicted CPU utilization for each individual instance i by:

$$CPU\_Pred_i = \frac{1}{m}\sum_{j=1}^{m} CPU_j \tag{6}$$

where m refers to the number of HTs allocated to instance i. Subsequently, the instance-level MSE is calculated as expressed in the loss function:

$$loss = \frac{1}{k}\sum_{i=1}^{k}(CPU\_Pred_i - CPU\_Truth_i)^2 \tag{7}$$

where k denotes the number of instances within a socket, and CPU_Pred_i and CPU_Truth_i represent the predicted and true CPU utilization of instance i, respectively.

E. Interference Scoring Mechanism

When presented with a new instance and the suggested HT sets across each candidate server, the scheduler needs to determine the optimal candidate that considers performance interference. Consequently, it is necessary to propose a metric for evaluating the candidate servers and candidate HTs, aiming to minimize interference. By leveraging the CPU utilization predictor, it becomes feasible to accurately estimate the potential interference experienced by the new instance before scheduling it on each candidate server.

We propose a two-step interference scoring mechanism. Firstly, utilizing the CPU utilization predictor, the CPU utilization of each LS instance i under various RPS when executed independently can be predicted as:

$$CPU_i^{woi} = Predictor(Instance\ i) \tag{8}$$

where CPU_i^{woi} represents the CPU utilization of instance i in the absence of interference, serving as the baseline reference. Secondly, in scenarios where multiple LS instances are co-located within a socket, we can estimate the CPU utilization of LS instances i with interference as CPU_i^{wi}.

Therefore, the interference score for an individual instance i can be defined as:

$$Score_i = \frac{CPU_i^{wi} - CPU_i^{woi}}{CPU_i^{woi}} \tag{9}$$

$Score_i$ represents the extent of CPU utilization increase caused by interference for instance i. To mitigate the influence of LS instances' workload on interference scoring, the score is normalized by CPU_i^{woi}.

By aggregating the interference scores of the new instance and its neighboring instances, we define the socket-level interference score as:

$$Score_{skt} = \sum w_i * Score_i \tag{10}$$

where w_i represents the ratio of the number of HTs assigned to each instance i to the total number of HTs allocated to LS instances in the socket. This equation quantifies the ratio of total CPU usage increase in the entire socket caused by interference among LS instances to the CPU usage of the real workload. Data analysis indicates that the CPU interference between sockets is negligible. Thus, Hestia employs socket-level scores as the basis for scheduling.

F. Discussion

We discuss the generalization of Hestia, highlighting its applicability in schedulers across various clusters, not limited to our own. Firstly, the practice of binding LS instances to specific cores is a prevalent approach for ensuring their performance [21] [30] [32] [33]. Thus, the challenges we have identified are widespread. Secondly, CPU resource contention is pervasive in large-scale clusters [36] [15]. Thirdly, the prediction and scoring models employed by Hestia are interpretable and grounded in common server architectures.

Several studies have highlighted the ubiquitous nature of CPU interference and emphasized the need for effective solutions to address this issue [15] [36]. Our work serves as a valuable addition to the existing body of research on CPU interference. It is worth noting that CPU utilization alone may not adequately reflect interference caused by memory-intensive or I/O-intensive applications. Several efforts have focused on exploring memory interference [23] [34] or interference arising from multiple resources [9] [26] .

Overall, in comparison to existing schedulers [10] [9] [26], Hestia offers four compelling features. Firstly, Hestia excels in granularity by recommending a set of HTs within a server for each new instance, while simultaneously minimizing interference. Secondly, Hestia considers the mutual interference among multiple co-located instances, rather than solely focusing on pair-wise interference. Thirdly, Hestia is non-intrusive, allowing for seamless integration into any scheduler. Lastly, Hestia is lightweight and imposes minimal overhead in terms of performance metric collection.

TABLE II
THE CONFIGURATION OF THE VALIDATION EXPERIMENTS.

