Suppressing the Interference Within a Datacenter: Theorems, Metric and Strategy

Yuhang Liu, *Member, IEEE*, Xin Deng, *Student Member, IEEE*, Jiapeng Zhou, *Student Member, IEEE*, Mingyu Chen, *Member, IEEE*, and Yungang Bao, *Member, IEEE*

***Abstract*—As the paradigm of cloud computing, a datacenter accommodates many co-running applications sharing system resources. Although highly concurrent applications improve resource utilization, the resulting resource contention can increase the uncertainty of quality of services (QoS). Previous studies have shown that achieving high resource utilization and high QoS simultaneously is challenging. Moreover, quantifying the intensity of interference across multiple concurrent applications in a datacenter, where applications can be either latency-critical (LC) or best-effort (BE), poses a significant challenge. To address these issues, we propose Ah-Q, which comprises two theorems, a metric, and a scheduling strategy. First, we present the necessary and sufficient conditions to precisely test whether a datacenter is both QoS guaranteed and high-throughput. We also present a theorem that reveals the relationship between tail latency and throughput. Our theoretical results are insightful and useful for building datacenters that have desirable performance. Second, we propose the "System Entropy" (E_S) to quantitatively measure the interference within a datacenter. Interference arises due to resource scarcity or irrational scheduling, and effective scheduling can alleviate resource scarcity. To assess the effectiveness of a resource scheduling strategy, we introduce the concept of "resource equivalence". We evaluate various resource scheduling strategies to demonstrate the correctness and effectiveness of the proposed theory. Third, we introduce a new resource scheduling strategy, ARQ, that leverages both isolation and sharing of resources. Our evaluations show that ARQ significantly outperforms state-of-the-art strategies PARTIES and CLITE in reducing the tail latency of LC applications and increasing the IPC of BE applications.**

***Index Terms*—Datacenter, high-throughput, performance uncertainty, quality of services (QoS), resource contention.**

I. Introduction

THE capabilities of computers have been formally examined in multiple instances. Alan Turing developed an abstract machine (often referred to as the Turing machine) to prove that not all definable problems are computable [56]. While this was a negative conclusion, it opened the door to research on the capabilities of computers. Popek and Goldberg [45] presented the formal requirements for virtualizable third-generation architectures, which derived precise conditions to test whether a given architecture can support virtual machines. Another example is the CAP theorem proposed by Brewer [10], [22], which states that a distributed database system running on a cluster can only support two out of three properties: Consistency, Availability, and Partition tolerance. This theorem has had a widespread impact on the design of modern distributed systems.

These examples present instructive assertions about different abilities of computers and inspire us to explore a new capability for simultaneously ensuring high quality of service (QoS) and high throughput. In this study, for latency-critical applications, "throughput" is defined as the number of responses completed by a datacenter per second, with each response corresponding to a request. For best-efforts applications, "throughput" is defined as the number of instructions executed per second. Given the widespread deployment of cloud computing, it becomes crucial to construct datacenters capable of ensuring QoS and achieving high throughput, effectively addressing the performance uncertainties associated with cloud computing.

Cloud computing relies on the shared use of a datacenter by multiple concurrent applications to optimize resource utilization. However, this high concurrency can lead to resource contention and degraded user experience [15], [16], [25], [37], [38], [58], [67], [68], making it challenging to achieve both high throughput and user satisfaction. Interference within the datacenter must be carefully monitored, quantified, and addressed. While our conference paper [32] has conducted preliminary research on the measure and suppression of interference in datacenters, the underlying general principles have not been provided.

In our study, we present a theoretical framework for designing a datacenter that can meet the user experience in highly concurrent scenarios while considering the different criticalities of user requests. It is common for concurrent applications to have diverse resource requirements and different levels of criticality. Additionally, users associated with these applications often have varying requirements for tail latency. Therefore, the user experience of a application in a datacenter is influenced by three factors: the user's tail latency requirements, the ideal tail latency in the absence of interference, and the interference caused by co-running applications.

Manuscript received 25 April 2023; revised 27 December 2023; accepted 2 January 2024. Date of publication 16 January 2024; date of current version 18 March 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4503904 and Grant 2016YFB1000201, and in part by the Innovation Funding of ICT, CAS under Grant E361100. Recommended for acceptance by R. Prodan. *(Corresponding author: Yuhang Liu.)*

The authors are with the University of Chinese Academy of Sciences, Beijing 100190, China, and also with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China (e-mail: liuyuhang@ict.ac.cn; dengxin19g@ict.ac.cn; zhoujiapeng22s@ict.ac.cn; cmy@ict.ac.cn; baoyg@ict.ac.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2024.3354418, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2024.3354418

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

Previous research has acknowledged the importance of QoS [47], [66]. However, several key issues have yet to be analytically addressed, including: (1) the conditions necessary for a datacenter to achieve high-throughput, (2) the relationship between tail latency and throughput, and (3) the scalability of a datacenter.

The presence of two types of applications is typical in a datacenter, namely the latency-critical (LC) applications and the best-effort (BE) applications. LC applications, such as Redis [3] and Moses [29], prioritize user experience and are susceptible to tail latency and user expectation. On the other hand, BE applications, like Spark [65] and Fluidanimate [8], are concerned with performance and are measured based on instructions-per-cycle (IPC).

In the pursuit of resource efficiency in a datacenter, it is common practice to simultaneously run multiple applications on the same node. However, this can result in contention for the shared hardware resources, leading to a negative impact on the performance of the applications [11], [12], [15], [25], [37], [50], [61], [63], [64]. The impact of contention can be particularly severe for LC applications since tail latency can significantly affect user experience. Although the effects of interference on BE applications may not be as fatal, minimizing the drop in IPC is still crucial to maintain a satisfactory user experience. Therefore, it is important to consider the relative importance (RI) between LC and BE applications when assessing the impact of interference.

The simultaneous operation of various applications in a datacenter creates a multitude of tail latency or IPC values, which pose a significant challenge in quantifying the exact intensity of the interference in the overall system. This is because these values are calculated at the individual application level and do not provide a comprehensive system perspective. Therefore, the need to holistically quantify and reduce interference in a datacenter is a crucial issue that requires attention. In Section V-C, we will present a detailed example to illustrate this challenge.

Previous research has utilized different approaches to quantify interference in a datacenter, such as the ratio of tail latency over instruction throughput [52], reduced service rate of a virtual machine (VM), and the duration of interference [57], [58]. Although these methods have demonstrated effectiveness in specific cases, they are primarily ad hoc, and their units are not well-defined, rendering them challenging to apply in diverse scenarios.

In this study, we present system entropy (E_S) to quantify the interference within a datacenter. Conventionally, entropy is a physical concept with multiple versions, including thermodynamic entropy and information entropy. The reason we chose to use the term "system entropy" is because the interference in a datacenter involves multiple applications contending for limited resources, similar to the thermodynamic entropy caused by collisions between molecules. To demonstrate the correctness and rationality of system entropy, we adhere to the three-step paradigm of information entropy (see pages 10 and 11 in [49]). Specifically, we first define the required properties of E_S, propose an analytical expression for E_S, and validate that the expression satisfies the required properties.

In this study, we categorize the resource management capabilities of data centers into three aspects (differentiation, isolation, prioritization, as discussed in Sections II and III) and break down the causes of interference into three aspects (resource scarcity, switch overhead, scheduling inappropriateness, as described in Section IV-B). Subsequently, the DIP theorem is proposed to leverage the first three aspects to mitigate the latter three.

The DIP theorem has significant implications for the design of resource isolation or sharing strategies among LC applications, among BE applications, and between LC and BE applications. For example, various state-of-the-art resource managers [13], [17], [26], [27], [33], [34], [40], [41], [42], [43] utilized resource isolation techniques to isolate colocated applications and eliminate resource interference. However, some researchers have shown that such isolation may reduce resource utilization [12], [19], [44]. Furthermore, these methods primarily focus on cache partitioning and are only suitable for BE applications. In this study, we demonstrate that strict isolation often results in reduced resource utilization, and that allowing resources to be flexibly shared or isolated among applications has the potential to effectively mitigate the interference of both LC and BE applications.

In this paper, we propose the Ah-Q toolkit, which includes two theorems, a metric and a scheduling strategy. Specifically, we present the following contributions:

- We formalize a series of frequently used cloud computing concepts, propose the DIP theorem to determine whether a datacenter can guarantee QoS and achieve high-throughput, and propose the TLT theorem to formulate the relationship between tail latency and throughput.
- We introduce E_S, a dimensionless single "figure of merit" for datacenters that can be used to quantify and evaluate interference. To develop E_S, we identify its required properties, propose an analytical expression, and demonstrate that the expression satisfies the necessary properties. We also introduce the concept of "resource equivalence" based on E_S, which can be used to evaluate the effectiveness of different scheduling strategies.
- We design an associative scheduling strategy, ARQ, that leverages detected entropy as feedback to reduce interference. ARQ allows partial resource sharing between LC and BE applications and dynamically adjusts the size of isolated and shared resources. We build a space-time resource utilization model to interpret the advantages of ARQ over previous strategies and to explain the causes of interference. We evaluate ARQ against state-of-the-art strategies, CLITE and PARTIES. Our results show that ARQ has significantly reduced E_S, leading to an overall improvement in user experience and throughput.

In the following, Section II formalizes the key concepts of cloud computing and formally defines the resource management abilities of a datacenter. Section III presents three lemmas to establish the relationship among the abilities. Section IV presents

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

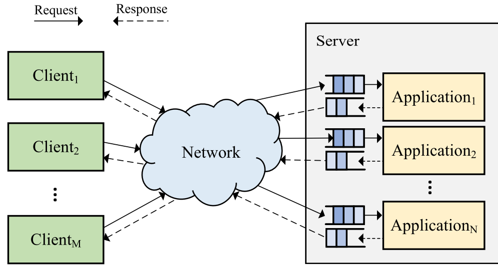


Fig. 1. Service framework of cloud computing.

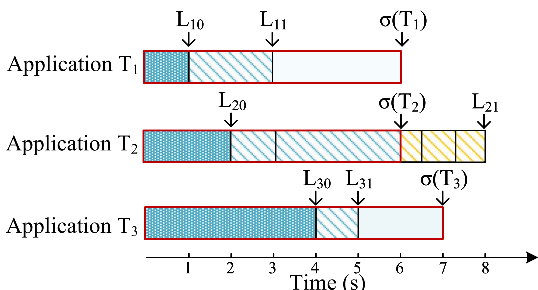


Fig. 2. Example of concurrent applications.

two theorems for managing the interference within a datacenter. Section V proposes the system entropy E_S. Section VI presents the ARQ scheduling strategy. Section VII validates the system entropy E_S. Section IX evaluates the ARQ scheduling strategy. Section X overviews related work, and finally Section XI concludes our study.

II. Formalizing Cloud Computing Concepts

To the best of our knowledge, there is no formalization for the key concepts of cloud computing and the resource management abilities of a datacenter.

The service framework of cloud computing is shown in Fig. 1. We define a datacenter S as an ordered triple (i.e., $S = \langle t, r, c\rangle$), where t represents the set of time-slices $t_1, t_2, \ldots, t_m$, r represents the set of resource-slices $r_1, r_2, \ldots, r_n$, and c represents the set of resource management abilities.

Tail latency refers to the response time of a application i in a datacenter S, running with other co-running applications $\bar{i}$, and is defined as the θth percentile response times, where $\theta\% = P\{L(i,\bar{i},S) \leqslant L_\theta(i,\bar{i},S)\}$. The tail latency of application i is a function of three variables, namely application i, i's co-runners (i.e., $\bar{i}$), and datacenter S, denoted as $L_\theta = L_\theta(i,\bar{i},S)$. The tail latency of application i with no co-runners is referred to as the ideal tail latency, $L_{i0} = L_\theta(i,\varnothing,S)$, while the actual tail latency of application i with co-runners is denoted by $L_{i1} = L_\theta(i,\bar{i},S)$.

Turing's definition of computability assumes that a computer has infinite memory resources and can run indefinitely, which is not applicable in practical scenarios. In order to account for these practical limitations, we introduce the concept of *practical computability*. Specifically, given a set of co-running applications ($\bar{i}$) in a datacenter S, a application i is considered to be Practically Computable (PC) if its tail latency is below a user-defined threshold value (i.e., M_i) that is deemed acceptable for a satisfactory user experience. More formally, we define $(PC(i,\bar{i},S) = 1) \leftrightarrow (L_\theta(i,\bar{i},S) \leqslant M_i)$, where M_i represents the minimum user experience requirement for application i. The predicate $PC(i,\bar{i},S)$ takes values of 0 or 1, indicating the application's practical computability status.

We define *Weighted Number of Inversions (wNoI)* to measure the degree of deviation from an optimal scheduling order in a datacenter, accounting for the differential impacts of interference on different applications. To calculate $wNoI$, we define a permutation π, where $\pi(i)$ represents the position of application i in the scheduling sequence. If $i < j$ and $\pi(i) > \pi(j)$, either the pair of positions (i, j) or the pair of applications ($\pi(i)$, $\pi(j)$) is called an inversion of π. The set of all inversions is referred to as the inversion set, and the number of inversions is denoted by NoI. For each inversion (i, j) in the inversion set, we associate a tail latency addition $w(S, i, j)$, representing the impact of application j taking precedence over application i. Then $wNoI(S,T_1,T_2) = w(S,T_1,T_2) \times NoI(T_1,T_2)$, where $NoI(T_1,T_2)$= 1 or 0 depending on whether T_2 takes precedence over T_1.

A datacenter has the option to choose whether or not to implement resource management abilities. The *Tolerance Ability (TA)* reflects the gap between the standalone performance of a datacenter and the user experience requirement. On the other hand, *Distinguishing Ability (DA)*, *Isolation Ability (IA)*, and *Prioritizing Ability (PA)* reflect abilities to eliminate the interference caused by the co-running applications.

For application i, TA is determined by its user-defined threshold, M_i, and the ideal tail latency, TL_{i0}, which is the tail latency when no interference exists. The anti-interference capacity of application i, represented by X_i, is the interference that it can tolerate and is given by $X_i = M_i - TL_{i0}$, where $i = 1, 2, \ldots, N$. In practice, $X_i > 0$, indicating that application i has some slack that can be used to increase the tail latency without compromising the user experience. As shown in (1), TA of a datacenter is a function of the tolerance abilities of all its applications.

$$TA = \sum_{i=1}^{N} \frac{X_i}{M_i} = \sum_{i=1}^{N} \left(1 - \frac{TL_{i0}}{M_i}\right) \tag{1}$$

Fig. 2 illustrates the values of TL_{i0}, TL_{i1} and M_i for three applications. The TA of a datacenter depends on TL_{i0} and M_i and is not related to TL_{i1}. In this example, the anti-interference capacities X_1, X_2, and X_3 are 5, 4, and 3, respectively, resulting in an overall TA of 12. Applications 1 and 3 have practical computability since their respective TL_{i1} values, TL_{11} and TL_{31}, are less than their M_i values. However, application 2 does not have practical computability because its tail latency, TL_{21}, is greater than its M_2 value.

DA is the ability of a datacenter to differentiate and identify the owners of any resource-slice (r_s) for any time-slice (t_s) within a datacenter. A datacenter with DA=1 is able to identify the owners ($Owners(r_s,t_s)$) of any resource for any application. DA is an important property for ensuring fairness and avoiding

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

resource monopolization. Without DA, a malicious application may intentionally monopolize resources to the detriment of other applications. The ability to identify resource owners enables a datacenter to monitor resource usage and enforce appropriate policies.

IA is a critical property that ensures that a datacenter can strictly isolate resources allocated to different applications. A datacenter is said to have isolation ability if, during a continuous time-slice sequence $\{t_n, t_{n+1}, \ldots, t_{n+i}\}$, a resource-slice r_s can only be accessed by at most one application. It should be noted that the resource adjustment time granularity of IA is a continuous sequence of time-slices. During this sequence of time-slices, even if the application to which the resource is allocated does not utilize the resource, other applications are still unable to use the resource. IA can also be referred to as the non-preemptible exclusivity, which means that the resource cannot be used by any other application during the time-slice sequence.

PA is the ability of a datacenter to prioritize concurrent applications according to a certain criterion. Without any prioritizing ability, the order of the co-running applications is random.

III. Relationship Between DA, IA and PA

In this section, we present three lemmas to elaborate the relationship among the management abilities of a datacenter.

Lemma 1: If a datacenter S has isolation ability, then it must also have distinguishing ability.

Proof. Suppose that datacenter S does not have distinguishing ability. This means that S cannot distinguish which application is using (r_s, t_s). Suppose further that multiple applications, $t_n, t_{n+1}, \ldots, t_{n+i}$, are using the (r_s, t_s) simultaneously in practice. Since S cannot distinguish which application is using the (r_s, t_s), S cannot guarantee $|Owners(r_s, \{t_n, t_{n+1}, \ldots, t_{n+i}\})| \leqslant 1$, which contradicts the assumption that S has isolation ability. Therefore, we have shown that if datacenter S has isolation ability, then it must have distinguishing ability.

Lemma 2: If datacenter S possesses prioritizing ability, then it must also possess distinguishing ability.

Proof: We prove Lemma 2 by contradiction. Let us assume that $DA(S) = 0$, i.e., datacenter S does not have distinguishing ability. In this case, we need to prove that $PA(S) = 0$, i.e., S does not have prioritizing ability. If S cannot identify the applications that are using (r_s, t_s), S cannot ensure that $|Owners(r_s, \{t_n, t_{n+1}, \ldots, t_{n+i}\})| \leqslant 1$, which means that S is unaware when multiple applications are using the (r_s, t_s) simultaneously. Consequently, S cannot prevent this case from occurring, and a less important application would potentially be prioritized by a more critical one.

Lemma 3: The isolation ability (IA) and prioritizing ability (PA) of a cloud datacenter are complementary to each other.

Proof: The proof of Lemma 3 follows from the definition of IA and PA. IA achieves isolation by exclusively allocating a resource-slice to a application for several time-slices, which eliminates the uncertainty of performance of the application and reduces the overhead of resource owner switching. On the other hand, PA achieves prioritization by changing the owner of resource-slices more granularly, adapting to the fluctuating resource requirements of applications in a timely manner. These two mechanisms complement each other, as IA provides stability and predictability to the allocation of resources, while PA provides flexibility and adaptability. Thus, IA and PA are two important and complementary mechanisms for ensuring efficient and effective resource allocation in cloud datacenters.

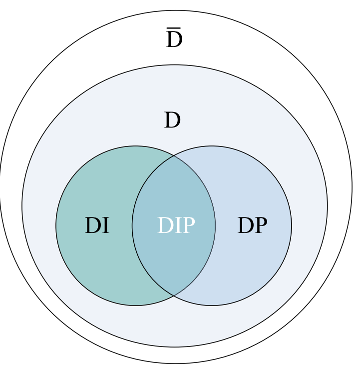


Fig. 3. Venn diagram of the resource management abilities of diverse computers in terms of the DIP.

According to Lemmas 1, 2 and 3, we can categorize datacenters into five different types based on their control abilities, as shown in Fig. 3. The control ability set, c, is progressively expanded by adding DA, IA, and PA to it. The five categories are as follows: (1) $c = \varnothing$, indicating that the datacenter has no control abilities; (2) $c = \{\text{D}\}$, indicating that the datacenter has only DA; (3) $c = \{\text{D, I}\}$, indicating that the datacenter has both DA and IA; (4) $c = \{\text{D, P}\}$, indicating that the datacenter has both DA and PA; and (5) $c = \{\text{D, I, P}\}$, indicating that the datacenter has all three control abilities.

IV. Theorems for Managing Interference

We establish a pair of theorems for managing the interference within a datacenter.

A. The DIP Theorem

Theorem 1: **(DIP Theorem).** *To ensure high-throughput computing and meet the practical computablility (PC) of highly concurrent applications, a datacenter with finite resources has limited TA and therefore must have possess DIP (DA, IA, and PA) abilities to eliminate application interference before TA is exhausted. Specifically,* ❶ *when available resources are plentiful, a datacenter does not require DIP abilities to fulfill the PC of all applications;* ❷ *when resources are limited, PA is necessary to ensure the critical applications' resource requirements are met;* ❸ *when application switching overhead is significant, IA is required to prevent frequent resource switching among applications;* ❹ *when resources are scarce, and application switching overhead is significant, DIP abilities that strikes a*

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

balance between "isolation" and "sharing" to simultaneously reap their benefits can guarantee the PC of critical applications to the maximum degree.