Description	Value
Percentage of training dataset	0.8
Percentage of testing dataset	0.2
Optimizer	SGD
Learning rate	0.001
LS embedding dimension	32
Number of heads	2
Number of each encoder (N_x in Fig. 7)	2

VI. EVALUATION

A. Evaluation for CPU Utilization Predictor

We first evaluate the accuracy of the attention-guided CPU utilization predictor by leveraging monitoring data obtained from a 40-minute load-testing scenario. The dataset comprises 2 clusters, encompassing a total of 3,073 servers. These servers accommodate a total of 27,868 LS instances belonging to 200 services. The server architecture within the clusters can be categorized into two types. The first type consists of servers equipped with two Intel Xeon Platinum 8200 Series CPUs, which feature the Cascade Lake micro-architecture, as illustrated in Fig. 1. The second type consists of servers equipped with two Intel Xeon Platinum 8100 Series CPUs, featuring the Skylake micro-architecture [2]. The number of servers for each type is 945 and 2,128 respectively.

1) Prediction Accuracy: The configuration details of the validation experiments are outlined in Table II. We divide the dataset into an 80% training dataset and a 20% testing dataset. Additionally, Table II provides an overview of the hyperparameters employed in our CPU utilization predictor. We use two metrics to measure prediction accuracy: mean absolute error (MAE) and root mean squared error (RMSE).

Firstly, Fig. 8 shows the MAE and CPU quota coverage for the 200 LS services, arranged in ascending order based on their MAE values. All LS instances exhibit an MAE of 1.41% on average. Specifically, the MAE of 192 services falls below 3%, representing 91.8% of the total CPU quota. Secondly, Fig. 9 shows the RMSE and CPU quota coverage, with LS services also sorted in ascending order according to their RMSE values. Of these services, the RMSE of 185 services is below 4%, accounting for 91.86% of the total CPU quota, with an average RMSE of 1.81%. Additionally, we present results pertaining to five critical LS services that account for significant CPU quotas in our clusters. Table III highlights the high accuracy of these services achieved by the CPU utilization predictor.

2) Ablation Studies: To demonstrate the effectiveness of each module in the CPU utilization predictor, we compare the predictor with three variants, which are the predictor without SC encoders, the predictor without SS encoders, and the predictor disregarding the CPU model.

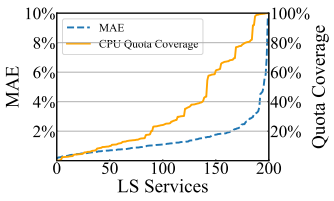


Fig. 8. The MAE and CPU quota coverage of services.

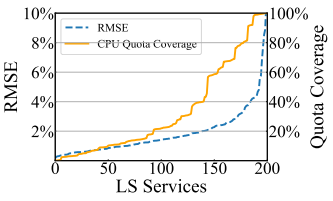


Fig. 9. The RMSE and CPU quota coverage of services.

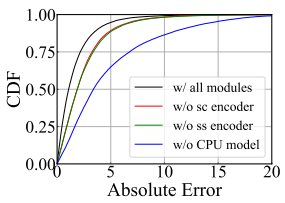


Fig. 10. The CDF of absolute error for all instances.

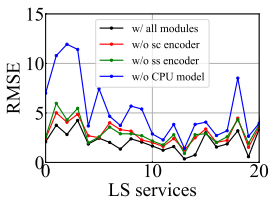


Fig. 11. RMSE for Top 20 services with the highest CPU quota.

TABLE III
THE VALIDATION RESULT OF FIVE KEY LS SERVICES.

Service name	Avg CPU Util.(%)	CPU quota ratio(%)	MAE(%)	RMSE(%)
Coupon	38.88	11.13	1.60	2.08
Cart	71.81	9.82	2.96	3.75
Detail	53.20	7.45	2.15	2.82
Ordering	65.42	5.95	3.18	4.24
Inventory	30.19	4.98	1.27	1.85

TABLE IV
THE RMSE AND MAE OF OUR PREDICTOR AND THREE BASELINES.

	w/ all modules	w/o SC encoder	w/o SS encoder	w/o CPU model
MAE	1.64	2.49	2.52	4.88
RMSE	2.48	3.66	3.54	6.65

Table IV lists the MAE and RMSE of our predictor and its three variants. The MAE of predictors w/o SC encoder, w/o SS encoder, and w/o CPU model are 1.51×, 1.53×, and 2.97× compared with the predictor with all modules. Similarly, the RMSE of predictors w/o SC encoder, w/o SS encoder, and w/o CPU model are 1.47×, 1.42×, and 2.68× compared with the predictor with all modules. These results indicate that the CPU model is the most significant factor influencing CPU utilization, while the SC encoder and SS encoder contribute approximately equally to this impact.