Proof. We decompose the causes of interference into three factors.

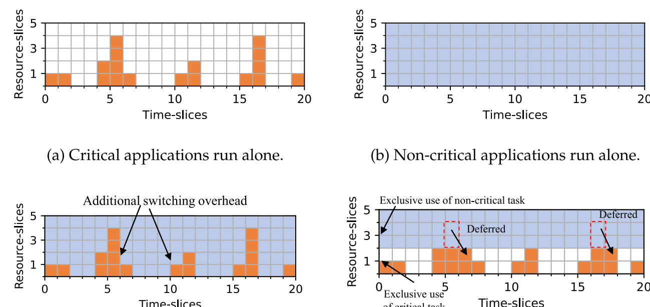
Factor A: *In a datacenter, there is a contradiction between finite resources and the high concurrency of applications.* With multiple applications contending for limited resources, some applications may not receive sufficient resources, leading to reduced QoS. Without IA and PA, colocated applications are allocated resources randomly, making it impossible to guarantee the QoS of critical applications. If the datacenter has PA, it can prioritize resources for high-priority applications. Meanwhile, IA can allocate exclusive resources to critical applications, but due to the non-preemptible exclusivity of IA, it cannot handle fluctuating resource requirements. Therefore, PA is essential to cope with the fluctuating resource requirements of applications when resources are scarce.

Factor B: *The use of resource slices alternately leads to non-negligible resource switching overhead, which causes interference.* When multiple applications are time-division multiplexed on the same resource-slice, or the same application is switched on different resource-slices, the switching overhead of time-slices or resource-slices occurs. This overhead exists widely across various types of resources, such as processing core resources where the context of the application needs to be saved and switched, and shared cache resources where the cache lines need to be refilled [27].

While IA can avoid this switching overhead, PA cannot. Without IA, low priority applications can still occupy idle resource-slices, resulting in switching overhead when high priority applications require those resource-slices. To avoid this interference, IA makes the resource-slices be exclusively used. Therefore, IA is needed to handle the interference caused by resource switching overhead.

Factor C: *Inappropriate resource scheduling can lead to resource waste.* The resource scheduler must take into account various factors, including space-time interleaving of applications, the urgency of the applications, and the overhead caused by switching time-slices or resource-slices. If a application does not fully utilize the isolated resources allocated by the scheduler, it can result in lower resource utilization and wastage. This wastage can occur in various types of resources such as processing cores, cache, memory, and storage. Proper resource scheduling can ensure that the resources are utilized efficiently, which can lead to better performance and cost savings.

Fig. 4 provides an example to illustrate the three interference factors discussed above with respect to the characteristics of IA and PA. Fig. 4(a) and (b) depict the resource requirements of critical and non-critical applications when executed alone, respectively, where the resource requirements of the critical applications are subject to fluctuation. Fig. 4(c) displays the cyberspacetime when these two applications are colocated on a datacenter equipped with PA but lacking IA. Although the available resources cannot satisfy the resource requirements of both applications simultaneously, the resource requirements of



(a) Critical applications run alone.

(b) Non-critical applications run alone.

(c) Applications colocated in system with PA.

(d) Applications colocated in system with IA.

Fig. 4. Cyberspacetime example of two applications running on the datacenter with IA or PA.

the critical application are always fulfilled through PA, effectively addressing interference factor A. However, the space-time interleaving of the resource usage of the two applications causes interference factor B to arise.

Fig. 4(d) depicts the cyberspacetime when the two applications are colocated on a datacenter with IA but not PA. To avoid interference factor B, critical applications are allocated to run on R_1 and R_2, while non-critical applications are allocated to run on R_3, R_4, and R_5 through IA. However, the non-preemptible exclusivity of IA limits the datacenter's ability to cope with the fluctuating resource requirements of applications, which results in the bursty resource requirement of the critical application (e.g., the 6-th time-slice of Fig. 4(d)) being restricted by the amount of isolated resources. This limitation causes some requests to be deferred, resulting in interference factor A. Moreover, some isolated resources allocated to critical applications are not fully utilized, resulting in interference factor C.

In practice, Heracles [34] and PARTIES [13], as well as other resource scheduling strategies, have demonstrated the effectiveness of IA in reducing interference between applications. However, they are unable to schedule resources by priority at the granularity of the time-slice T_s. Consequently, they are unable to address the resource requirements of microsecond-level bursts, and can only reserve enough isolated resources to ensure QoS. On the other hand, the resource scheduling strategy Caladan [21] can meet all of the PA conditions and can schedule shared resources with microsecond-level time granularity to address bursty resource requirements of applications. However, although it includes a scoring mechanism to reduce switching overhead, it does not use IA to avoid switching overhead.

The objective function in the DIP theorem's resource scheduling is system entropy, which will be presented in Section V. Details regarding achieving a balance between "isolation" and "sharing" will be discussed in Section VI.

B. The TLT Theorem

Tail latency is a widely accepted metric for measuring the performance of latency-critical applications. However, as far as we know, there is currently no precise formula describing the relationship between tail latency and throughput. We now

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

present the TLT (acronym for "tail latency and throughput") theorem in the following.

Theorem 2: **(TLT theorem).** *Assume that the latency values that are no more than the tail latency contribute to the average latency by at least $f \cdot 100\%$. For any distribution of request processing latency, the upper bound of the throughput of a datacenter is shown in (2), where RLP is request level parallelism which represents the number of outstanding requests in a datacenter (including requests that are being processed and requests that are queued for processing), φ is the throughput in terms of the number of responses per second delivered by a datacenter, and TL_θ represents the θth percentile tail latency. Therefore, reducing tail latency can effectively increase the throughput.*

$$\varphi \leqslant \frac{RLP}{TL_\theta} \cdot \frac{1-f}{1-\theta\%} \tag{2}$$

Proof: Using Little's Law [31], we can derive (3), where AL represents the average latency of requests.

$$RLP = \varphi \cdot AL \tag{3}$$

Let (Ω, p) be a finite probability space, meaning that Ω is a finite set, and $p = \text{Prob}$ is a mapping from Ω to the interval $[0, 1]$ and satisfies the condition $\sum_{w \in \Omega} p(w) = 1$. A random variable X on Ω is a mapping $X : \Omega \to \mathbb{R}$. We define a probability space $X(\Omega)$ on the image set by setting $p(X = x) = \sum_{X(\omega = x)} p(w)$.

The probability of the latency L being greater than L_θ can be represented by (4).

$$Prob(L \geqslant TL_\theta) = \sum_{\omega:\, L(\omega) \geqslant TL_\theta} p(\omega) \tag{4}$$

For AL, we have (5).

$$\begin{aligned} AL &= \sum_{\omega:\, L(\omega) \geqslant TL_\theta} p(\omega)L(\omega) \;+ \sum_{\omega:\, L(\omega) < TL_\theta} p(\omega)L(\omega) \\ &\geqslant \sum_{\omega:\, L(\omega) \geqslant TL_\theta} p(\omega)L(\omega) \\ &\geqslant TL_\theta \sum_{\omega:\, L(\omega) \geqslant TL_\theta} p(\omega) \end{aligned} \tag{5}$$

Combining (4) and (5), we derive (6).

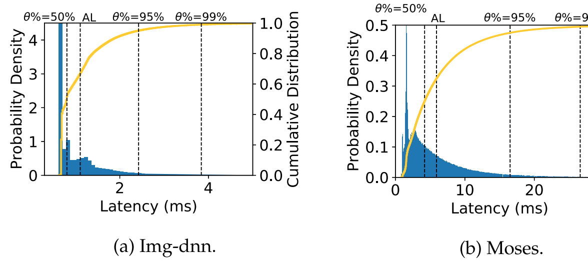
$$Prob(L \geqslant TL_\theta) \leqslant \frac{AL}{TL_\theta} \tag{6}$$

Then the relationship between average latency and tail latency can be expressed as (7).

$$\frac{TL_\theta}{AL} \leqslant \frac{1}{1-\theta\%} \tag{7}$$

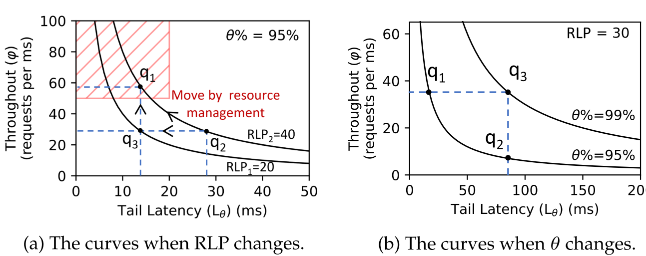
Furthermore, given that the latency distribution of real-world applications tends to be irregular, we divide the latency into multiple intervals as shown in (8).

$$\begin{aligned} AL &= \sum_{\omega:\, L(\omega) \geqslant TL_\theta} p(\omega)L(\omega) + \sum_{\omega:\, L(\omega) < TL_\theta} p(\omega)L(\omega) \\ &\geqslant TL_\theta \cdot Prob(L \geqslant TL_\theta) + \sum_{\omega:\, L(\omega) < TL_\theta} p(\omega)L(\omega) \end{aligned} \tag{8}$$



(a) Img-dnn. (b) Moses.

Fig. 5. Probability density functions and cumulative density functions of requests latency.



(a) The curves when RLP changes. (b) The curves when θ changes.

Fig. 6. Example of little's law extension for tail latency (the x-axis is the tail latency, the y-axis is the throughput of the system, and the upper left corner is the zone of satisfactory solutions).

Since the latency values that are smaller than the tail latency contribute to the average latency by a factor of f, we can derive (9).

$$Prob(L \geqslant TL_\theta) \leqslant \frac{AL - f \cdot AL}{TL_\theta} \tag{9}$$

Therefore, we have (10)

$$\frac{TL_\theta}{AL} \leqslant \frac{1-f}{1-\theta\%} \tag{10}$$

Combining (10) and (3), we can derive (2).

In the field of cloud computing, the term "load" is generally used to represent RLP with the unit being queries per second (QPS). In practical applications, "load" is usually normalized QPS (relative to the max load), presented as a percentage.

Fig. 5 presents the probability density functions and cumulative density functions of request latency, using Img-dnn and Moses as examples. The experimental setup for this analysis will be elaborated in Section VII-A.

Theorems 1 and 2 demonstrate that a high concurrency level does not necessarily result in high throughput. In fact, a high-concurrency datacenter can exist in one of two states: either it has low latency and high throughput, or it has high latency and low throughput. The decision between these two states depends on whether the datacenter has the DIP resource management abilities.

Fig. 6(a) illustrates two possible states of a datacenter that has a high RLP: "low-tail-latency and high-throughput" (represented by q_1) or "high-tail-latency and low-throughput" (represented by q_2).

The tail latency values in a datacenter can vary depending on the level of RLP, even when the throughput remains the same. For instance, in Fig. 6(a), the q_2 point represents a datacenter with high RLP (i.e., 40) and high tail latency, while the q_3 point

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

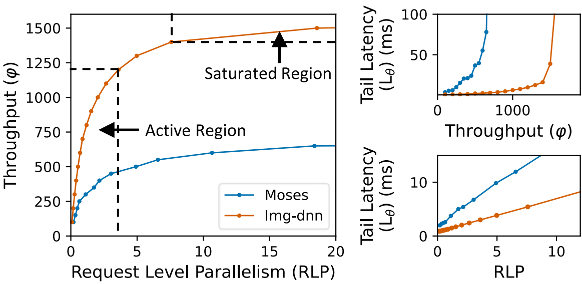


Fig. 7. Scalability of a datacenter (take Moses and Img-dnn applications as example).

represents a datacenter with low RLP (i.e., 20) and low tail latency. It is important to note that as RLP increases it becomes more challenging to achieve the same tail latency in a datacenter. To achieve a high-throughput system, DIP is needed to reduce interference among high-concurrent applications, moving the system state from q_3 to q_1, as illustrated in Fig. 6(a).

Fig. 6(b) displays the Little's Law curves for various percentiles of tail latency, with the q_1 and q_3 points corresponding to the 95th and 99th percentiles of tail latency, respectively. For datacenters with the same RLP and throughput, increasing the value of θ leads to larger tail latency values. The q_2 and q_3 points represent the difference in throughput resulting from more stringent tail latency targets; specifically, more stringent tail latency targets lead to higher throughput.

While cloud computing differs from supercomputing, data centers still face scalability challenges that have yet to be fully defined and addressed. In this study, we define data center scalability as the improvement in throughput with increased RLP.

The $wNoI(Ti)$ metric is used to quantify the impact of interference on tail latency as RLP increases, making it a useful indicator of a datacenter's scalability. By contrast, for a supercomputer, the memory-bounded speedup (also known as Sun-Ni's law [53]) is used to measure scalability as workload size increases. In this case, a function $g(x)$ is used to map available memory size to memory-bounded workload size, and $g(x)$ serves as a scalability indicator for the supercomputer. Fig. 7 illustrates how tail latency and throughput increase with RLP, with the active and linear regions being desirable and the occurrence of the saturated region being what the DIP abilities aim to prevent.

V. The System Entropy (E_S)

In this section, we introduce the system entropy (E_S) to measure interference in a datacenter. We first present the properties that this measure should possess in Section V-A. Next, we propose an analytical definition for E_S in Section V-B. Finally, we summarize the advantages of E_S in Section V-C.

A. The Required Properties of E_S

To ensure that E_S effectively measures the interference in a datacenter, we propose three required properties for this measure.

TABLE I
List of Symbol Abbreviations

Symbol	Description
S	Datacenter
t_i	Time slice i
r_i	Resource slice i
c	The set of resource management abilities
X_i	The anti-interference capacity of application i
RLP	Request level parallelism within a datacenter
m	The number of BE applications within a datacenter
n	The number of LC applications within a datacenter
TL_{i0}	Application i's ideal tail latency
TL_{i1}	Tail latency of application i when it is suffering interference
M_i	Maximum tail latency that application i can tolerate
A_i	Interference tolerance of application i
R_i	Interference that application i suffers
ReT_i	Remaining tolerance of application i
Q_i	Interference that the application i cannot tolerate
$IPC_{solo}(i)$	IPC when application i is running alone
$IPC_{real}(i)$	IPC when application i is suffering
RI	Relative importance
E_{LC}	LC entropy
E_{BE}	BE entropy
E_S	System entropy

❶ E_S should be dimensionless, meaning that it should not have any units (such as time or resource units) and its value should fall between 0 and 1. The closer E_S is to 1, the greater the interference in the datacenter.

❷ E_S should be sensitive to changes in resource amount. Specifically, if the number of available resources in the datacenter increases, E_S should decrease or at least not increase, given a set of co-running applications and a resource scheduling strategy.

❸ E_S should be sensitive to changes in scheduling strategy. If a fixed number of available resources is given, and the scheduling strategy has reduced resource contention among applications, E_S should decrease, given a set of co-running applications.

In the remaining part of this section, we will present the analytical expressions for E_S in three different scenarios within a datacenter. For ease of reference, Table I provides a list of symbol abbreviations used throughout this paper.

B. The Analytical Expression of E_S

The first scenario considers a datacenter where only n different LC applications are running, and no BE application exists. In this case, the system entropy is equivalent to the entropy of the set of LC applications, denoted as E_{LC}. The definition of E_{LC} is as follows.

In a datacenter, each LC application has three fundamental attributes. We consider application i $(i = 1, 2, \ldots, n)$, where TL_{i0} represents the ideal tail latency of application i, i.e., the tail latency when application i is not subject to any interference. TL_{i1} denotes the tail latency of application i when it is under colocation, which may result in interference. Furthermore, M_i is the maximum tail latency that application i can tolerate. It is worth noting that the ideal latency TL_{i0} can be achieved by temporarily allocating sufficient resources to application i using resource isolation technology. We can quantify the interference tolerance of application i using (11).

$$A_i = 1 - \frac{TL_{i0}}{M_i} \tag{11}$$

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

The user-defined target for M_i is influenced by various factors and is considered to be only of reference significance. Users determine the M_i value based on two principles - the more critical the application is, the lower the tail latency threshold, and the value is usually chosen from the flat to small-slope region. As a result, the target has some flexibility, and in this study, we assume that the relative elasticity of M_i is 5%.

The value of A_i in (11) lies in the range of $[0, 1]$, since TL_{i0} is always no more than M_i. As M_i decreases, the value of A_i approaches 0, indicating a lower tolerance for interference by the application. Conversely, as M_i increases, A_i approaches 1, indicating a higher tolerance for interference. To quantify the interference experienced by application i, we use R_i in (12).

$$R_i = 1 - \frac{TL_{i0}}{TL_{i1}} \tag{12}$$

As TL_{i0} is no more than TL_{i1}, the range of R_i falls between 0 and 1. If TL_{i1} is smaller, then R_i approaches 0, indicating minimal interference to the application, and vice versa. To denote the remaining tolerance of application i after being interfered, we use ReT_i as expressed in (13).

$$ReT_i = \left(A_i > R_i \ ? \ 1 - \frac{TL_{i1}}{M_i} : 0 \right) \tag{13}$$

(14) introduces Q_i as a measure of the interference that an application i cannot tolerate. If the interference suffered by the application (R_i) is greater than its interference tolerance (A_i), then Q_i is computed as 1 minus M_i/TL_{i1}. On the other hand, if the interference suffered by the application is smaller than or equal to its interference tolerance, then Q_i is set to 0. By definition, the value of Q_i lies between 0 and 1. The closer Q_i is to 1, the more severe the interference that the application i cannot tolerate.

$$Q_i = \left(R_i > A_i \ ? \ 1 - \frac{M_i}{TL_{i1}} : 0 \right) \tag{14}$$

The values of A_i, R_i and Q_i introduced above have motivated us to propose a resource scheduling strategy called ARQ, which will be presented in Section VI. In addition, we define E_{LC} as the interference that the LC applications are unable to tolerate, which is represented by (15).

$$E_{LC} = \frac{1}{n} \sum_{i=1}^{n} Q_i \tag{15}$$

In the second scenario where only m different BE applications are running, but no LC application exists in the datacenter, we define E_S as the BE entropy (E_{BE}). E_{BE} is defined as the slowdown incurred by the interference that the BE applications have suffered. The E_{BE} is expressed as shown in (16), where $IPC_{solo}(i)$ denotes the IPC when the BE application i runs alone and $IPC_{real}(i)$ denotes the IPC when the BE application i suffers interference.

$$E_{BE} = 1 - \frac{m}{\sum_{i=1}^{m} \frac{IPC_{solo}(i)}{IPC_{real}(i)}} \tag{16}$$

When all BE applications are free from interference, E_{BE} is 0. The closer the ratio of $IPC_{solo}(i)$ over $IPC_{real}(i)$ is to 1 for each BE application i, the lower the interference level, and the closer the value of E_{BE} is to 0. Conversely, the higher the interference level for any BE application i, the larger the ratio of $IPC_{solo}(i)$ over $IPC_{real}(i)$ becomes, and the closer the value of E_{BE} is to 1.

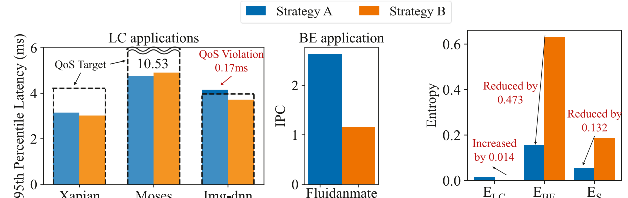


Fig. 8. Tail latency of the LC applications, IPC of the BE application and the entropy values under resource scheduling strategies A and B. The dotted box represents the QoS target of the LC applications.

The third scenario involves the coexistence of LC and BE applications in a datacenter, and in this case, the E_S is expressed as a linear combination of E_{LC} and E_{BE}, as shown in (17). The relative importance (RI) is introduced to determine the weight of each component in the combination.

$$E_S = RI \times E_{LC} + (1 - RI) \times E_{BE} \tag{17}$$

The rationale behind (17) is to minimize both E_{LC} and E_{BE} simultaneously to obtain the minimum E_S. Normally, the value of RI ranges from 0 to 1. However, when there are insufficient resources, minimizing E_{LC} takes priority over minimizing E_{BE}. In such cases, the range of RI is adjusted to [0.5, 1].