Fig. 10 shows the CDF of absolute error for instances in the testing dataset. Specifically, 5.85%, 11.81%, 12.82%, and 41.49% of instances demonstrate an absolute error exceeding 5% for the four predictors respectively. Fig. 11 shows the RMSE values for the 20 largest CPU quota services, accounting for 68.11% of the total CPU quota. It is noteworthy that the predictor w/o CPU model exhibits notably poor performance in certain LS services, with an RMSE exceeding 10.0%. Although the predictors w/o SC encoder and w/o SS encoder show improved performance, some LS services still exhibit an RMSE exceeding 5.0%. In contrast, the predictor incorporating all modules outperforms all the three variants, showcasing the lowest RMSE for these 20 LS services. These findings underscore the effectiveness of all modules in enhancing the accuracy of the CPU utilization predictor.

3) Comparison with Other Predictors: We compare our attention-based CPU utilization predictor with three performance prediction methods from existing research. To minimize the impact of load fluctuations on evaluation, we sampled a dataset with a constant workload during load testing. Our CPU utilization predictor is compared against the following methods:

i) Avg. This method calculates the CPU utilization of the target instance based on historical averages, disregarding the impact of interference.

ii) FECBench [4]. Barvel et al. introduced a model that utilizes machine learning techniques such as K-means, random forest, and decision tree to forecast the performance of applications affected by interference.

iii) MLP (multi-layer perceptron). This approach employs embedding vectors from multiple co-located applications as inputs to a multi-layer perceptron, with each application's performance serving as the output.

Fig. 12 shows the MAE and RMSE of CPU utilization prediction using different predictors. The RMSE values are 3.04%, 2.78%, 2.39%, and 1.93% when using Avg, Fecbench, MLP, and our predictor respectively. In terms of RMSE, the results are 4.25%, 4.14%, 3.22%, and 2.66% respectively. Similarly, Fig.13 presents the RMSE for five services with the highest CPU quota. Our predictor outperforms the others for four of these services. For example, our predictor achieves 1.13% to 3% lower RMSE for the *Coupon* service compared to the other predictors.

Compared to the attention-based predictor in Hestia, the three baseline predictors have several shortcomings: 1) Avg ignores interference between co-located services, resulting in the worst performance among all predictors. 2) Both MLP and FECBench assume that only one type of interference relationship exists among different services running on a single server, leading to poorer performance compared to our method. Our predictor considers both SS and SC interference, resulting in higher accuracy in CPU utilization prediction.

B. Simulation based on Large-Scale Monitoring Data

We compare Hestia with five baselines in a simulator based on the container monitoring data from our production cluster.

1) Methodology: **Simulator.** The simulator is implemented using Python, which simulates the scheduling workflow of Hestia and baselines, as shown in Fig. 5. First, the server filter selects a group of candidate servers using the *spread* policy[9] which prefers servers with the most residual resources such as CPU cores. For each candidate server, a certain number of HT sets are selected from idle HTs. Then, based on the rules of corresponding schedulers, the best server and the corresponding HTs $(Server, HTs)$ are selected from the candidates[10].

To evaluate the performance of co-located LS services in the simulation, we employ the CPU utilization predictor described in section V-D to estimate their CPU utilization.

[9]The *spread* policy is widely used in actual schedulers such as Kubernetes [3] and Docker Swarm [1].

[10]A candidate refers to a pair comprising a candidate server and an HT set.