It is worth noting that Scenario 1 and 2 are the two extreme cases of Scenario 3. When the datacenter only runs BE applications, the system entropy only needs to consider E_{BE}, and therefore RI is set to 0. This is a common scenario in traditional high-performance computing environments. Conversely, when the datacenter only runs LC applications, RI is set to 1. The larger the value of RI, the higher the priority of the LC applications over that of the BE applications. The value of RI can be determined by datacenter managers based on various factors such as the criticality of LC applications, fairness among all applications, and economic benefits. In this study, we set RI to 0.8, which is representative and captures the trade-off between LC and BE applications.

In our current model, all LC applications are treated equally, and so as BE applications. The reason is that we focus on the criticality difference between LC and BE applications. However, if necessary, the E_S model can be extended to involve different RI factors among the same type of applications.

C. The Advantages of E_S Over Other Metrics

We demonstrate the superiority of the proposed E_S over tail latency and IPC metrics using a simple example. Fig. 8 depicts the tail latency, tail latency threshold of the LC applications, and IPC of the BE applications under two different strategies (i.e., A and B). With the information presented in Fig. 8, it is not easy to determine which strategy is superior. However, the E_S metric allows us to do so precisely and reasonably. The E_S

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

metric offers several advantages over traditional metrics like tail latency and IPC, which will be discussed in the following.

First, E_S provides a concise representation of the overall system performance, making it easy for datacenter managers to compare different strategies and make decisions accordingly. In contrast, using tail latency and IPC separately can lead to a complex and cumbersome analysis, particularly in large-scale datacenters with numerous applications [23]). For instance, in the example presented in Fig. 8, even though there are only a few applications, analyzing the performance of each application under different strategies requires examining multiple performance metrics (i.e., the tail latency and the target threshold of each LC application, and the IPC of each BE application) simultaneously, which can be time-consuming and error-prone.

Second, E_S provides a more comprehensive reflection of the overall user experience of many colocated applications. When the resource scheduling strategy changes, it may improve the performance of some applications but degrade the performance of others. With IPC and tail latency, it is difficult to determine whether the overall user experience of a datacenter is improved or not. It is worth noting that QoS guarantee does not require reducing E_{LC} to zero, and a small E_{LC} is tolerable. The definition of E_S takes this into account. In the example, strategy A is not inferior to strategy B because the QoS violation in strategy A is tolerable. The QoS violation of the LC application (Img-dnn) in strategy A is small (i.e., 4.4%), which is less than the elasticity of the tail latency threshold (i.e., 5%), and the IPC improvement of the BE application (Fluidanimate) is significant (from 1.15 to 2.63, that is 128.7%). Therefore, it is more reasonable to prefer strategy A over strategy B.

Third, E_S can be used to define resource equivalence. In a datacenter, increasing available resources is challenging due to budget and power constraints [59]. Therefore, it is vital to focus on improving resource usage and increasing utilization. E_S can be used to evaluate the effectiveness of a scheduling strategy in terms of resource saving. Specifically, when comparing two scheduling strategies, we can evaluate their resource savings by achieving the same "overall user experience". We can say that a scheduling strategy p_1 is inferior to p_2 if p_1 requires more resources than p_2 to achieve the same E_S. If the amount of resources used by p_2 is R and p_1 uses ΔR more resources, then $E_S(p_1, R + \Delta R) = E_S(p_2, R)$, and ΔR is the resource equivalence of strategy p_2 relative to p_1. This allows for a more comprehensive evaluation of the efficiency of scheduling strategies and can help identify the most resource-efficient options.

VI. The ARQ Scheduling Strategy

In this section, we propose a new scheduling strategy called ARQ, which aims to combine the benefits of resource sharing and resource isolation to minimize system entropy. The name "ARQ" is an acronym for the three essential factors of an LC application: A_i, R_i and Q_i, as discussed in Section V-B. By leveraging these factors, ARQ strives to maximize resource utilization while maintaining a low level of interference among applications. We present the allocation and overhead comparisons over other strategies in Section 2.2 of the separate supplemental file.

A. Demonstrating the Key Insight Via a Space-Time Model

The observation has been made that resource isolation can reduce performance uncertainty, while resource sharing can increase resource utilization and overall throughput. Therefore, in order to minimize E_S, we propose to exploit the combination of both resource isolation and resource sharing.

The existing resource scheduling strategies, such as those presented in references [12], [13], [21], [34], [43], have utilized resource isolation techniques to ensure QoS. This means that each application is assigned its own dedicated resources and cannot use the resources assigned to other applications. However, this approach often results in low resource utilization due to the underutilization of resources assigned to idle applications.

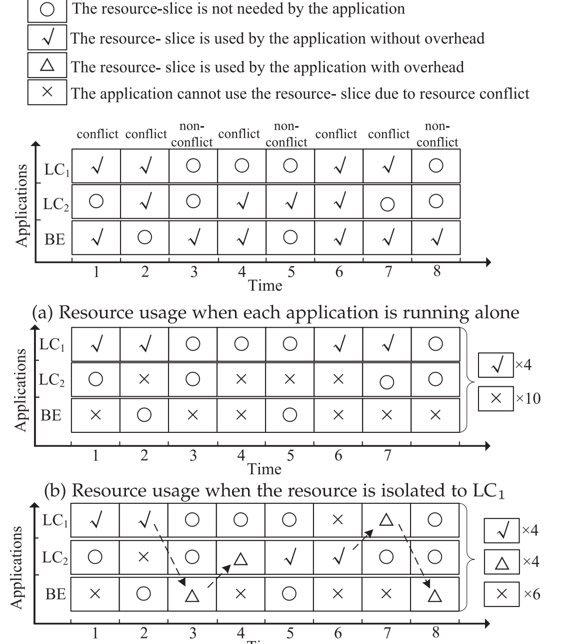
Take the processing unit resources as an example. We assume that when the datacenter can provide a service rate of at least U, the QoS target of the LC applications can be satisfied. We also assume that one core can provide a service rate of $0.8U$, and two cores can provide a service rate of $1.6U$. If only one core is allocated to the LC application, its service rate will be lower than the required minimum of U, resulting in violation of the QoS target. On the other hand, allocating two cores to the LC application would meet its QoS target, but it would lead to wastage of resources and hence reduce the throughput of the BE application. Therefore, a trade-off needs to be made between meeting the QoS target of the LC application and maximizing the resource utilization of the data center. This is where the ARQ strategy comes into play by combining resource sharing and resource isolation techniques to achieve the optimal balance between QoS and resource utilization.

Fig. 9 illustrates a space-time model that demonstrates various resource scheduling schemes. The model considers two LC applications (i.e., LC_1 and LC_2) and one BE application (i.e., BE) using only one resource-slice (e.g., one processing unit or one LLC way) and eight time-slices. Three different scenarios are examined. In scenario (a), each application runs alone, enabling us to determine the space-time resource requirement of each application. When there are two or more ticks in a time-slice, resource contention occurs. For example, in time-slice 6, all three applications require the same resource-slice, resulting in resource conflict.

In scenario (b), the resource-slice is isolated and exclusively allocated to LC_1, ensuring that only LC_1 can use the resource-slice and meeting the QoS target of LC_1. However, during some time-slices (e.g., time-slice 3), the resource-slice is not needed by LC_1, but other applications that require the resource-slice cannot use it, resulting in resource waste.

In scenario (c), all applications share the resource, but LC applications are given priority over BE applications. At the start of time-slice 3, the ownership of the resource is transferred from LC_1 to BE, which boosts the throughput of BE. However, this transfer incurs a cost in terms of context switching overhead and/or cache pollution. The triangle in the figure represents the performance boost obtained through the use of the resource,

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.



(c) Resource usage when all the applications share the resource and the LC applications take precedence over the BE applications

Fig. 9. Illustration of the space-time model (for brevity, only one resource slice and eight time-slices are considered).

with the associated overhead. At the beginning of time-slice 4, the resource owner is transferred from BE to LC_2, improving the QoS of LC_2.

Comparing scenario (c) with scenario (b), we can observe that the number of crosses is reduced from 10 to 6, and there are four more triangles in scenario (c). As a result, the resource utilization ratio has been almost doubled. The key insight here is that while resource isolation can reduce performance uncertainty, resource sharing is essential for improving system utilization. Therefore, neither complete isolation nor sharing is the optimal strategy for enhancing the overall user experience. Instead, it is necessary to simultaneously leverage the benefits of both isolation and sharing to achieve optimal resource utilization and performance.

B. Design of the ARQ Strategy

The ARQ (Adaptive Resource sharing and isolation for QoS) strategy divides resources into two regions: shared and isolated. These regions include a number of cores and cache ways. LC applications have access to both their own isolated region and the shared region, while the BE application can only run in the shared region. The key idea is to simultaneously harvest the benefits of resource sharing and isolation, as neither approach is optimal on its own.

If an LC application running in the shared region meets its QoS target, its isolated region's resources will gradually decrease to 0, indicating that it can safely share resources with other applications. However, if an LC application's QoS is severely impacted while running in the shared region, the ARQ strategy detects this interference and gradually increases its isolated

Algorithm 1: ARQ Resource Scheduling Algorithm.

1: **function** ARQ
2: $isAdjust \leftarrow$False, $E_S \leftarrow 1$
3: **while** True **do**
4: Monitor the tail latency values of the LC applications and the IPC values of BE applications periodically
5: $E'_S \leftarrow E_S$
6: $E_S \leftarrow$ computeEntropy()
7: // ReT is an array, the elements of which are the remaining tolerance of each LC application.
8: $ReT \leftarrow$ computeRemainingTolerance()
9: **if** $isAdjust$ and $E_S > E'_S$ **then**
10: Cancel the last adjustment and do not allow the last victim region to be penalized in the next 60s.
11: $isAdjust \leftarrow$ False
12: **else**
13: $isAdjust \leftarrow$ AdjustResource(ReT)
14: **end if**
15: **end while**
16: **end function**
17:
18: **function** adjustResource
19: $victimRegion \leftarrow$ findVictimRegion(ReT)
20: $beneficiaryRegion \leftarrow$ findBeneficiaryRegion(ReT)
21: // Choose one type of the resources (i.e., core, LLC, or memory bandwidth, etc) of $victimRegion$.
22: $\Delta R \leftarrow$ findVictimResource($victimRegion$)
23: Move one unit resource of type ΔR from the $victimRegion$ to the $beneficiaryRegion$
24: **return** whether the resource has been actually adjusted
25: **end function**
26:
27: **function** FINDVICTIMREGION
28: **for** each ReT_i in descending order **do**
29: **if** $ReT_i > 0.1$ and application i has isolated resource that allows to be penalized **then**
30: **return** the isolated region of application i
31: **end if**
32: **end for**
33: **return** the shared region
34: **end function**
35:
36: **function** FINDBENEFICIARYREGION
37: Identify the application i that has the smallest ReT.
38: **if** $ReT_i < 0.05$ **then**
49: **return** the isolated region of application i
40: **else**
41: **return** the shared region
42: **end if**
43: **end function**

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

region's resources until the QoS target is met. In this way, the ARQ strategy dynamically adjusts resource allocation based on application QoS requirements to optimize resource utilization while maintaining QoS guarantees.