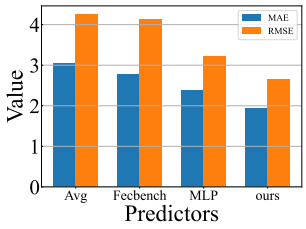


Fig. 12. The MAE and RMSE for 4 CPU utilization predictors.

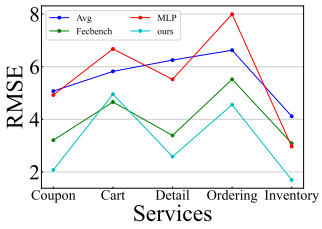


Fig. 13. RMSE for top 5 services with the highest CPU quota.

TABLE V
TOTAL CLUSTER CPU UTILIZATION OF TESTED SCHEDULERS.

	CPU_total	$Increase(\%)$
Hestia	80,217	0.00%
FF	85,673	6.80%
Socket-Spread	85,610	6.72%
LSC	84,625	5.49%
LSS	85,160	6.16%
Kambadur	84,945	5.89%

Settings. In the simulator, there are two parameters, namely the number of candidate servers and the step size of the sliding window. We set the number of candidate servers to 20 and the step size to 2, aiming to achieve a balance between performance and overhead. For different scenarios characterized by varying scheduling waiting time limitations, these two parameters can be configured accordingly.

Baselines. We compare Hestia with five practical schedulers that differ in their HT binding policies. Note that all schedulers select candidate servers using the *spread* policy.

i) FF (first-fit). This policy is widely used in production clusters, including our own. When a new instance is scheduled to a server, HTs are allocated in ascending order based on their IDs.

ii) Socket-Spread. HTs are assigned using a spread policy across two sockets, with a preference for sockets containing more idle HTs.

iii) LSC. This policy selects the HTs whose sharing core neighbors have the lowest CPU utilization.

iv) LSS. This policy prioritizes the socket with lower CPU utilization.

v) Kambadur[11] [15]. This policy was proposed by Kambadur et al., prioritizing HTs whose SC neighbors are classified as positive beyond noisy interferers ($pbnis$). They utilize CPI as the metric and employ statistical methods to identify $pbnis$.

We do not compare Hestia with existing state-of-the-art interference-aware schedulers or dynamic resource partition techniques for two main reasons. Firstly, the use of different metrics for evaluating interference and performance makes fair comparisons challenging within a simulator environment. Secondly, our study is orthogonal with existing research. Our focus on studying the binding core policy in large-scale clusters constitutes a novel contribution, distinct from existing research that typically concentrates on selecting the best server to mitigate interference. For example, previous works such as Paragon [9] and Bubble-up [23] have primarily focused on server-level scheduling, while others like Heracles [22] and PARTIES [6] have explored dynamic resource partitioning. These studies are orthogonal with ours. To showcase Hestia's potential, we compare it with four practical HT binding policies and one proposed by Kambadur et al. [15], demonstrating its capability to enhance the performance of LS instances through a well-designed HT binding strategy. Moreover, Hestia can be integrated into existing server-level schedulers to further improve their performance.

[11]For illustrative purposes, we refer to this policy as the first author's name.

Workloads. We simulate LS instance scheduling based on the monitoring data of LS services from the real production cluster. The dataset consists of 18,737 instances belonging to 75 LS services. Note that the workload of all services is the same across the six tested schedulers, so we focus on the interference caused by SC and SS neighbors. We simulated 2, 080 servers to accommodate these instances, half of them equipped with Intel Xeon 8200 Series CPUs and the remaining half with Intel Xeon 8100 Series CPUs.

2) Improving Performance Using Hestia: Based on Observation 1, given the same workload, a higher CPU utilization indicates a lower performance. In the following, we analyze the performance improvement using Hestia from the perspectives of cluster, service, and socket.

Cluster Level. We define the total cluster CPU usage as:

$$CPU\_total = \sum_{i \in instances} requests.ht_i * CPU_i \qquad (11)$$

where $requests.ht_i$ refers to the number of HTs requested by instance i, and CPU_i refers to the CPU utilization of instance i which is estimated by the predictor in Fig. 5. Table V lists the CPU_total using different schedulers. Compared with Hestia, the CPU_total of FF, Socket-Spread, LSC, LSS and Kambadur increases by 6.80%, 6.72%, 5.49%, 6.16%, and 5.89% respectively. These results validate that Hestia can effectively improve cluster efficiency by reducing interference among co-located LS services.

The performance improvement of Hestia, in comparison with the baselines, can be attributed to three factors. Firstly, Hestia considers the interference from neighbors while FF and Socket-Spread do not. Secondly, Hestia quantifies interference from SC and SS neighbors. Instead, while LSC accounts for interference from SC neighbors and LSS from SS neighbors, neither quantify the interference. Therefore, LSC and LSS perform better than FF and Socket-Spread but still worse than Hestia. Thirdly, while the method proposed by Kambadur et al. considers SC interference, it primarily conducts a qualitative analysis of the SC interference.

Service Level. We then evaluate the performance improvements of LS services with Hestia. Compared with FF, Socket-Spread, LSC, LSS, and Kambadur, Hestia lowers the average CPU utilization of all 75 LS services. Moreover, Hestia decreases the 95th percentile (P95) CPU utilization of 74, 74, 75, 75, and 75 LS services compared to the five baselines respectively. These results indicate that Hestia can substantially decrease the tail latency of LS services by mitigating interference.

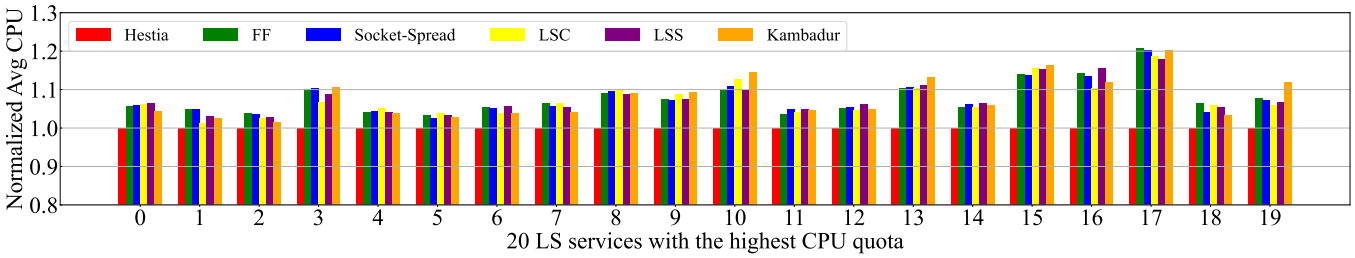


Fig. 14. Average CPU utilization increase of the baselines for 20 LS services with the highest CPU quota compared with Hestia.

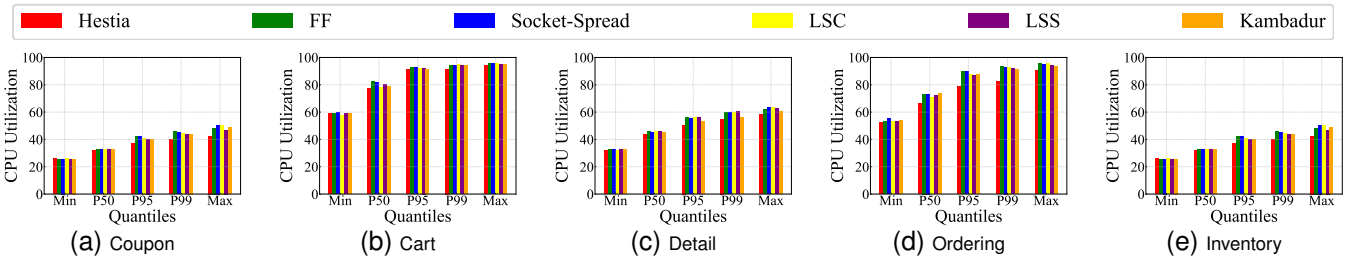


Fig. 15. CPU utilization of instances belonging to five key services, where the x-axis plots the instances sorted in ascending order by the value of CPU utilization.