The ARQ strategy is described in Algorithm 1. It involves periodic monitoring of the tail latency of each LC application and the IPC of each BE application to calculate the ReT of each LC application and E_S. Based on these calculations, ARQ adjusts resource allocation and evaluates the effectiveness of the adjustment by E_S. If the adjustment increases E_S, it is canceled and a new adjustment action is taken. To avoid getting trapped in a local optimum, the old adjustment is not allowed to occur again in the next 60 seconds. The monitoring period can be set to 500ms [13], 1s [40], or 2s [43]), depending on the specific implementation.

In the *AdjustResource* function of Algorithm 1, the aim is to transfer one resource slice from a rich region to a poor region in the hope of reducing E_S. To achieve this, we first determine the victim and beneficiary regions using the *findVictimRegion* and *findBeneficiaryRegion* functions based on the ReT array that records the ReT of each LC application. Then, using the *findVictimResource* function, we identify which type of resources should be moved or penalized. Finally, we transfer the selected resource from the victim region to the beneficiary region.

The *findVictimResource* function uses a finite state machine to determine the order in which to adjust resources, which is the same approach used in [13]. Each state in the machine represents a type of resource, such as processing cores, LLC capacity, or memory bandwidth. If the current resource type cannot be penalized, the function moves to the next type until it finds a type that can be penalized.

The function *findVictimResource* determines the victim region that will donate resources to other regions. It takes the ReT array as input and traverses it in descending order to identify the application with a ReT larger than 0.1. This is because an application with a high ReT may not have isolated resources. If no isolated region satisfies the requirements, the shared region will be returned.

Then, the *findBeneficiaryRegion* function takes the ReT array as input and outputs the beneficiary region which receives resources from the victim region. We only need to consider the application with the smallest ReT. If its remaining tolerance is less than 0.05, the isolated region of the application will become the beneficiary region. If all the LC applications have high ReT, meaning their ReT values are larger than 0.1, the shared region will become the beneficiary region. The thresholds of 0.05 and 0.1, established through experimental analysis, consistently exhibit robust performance in our various experiments.

If both the victim and beneficiary regions are shared regions, then no LC application requires additional resources or can donate resources; that is, the resource allocation has already reached an equilibrium. In this case, no further resource adjustments will be made.

In ARQ, the monitoring interval is set to 500ms, which is similar to that used in the PARTIES system (as described in Section 4.3 of [13]). Smaller intervals can enable the scheduler to detect and respond to QoS violations more quickly, but they can also make tail latency less stable and make it more challenging to accurately calculate tail latency. On the other hand, larger intervals can make tail latency calculations easier, but QoS violations can last for longer periods. We found that a monitoring interval of 500ms is a practical choice based on experimental evaluations.

TABLE II
DETAILS OF THE LC, BE AND SYSTEM ENTROPY UNDER THE UNMANAGED STRATEGY WITH DIFFERENT NUMBERS OF PROCESSING UNITS

Cores	Applications	TL_{i0}	TL_{i1}	M_i	A_i	R_i	ReT_i	Q_i	E_{LC}	E_{BE}	E_S
6	Xapian	2.77	23.99	4.22	0.34	0.88	0	0.82	-	-	-
	Moses	2.80	16.54	10.53	0.73	0.83	0	0.36	-	-	-
	Img-dnn	1.41	14.35	3.98	0.65	0.90	0	0.72	-	-	-
	System	-	-	-	0.57	0.87	0	-	**0.64**	**0.20**	**0.55**
7	Xapian	2.77	7.13	4.22	0.34	0.87	0	0.40	-	-	-
	Moses	2.80	6.78	10.53	0.73	0.61	0.36	0	-	-	-
	Img-dnn	1.41	5.65	3.98	0.65	0.59	0	0.29	-	-	-
	System	-	-	-	0.57	0.75	0.12	-	**0.23**	**0.03**	**0.19**
8	Xapian	2.77	4.18	4.22	0.34	0.34	0.01	0	-	-	-
	Moses	2.80	4.43	10.53	0.73	0.37	0.58	0	-	-	-
	Img-dnn	1.41	3.53	3.98	0.65	0.60	0.11	0	-	-	-
	System	-	-	-	0.57	0.44	0.23	-	**0**	**0.02**	**0**

TABLE III
EXPERIMENTAL PLATFORM

Component	Specification
CPU	Intel Xeon E5-2630 v4 (10 cores)
Processor Core Frequency	2.2GHz
Operating System	CentOS 7 (kernel 5.6.11)
L1 Caches	32KB×10, 8-way set associative, split D/I
L2 Caches	256KB×10, 8-way set associative
L3 Caches	25MB, 20-way set associative
Main Memory	16GB×7, 2400MHz DDR4
NIC	Intel Corporation I350 Gigabit Network Connection (1Gbps)

VII. VERIFICATION OF E_S

In this section, we conducted experiments to verify that the analytical expression of E_S satisfies all the necessary properties listed in Section V-A. The "dimensionless" property is straightforward to prove. We will focus on the other two properties, namely the "resource-sensitive monotonicity" and "strategy-sensitive monotonicity" properties.

A. Experimental Setup

In our experiments, we used a real server in a datacenter as our platform, as shown in Table III. We utilized the *taskset* command to set the core affinity for each application, and Intel's Cache Allocation Technology (CAT) [6], [24] to allocate the Last-Level Cache (LLC) for each core. CAT enables the assignment of a specific number of ways to each application, limiting the amount of LLC space it can occupy. To ensure consistency with previous studies [13], we disabled Hyper-Threading during the experiments. We evaluated the scheduling strategies using multiple combinations of LC and BE applications from different domains.

Xapian is a widely used search engine that we utilized in our experiments. To create the search index, we used a dump of the English version of Wikipedia and selected query terms randomly following a Zipfian distribution [7], [20]. **Moses**, another LC

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

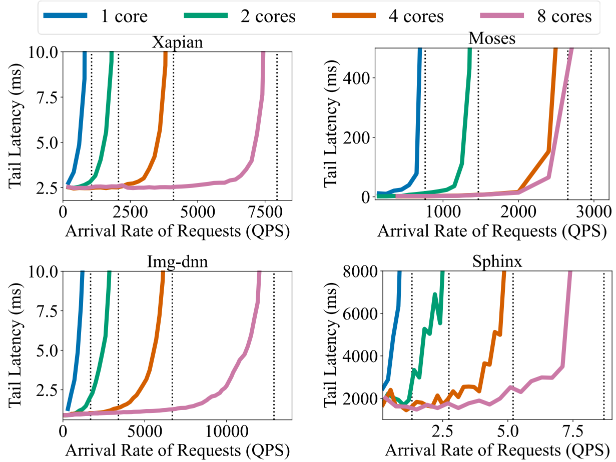


Fig. 10. Relationship between tail latency and arrival rate of requests with 1, 2, 4 and 8 processing units (the dashed lines denote the maximal service rate under varying core counts).

application we used, is a statistical machine translation system. To drive Moses, we randomly selected dialogue snippets from the English-Spanish corpus [54]. We also employed **Img-dnn**, a handwriting recognition application, and selected samples randomly from the MNIST database [18] to drive it. Masstree [35], a scalable in-memory key-value store, was also used in our experiments. We drove Masstree using a modified version of the Yahoo Cloud Serving Benchmark [14], [28]. Finally, we used **Sphinx** [60], an accurate speech recognition system, and **Silo** [55], an in-memory transactional database. These LC applications are from Tailbench [28] and are instantiated with 4 threads.

We present an example to show how we determine the maximum load that each LC application can tolerate. We select 4 LC applications (i.e., Xapian, Moses, Img-dnn and Sphinx), run each application with different number of processing units, gradually increase their arrival rate of requests, and measure the corresponding tail latency. In this study, the 95th percentile tail latency is used without losing generality. Fig. 10 shows the results, with different colored lines corresponding to different numbers of processor cores (1, 2, 4, and 8). For each LC application, the tail latency increases slowly at the beginning as the arrival rate of requests gradually increases. However, when the arrival rate exceeds a certain threshold, the tail latency increases exponentially. Similar to previous research [13], [43], we refer to the tail latency at the load threshold as *tail latency threshold*, which also means the maximum tail latency that an application can tolerate (i.e., the M_i in (11)), and refer to the load threshold as *max load*, which means the maximum load that an application can sustain under a reasonable tail latency target. Table IV summarizes the max load and the tail latency threshold for each application.

We run different BE applications in our experiments. **Fluidanimate** and **Streamcluster** are taken from the PARSEC benchmark suite [8]. Fluidanimate simulates the behavior of a liquid using computational methods to solve the Navier-Stokes equation. Streamcluster solves the online clustering problem.

TABLE IV
PARAMETER OF THE LC APPLICATIONS

	Xapian	Moses	Img-dnn	Masstree	Sphinx	Silo
Tail Latency Threshold (ms)	4.22	10.53	3.98	1.05	2682	1.27
Max Load (QPS)	3400	1800	5300	4420	4.8	220

Similar to the LC applications, Fluidanimate and Streamcluster are both instantiated with 4 threads. **Stream** [39] is a memory-intensive benchmark that performs computation on a large array that cannot fit in the LLC. To generate severe interference to other applications on the processing cores, LLC, and memory bandwidth, we instantiated Stream with 10 threads.

In addition to the proposed ARQ, we will evaluate the following scheduling strategies using the theory of system entropy and resource equivalence.

Unmanaged: This strategy does not distinguish between LC and BE applications, and relies on the default scheduling strategy of the operating system (i.e., Linux's Completely Fair Scheduler), and does not use any isolation mechanism.

LC-first: This strategy prioritizes LC applications by setting them to real-time priority and using the operating system's round-robin scheduling strategy. When a real-time process becomes ready, if the current core is running a non-real-time process, the real-time process immediately preempts the non-real-time process.