Fig. 14 shows the average CPU utilization increase of the different schedulers for 20 LS services with the highest CPU quota in comparison to Hestia. These LS services account for 85.85% of the total CPU quota. Compared to Hestia, the average CPU utilization of different LS services when using FF, Socket-Spread, LSC, LSS, and Kambadur increases by 3.41%-20.89%, 2.62%-20.18%, 1.21%-18.74%, 3.00%-18.04%, and 1.63%-20.42%. respectively, indicating huge improvements in performance.

We further study the performance improvements of the five key services, namely *Coupon*, *Cart*, *Detail*, *Ordering*, and *Inventory*. These services account for 11.13%, 9.82%, 7.45%, 5.95%, and 4.98% of the total CPU quota respectively. Thus, they are more susceptible to suffering interference. Fig. 15a to 15e show the different quantiles of CPU utilization of instances belonging to these services. Overall, Hestia effectively stabilizes the instance CPU utilization of these services. Taking *Coupon* in Fig. 15a as an example, compared to FF, Socket-Spread, LSC, LSS, and Kambadur, the median CPU utilization of the *Coupon* service when using Hestia is reduced by 4.40%, 3.88%, 4.50%, 4.63%, and 3.49% respectively. Considering the P95 CPU utilization, the decreased values are 10.89%, 9.90%, 10.78%, 10.88%, and 5.65% respectively, significantly improving tail latency. The results in Fig. 15b-15e indicate similar improvements for *Cart*, *Detail*, *Ordering*, and *Inventory*.

Socket Level. As defined in Eq. (10), the socket-level interference score quantifies the ratio of CPU utilization increase caused by interference to CPU utilization without interference. Fig. 16 shows the cumulative distribution function (CDF) of socket interference scores using different schedulers. Hestia effectively mitigates and equalizes socket interference, whereas the other schedulers result in some sockets experiencing either excessively low or high interference. These findings indicate that Hestia has the capability to enhance and stabilize the performance of LS services.

Specifically, we classify sockets with interference scores not exceeding 0.2 as low-interference (LI) sockets, and those exceeding 0.4 as high-interference (HI) sockets. When utilizing Hestia, the proportion of LI sockets reaches 52.78%, whereas the corresponding figures for FF, Socket-Spread, LSC, LSS, and Kambadur are 29.23%, 25.45%, 30.02%, 25.96%, and 32.81% respectively. In addition, the percentage of HI sockets is 0.21% with Hestia, compared to 11.11%, 9.63%, 8.67%, 8.22%, and 9.13% for FF, Socket-Spread, LSC, LSS, and Kambadur respectively. By mitigating interference, Hestia achieves an average increase of 24.08 percentage points in LI sockets.

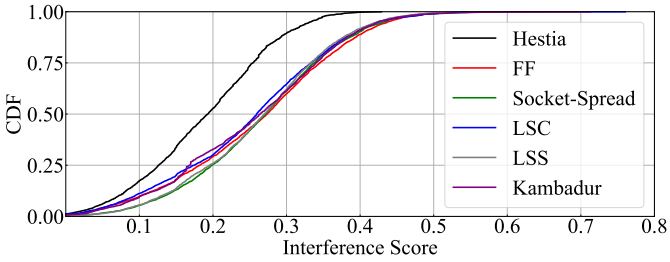


Fig. 16. CDF of the interference scores of 4,160 sockets.

C. Evaluation in a Production Cluster

We apply Hestia to a subset of a production cluster, comprising 190 servers that host 2,009 instances belonging to 247 LS services. The CPU utilization of LS instances in a service follows a normal distribution. Therefore, we classify instances whose CPU utilization falls outside $\mu + \sigma$ and $\mu + 2\sigma$ as mild and severe interference victims respectively. In total, we detect 121 instances belonging to 57 LS services as victims, including 74 mild and 47 severe victims. Notably, of these victims, 16 instances from the *Ordering* service are identified as severe victims. All victims are subsequently rescheduled using Hestia.

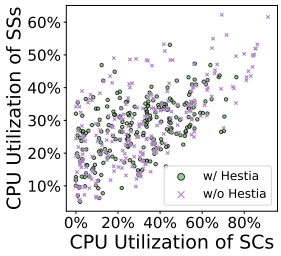


Fig. 17. CPU utilization of SC and SS neighbors of *Ordering* instances.

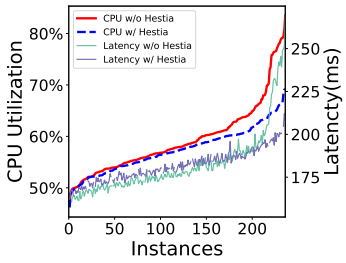


Fig. 18. CPU utilization and latency of *Ordering* instances in ascending order of CPU utilization.

We first evaluate the performance improvements achieved through the rescheduling of instances. To illustrate this, we consider the case of the 16 *Ordering* severe victims. Prior to rescheduling, the average CPU utilization of these instances, as well as their SC and SS neighbors, is recorded as 78.06%, 81.84%, and 50.44% respectively. Following the utilization of Hestia, these values decrease to 56.75%, 29.16%, and 27.80% respectively. In addition, before rescheduling, the average latency is 239ms. However, with the optimization by Hestia, this latency reduces to 180ms. Specifically, the rescheduled *Ordering* instances exhibit a 24.6% reduction in latency. Similarly, the latency of the 9 rescheduled *Coupon* instances decreases by 21.68%.

Owing to interference mutuality, the benefits derived from the rescheduling of instances extend beyond the individual instances themselves. We proceed to analyze the overall performance improvements at both the cluster and service levels. As defined in Eq. (11), the aggregate CPU usage of the cluster is 6714.26 prior to optimization. However, with Hestia, this value decreases to 6565.05. Consequently, Hestia succeeds in reducing the total CPU usage by 2.27% under the same workload, effectively mitigating interference.

At the service level, we analyze the performance enhancements of 236 *Ordering* instances. Fig. 17 depicts the CPU utilization of *Ordering*'s SC and SS neighbors with (w/) and without (w/o) Hestia. Firstly, Hestia effectively reduces the CPU utilization of *Ordering*'s SC and SS neighbors from 32.59% to 29.72% and from 28.92% to 27.29%, respectively. Secondly, as the pressure exerted by neighbors decreases, Hestia significantly enhances the performance of *Ordering*, as shown in Fig. 18. Specifically, the P95 CPU utilization of *Ordering* decreases from 76.46% to 64.78%. Correspondingly, the P95 latency of *Ordering* is reduced from 230ms to 198ms, resulting in an improvement of 13.9%.

VII. CONCLUSION

This study introduces Hestia, a scheduling framework that addresses interference among co-located LS instances by considering their neighbors in the CPU. Hestia achieves proactive interference mitigation by establishing a performance prediction model for co-located LS instances. Hestia is designed to be non-intrusive and lightweight, allowing for seamless integration into any scheduler. We also verify the effectiveness of Hestia through large-scale data-driven simulation and its actual deployment in a production cluster. In the future, there are two potential areas for further optimization in Hestia. Firstly, Hestia currently lacks the ability to model inter-calling dependencies among LS services. Furthermore, the greedy algorithm employed by Hestia disregards global scheduling information. Both of these aspects may lead to sub-optimal performance, thereby warranting further investigation.

ACKNOWLEDGMENTS

This work was supported by Alibaba Group through Alibaba Innovative Research Program and Alibaba Research Intern Program.

REFERENCES

[1] Docker swarm. https://docs.docker.com/engine/swarm, 2023.
[2] Intel product specifications. https://ark.intel.com/, 2023.
[3] Kubernetes. https://kubernetes.io/, 2023.
[4] Yogesh D. Barve, Shashank Shekhar, Ajay Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. FECBench: A Holistic Interference-aware Approach for Application Performance Modeling. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 211–221, 2019.
[5] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. *SIGARCH Comput. Archit. News*, 45(1):17–32, apr 2017.
[6] 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*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.
[7] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263, Santa Clara, CA, July 2017. USENIX Association.
[8] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Dynamic Bin Packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.
[9] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.
[10] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.*, 49(4):127–144, feb 2014.
[11] Christina Delimitrou and Christos Kozyrakis. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.
[12] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 19–33, New York, NY, USA, 2019. Association for Computing Machinery.
[13] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.
[14] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861. USENIX Association, November 2020.
[15] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring Interference between Live Datacenter Applications. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, page 1–12, USA, 2012. IEEE Computer Society.