PARTIES [13]: This strategy leverages hardware and software resource partitioning technology to dynamically adjust resource allocations between colocated applications. It strictly partitions resources between applications without resource sharing and calculates the slack of multiple LC applications during a fixed time interval to determine whether resources need to be upsized or downsized. This ensures that the QoS targets of the LC applications are not violated.

CLITE [43]: This strategy is also based on resource isolation. It uses Bayesian optimization to identify or predict desirable resource allocations. It builds a predictive model for different resource partitioning configurations by sampling several points in the large configuration space.

B. Resource-Sensitive Monotonicity of E_S

This experiment is to investigate how E_S changes as the number of available resources varies. Table II presents the values of E_{LC}, E_{BE} and E_S of Unmanaged, obtained while running one BE application (Fluidanimate) and three LC applications (Xapian, Moses, and Img-dnn at 20% of max load) concurrently on 6-8 processor cores and all LLC ways. The ideal tail latency TL_{i0} and the maximum tail latency that an application can tolerate (M_i) are constant values, measured with enough resources, and hence the interference tolerance A_i remains unaffected by the number of available resources. When only 6 processor cores are available, the real tail latency TL_{i1} of all three applications is higher than M_i, leading to ReT_i being equal to 0. However, when more processor cores are made available, ReT_i increases to 0.23, indicating a high remaining tolerance of the system and

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

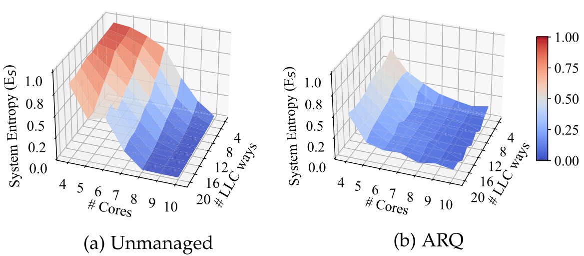


Fig. 11. Impact of the size of available resources on E_S (Xapian (20%), Moses (20%), Img-dnn (20%), Fluidmanate).

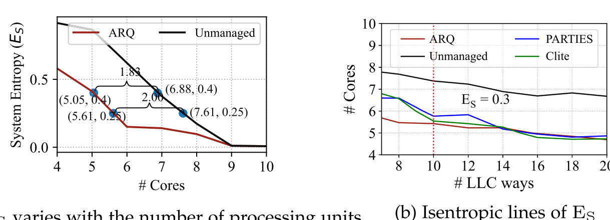


Fig. 12. Illustration of the concept of "resource equivalence".

the presence of redundant resources that can be used to handle additional requests.

When resources are scarce, such as when there are only 7 processor cores available, the E_{LC} value is high at 0.23. If the number of available processing cores is reduced to 6, the tail latency will deviate significantly from M_i, and E_{LC} will increase to 0.64. However, if the number of processor cores is increased to 8, the interference among applications will be reduced to a level that is tolerable for the applications (i.e., $(\forall i, R_i < A_i)$), and E_{LC} will become 0 at this point.

Fig. 11 depicts the variation in E_S of Unmanaged and ARQ strategies with respect to the number of available processing cores (ranging from 4 to 10) and the number of LLC ways per set (ranging from 4 to 20). As the number of available resources decreases for both strategies, E_S shows an increasing trend, which confirms the second property of E_S. In cases where the number of resources is sufficient (e.g., 10 processing cores, 20 LLC ways), even with the Unmanaged strategy, the interference among applications is minimal, resulting in E_S of only 0.006. However, when the number of resources is insufficient (e.g., 6 processing cores, 20 LLC ways), resource contention becomes severe, leading to a high E_S of 0.53. For the ARQ strategy, when the resources are sufficient (e.g., 10 processing cores, 20 LLC ways), E_S is 0.008. However, when the number of resources is inadequate (e.g., 6 processing cores, 20 LLC ways), the E_S of the ARQ strategy rises to 0.15.

C. Strategy-Sensitive Monotonicity of E_S

Fig. 12(a) demonstrates the concept of resource equivalence between two scheduling strategies, Unmanaged and ARQ. The x-axis represents the total number of available processing cores, while the y-axis corresponds to the E_S value. In order to achieve an E_S of 0.25, the Unmanaged strategy requires 7.61 cores, whereas the ARQ strategy only requires 5.61 cores. Therefore, the resource equivalence of the ARQ strategy relative to the Unmanaged strategy is the two-core resource saved. Similarly, when the E_S value is 0.4, the resource equivalence is 1.83 cores.

Fig. 12(b) presents isentropic lines of different scheduling strategies at E_S=0.3. Each line represents the number of processing cores (y-axis) and LLC ways (x-axis) required to achieve the same E_S. When there are more than 10 LLC ways (the right side of the red dashed line), the isentropic lines of ARQ, CLITE, and PARTIES are close to each other, indicating a similar resource equivalence (i.e., resource equivalence R is close to 0). However, when the number of available LLC ways is less than 10, the available resources are scarce, and resource contention is severe. In such cases, ARQ requires much fewer processing cores to achieve the same E_S. For example, when 8 LLC ways are available, compared to PARTIES and CLITE, ARQ saves one processing core, resulting in a resource equivalence of 1 processing core (i.e., 12.5% processing cores). Similarly, using ARQ instead of the Unmanaged strategy results in a resource equivalence of 2 processing cores when 8 LLC ways are available (i.e., 25% processing cores are saved).

TABLE V
CONFIGURATION OF THE EXPERIMENTAL PLATFORM

CPU	Intel® Xeon® Gold 6278C @ 2.60 GHz (26 cores)
L1 Caches	32 KB, 8-way set associative, split D/I
L2 Caches	1 MB, 17-way set associative
L3 Caches	35.75 MB, 11-way set associative
Memory	128 GB, DDR4 2933 MHz
NIC	Intel Corporation Ethernet Connection X722 (1Gbps)
OS	CentOS Linux release 7.6.1810

TABLE VI
LC AND BE WORKLOADS

Latency-critical (LC) workloads	
MySQL [62]	MySQL is a relational database management system (RDBMS). We generate a mix of reads and writes requests by SysBench [25].
Nginx [46]	We use Nginx as a web server. We use wrk2 [53] benchmark to generate query request.
Redis [44]	Redis is an in-memory key-value store. We generate a mix of PUTS and GETS request with the memtier benchmark [45].
Best-efforts (BE) workloads	
Fludanimate (FA)	Fluid dynamics for animation with Smoothed [2].
Blackscholes (BS)	Option pricing with Black-Scholes Partial Differential Equation (PDE) [2].
Stream (ST)	Memory bandwidth benchmark [2].
Bodytrack (BT)	Body tracking of a person [2].
Ferret (FR)	Content similarity search server [2].
Raytrace (RT)	Real-time raytracing [2].
Swaption (SW)	Pricing of a portfolio of swaptions [2].

VIII. VALIDATION OF THE DIP THEOREM

We conducted experiments on a datacenter to explore the impact of IA and PA on user experience. The platform and the applications are described in Tables V and VI, respectively. In each round of the experiment, two applications were selected to be colocated, resulting in a total of 90 different combinations. The colocated applications were scheduled to 8 processor cores, using three different strategies: Shared, Even, and RR. Each application combination was run with each of these strategies,

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

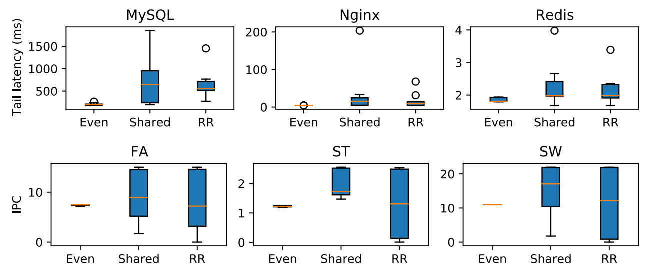
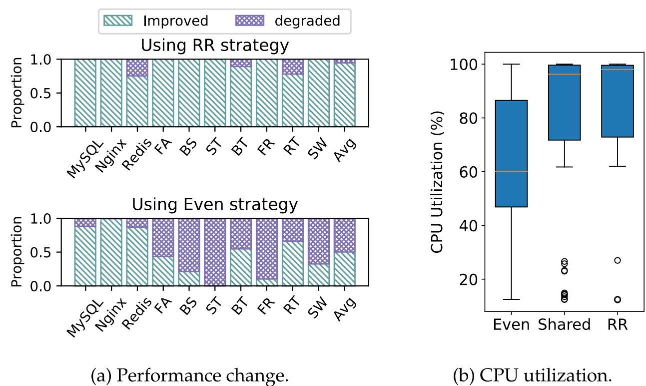


Fig. 13. Tail latency of LC application and IPC of BE application with isolation, sharing, priority (high priority for LC application and low priority for BE application) strategy.



(a) Performance change. (b) CPU utilization.

Fig. 15. Performance change and CPU utilization with different strategies.

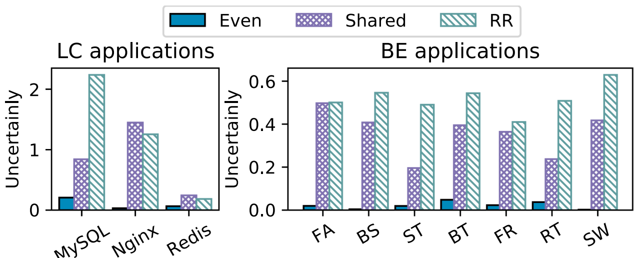


Fig. 14. Performance uncertainty with different strategies.

resulting in 270 sets of data. The three strategies are implemented as follows. The **Shared Strategy** does not implement any IA or PA. Instead, it relies on Linux's Completely Fair Scheduler (CFS) to allocate processor cores to colocated applications. The **Even Allocation Strategy** implements IA but does not have PA. This strategy evenly allocates processor cores to each application to minimize interference. On the other hand, the **Round-Robin (RR) Strategy** has PA but not IA. It uses the operating system's real-time strategy, SCHED_RR, to schedule high-priority applications in a round-robin fashion, ensuring that they receive a fair share of processor time without interference from low-priority applications.

We made the following observations from our experiments:

Observation 1: IA can reduce the performance uncertainty of an application, whereas PA cannot.