[16] 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, 2015.

[17] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 729–742, New York, NY, USA, 2014. Association for Computing Machinery.

[18] 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, 2016.

[19] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 258–272, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Li Liu, Haoliang Wang, An Wang, Mengbai Xiao, Yue Cheng, and Songqing Chen. Mind the Gap: Broken Promises of CPU Reservations in Containerized Multi-Tenant Clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 243–257, New York, NY, USA, 2021. Association for Computing Machinery.

[22] 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*, ISCA '15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 248–259, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Yipei Niu, Fangming Liu, and Zongpeng Li. Load Balancing Across Microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.

[26] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 219–230, San Jose, CA, June 2013. USENIX Association.

[27] Francisco Romero and Christina Delimitrou. Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Xiongchao Tang, Haojie Wang, Xiaosong Ma, Nosayba El-Sayed, Jidong Zhai, Wenguang Chen, and Ashraf Aboulnaga. Spread-n-Share: Improving Application Performance and Cluster Throughput with Resource-Aware Job Placement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[32] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. Characterizing job microarchitectural profiles at scale: Dataset and analysis. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA, 2023. Association for Computing Machinery.

[34] 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.

[35] Renyu Yang, Chunming Hu, Xiaoyang Sun, Peter Garraghan, Tianyu Wo, Zhenyu Wen, Hao Peng, Jie Xu, and Chao Li. Performance-Aware Speculative Resource Oversubscription for Large-Scale Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 31(7):1499–1517, 2020.

[36] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 210–225, New York, NY, USA, 2022. Association for Computing Machinery.

[38] 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. MICRO-47, page 406–418, USA, 2014. IEEE Computer Society.

[39] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-Based harvesting of spare cycles and storage in Large-Scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 755–770, Savannah, GA, November 2016. USENIX Association.

[40] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1443–1456, 2016.

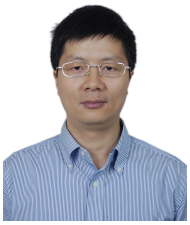
Dingyu Yang received his Ph.D. degree in computer science and technology from Shanghai Jiao Tong University, Shanghai, China, in 2015. He is currently a senior engineer at Alibaba Group, Shanghai, China. His research interests include resource prediction, anomaly detection in cloud computing, and distributed stream processing. He has published over 20 papers in some journals and conferences such as SIGMOD, VLDB, and VLDBJ.

Jie Dai is currently pursuing a Master degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests lie in resource management and interference mitigation of large-scale microservices.

Shiyou Qian (Member, IEEE) received his Ph.D. degree in computer science and technology from Shanghai Jiao Tong University Shanghai, China, in 2015. He is currently an associate researcher with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include event matching for content-based publish/subscribe systems, resource scheduling for the hybrid cloud, and driving recommendations with vehicular networks.

Hanwen Hu is currently pursuing a Ph.D. degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include intelligent transportation, map matching algorithm and visual SLAM.

Qin Hua is currently pursuing a Ph.D. degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include intelligent resource scheduling and scaling of micro services, deep learning, and time series forecasting.



Jian Cao (Senior Member, IEEE) received his Ph.D. degree in computer science and technology from the Nanjing University of Science and Technology, Nanjing, China, in 2000. He is currently a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His main research interests include service computing, network computing, and intelligent data analytics.

Guangtao Xue (Member, IEEE) received his Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai China, in 2004. He is a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include vehicular ad hoc networks, wireless networks, mobile computing, and distributed computing. He is a member of the IEEE, IEEE Computer Society, and the IEEE Communication Society.