Fig. 13 shows that the Even Allocation strategy with IA has a smaller distribution range of the performance of all applications compared to the Shared strategy without IA. This suggests that the Even Allocation strategy can reduce the performance differences among colocated applications. On the other hand, for the RR strategy without IA, even if an application has a higher priority, it may still have large performance variation caused by different colocated applications.

To quantify the variation, we define uncertainty as the ratio of standard deviation (σ) and mean performance ($\overline{Perf}$), where mean performance is in terms of the average tail latency for LC applications and the average IPC for BE applications when the application is colocated with others. Fig. 14 illustrates the uncertainty of each application with different strategies. We observe that the Even strategy with IA significantly reduces the uncertainty of application performance compared to the Shared or RR strategies.

Observation 2: PA can improve the performance of more critical applications, but IA cannot.

Fig. 13 shows that the tail latency of LC applications with RR strategy is mostly lower than that with the Shared strategy, and the IPC of the BE applications with RR strategy is mostly lower than that with the Shared strategy. This is because in the RR strategy, high-priority applications can immediately preempt low-priority applications on a processor core, ensuring that the resource requirements of high-priority applications are met. This is not possible in the Shared or Even Allocation strategies, where there is no prioritization mechanism.

IA does not always lead to better application performance and can sometimes even lead to worse performance. For instance, in the case of the Redis application, the lowest tail latency is achieved with the Shared strategy. When Redis is colocated with MySQL under the Shared strategy, the tail latency is 1.18ms, whereas under the Even strategy, the tail latency is 1.81ms.

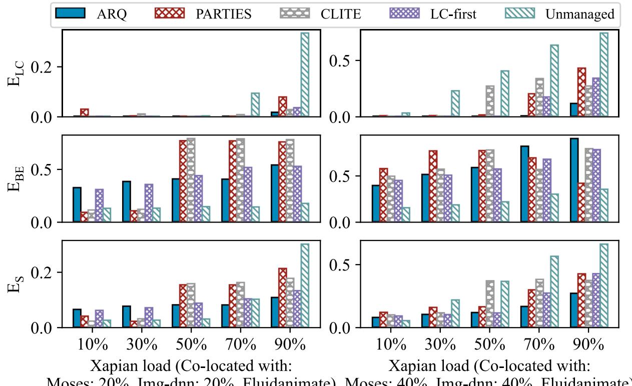
Fig. 15(a) illustrates the proportion of combinations where application performance has improved using the Even and RR strategies relative to the Shared strategy. The results show that compared to the Shared strategy, the Even strategy improves the performance of 50.9% of applications, but degrades the performance of 49.1% of applications. On the other hand, the RR strategy improves the performance of 94.2% of high-priority applications and only degrades the performance of 5.8% of high-priority applications.

Observation 3: PA does not reduce the resource utilization of a datacenter, but IA does.

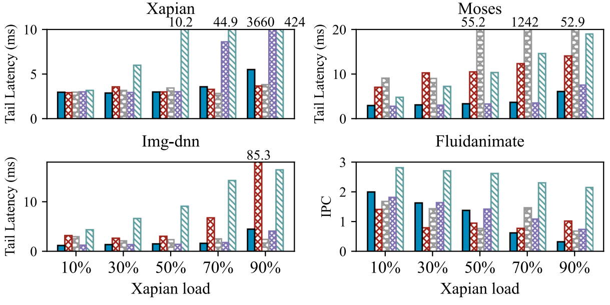
Fig. 15(b) illustrates the box plot of CPU utilization with different scheduling strategies. It is clear that the CPU utilization in the Even strategy is significantly lower than that of the Shared or RR strategy. In fact, the resource utilization of 96.4% of the combinations with the Even strategy is lower than that with the Shared strategy.

Besides CPU resource, the shared cache can also benefit from using IA and PA strategies. IA can be used to prevent applications from evicting cache lines from each other, which can improve data reuse in the cache. On the other hand, PA can be used to give high-priority applications priority access to cache lines, making

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.



(a) $\mathrm{E_{LC}}$, $\mathrm{E_{BE}}$ and $\mathrm{E_S}$ when the load of Moses and Img-dnn is 20% and 40%.



(b) Tail latency and IPC when the load of Moses and Img-dnn is 40%.

Fig. 16. Results when Xapian, Moses, Img-dnn and Fluidanimate are colocated.

them more difficult to evict. This can result in a reduction of their average memory access time and tail latency.

IX. Evaluation of the ARQ Strategy

In this section, we evaluate the ARQ strategy with LC, BE and system entropy in the situation of constant load and fluctuating load, respectively. The experimental set up is the same as that described in Section VII-A.

A. The Case of Constant Load

In this experiment, we colocate the BE application Fluidanimate with three LC applications (i.e., Xapian, Moses, and Img-dnn), while keeping the load of the LC applications constant. Fig. 16 displays the E_{LC}, E_{BE}, and E_S for various strategies, with a load of 20% (left) and 40% (right) of the maximum load for Moses and Img-dnn, while Xapian's load ranges from 10% to 90%.

In the case of low load on the LC applications, the Unmanaged strategy demonstrates the lowest E_S among all the strategies, indicating the advantages of resource sharing. This is because interference between applications is minimal, and resource sharing can achieve higher resource utilization than other strategies. However, under high load conditions, despite the low E_{BE}, the rapid increase in E_{LC} leads to an increase in E_S. This is because the Unmanaged strategy does not take any measures to ensure the QoS of the LC applications.

The LC-first strategy allows the LC applications to preempt the resources of the BE applications if needed, which is an improvement over the Unmanaged strategy. However, while the LC-first strategy has a much lower E_{LC} than the Unmanaged strategy, it incurs a substantial increase in E_{BE}.

Both PARTIES and CLITE utilize complete resource isolation to mitigate interference among applications and ensure the QoS of LC applications. When the load of the LC applications is low, the strategies allocate many resources to the BE application on the premise of guaranteeing the QoS of LC applications, resulting in low E_{BE} and E_S. When the load is high (e.g., the load of Moses and Img-dnn is 20% each and Xapian's load exceeds 50%), they allocate more resources to the LC applications and fewer resources to the BE application, leading to high E_{BE} and E_S.

As shown in Fig. 16(a), the ARQ strategy achieves the lowest E_S among all the strategies. It reduces E_{LC} more significantly than other strategies, implying that the QoS of the LC applications has been guaranteed preferentially. ARQ also has the lowest E_{BE} during most of the time among all the strategies based on resource isolation. When the load is extremely high, it is reasonable that ARQ has a higher E_{BE} than other strategies because ARQ lets the LC applications preferentially occupy the resources of the shared region. In this manner, the characteristic of all the applications has been well-utilized to improve the overall user experience.

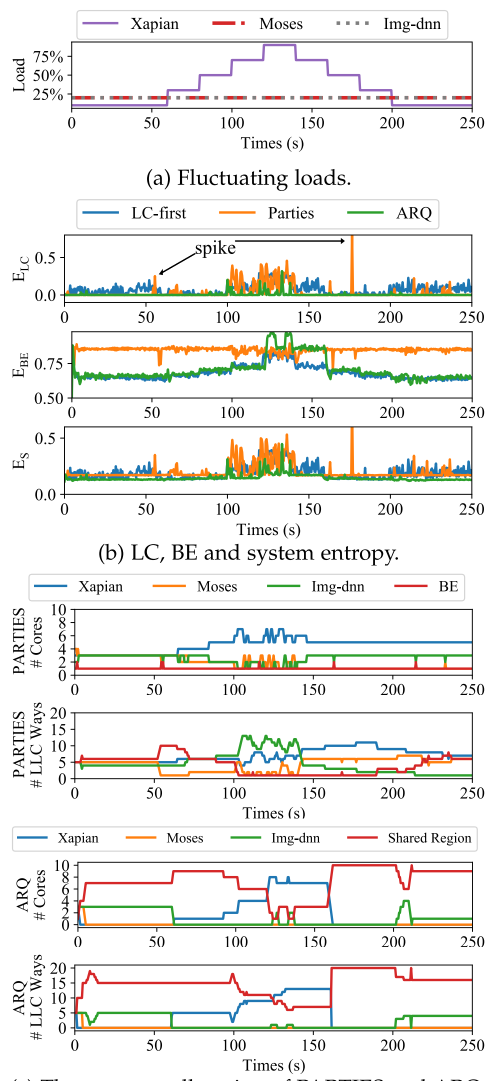
Fig. 16(b) provides a more detailed view of the tail latency and IPC for a specific scenario (i.e., when the load of Moses and Img-dnn is 40%). Using Unmanaged as the baseline, ARQ achieves the highest reduction in tail latency, by 66.5% on average, compared to 43.6% and 37.2% for CLITE and PARTIES, respectively. Meanwhile, ARQ improves the throughput of LC applications by 2.98, compared to 1.77 and 1.59 for CLITE and PARTIES, respectively. When the load is low (e.g., Xapian's load $\leqslant$ 50%), ARQ has improved the throughput of BE applications, achieving higher IPC compared to PARTIES and CLITE, by 63.8% and 37.1%, respectively. When the load is high (e.g., Xapian's load $\geqslant$ 70%), ARQ prioritizes tail latency over IPC and allocates resources to LC applications to ensure the QoS targets.

B. The Case of Fluctuating Load

In this section, we evaluate different strategies with a fluctuating load as many LC applications in a datacenter experience load fluctuations (e.g., high load in the daytime and low load at night) during execution. We still choose Xapian, Moses, and Img-dnn as the LC applications, and Stream as the BE application. The load of Moses and Img-dnn is set to 20%, and the load of Xapian varies from 10% to 90%. Fig. 17(a) depicts the fluctuations in Xapian's load, and Fig. 17(b) shows the changes of E_{LC}, E_{BE} and E_S for LC-first, PARTIES, and ARQ strategies. Additionally, Fig. 17(c) illustrates how ARQ and PARTIES dynamically schedule resources to adapt to load fluctuations.

As shown in Fig. 17, which covered a 250-second time frame with 500 data points, ARQ incurred 59 tail latency violations,

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.



(a) Fluctuating loads.

(b) LC, BE and system entropy.

(c) The resource allocation of PARTIES and ARQ.

Fig. 17. E_{LC}, E_{BE} and E_S and the corresponding scheduling process of LC-first, PARTIES, and ARQ (Xapian's load is fluctuating).

while PARTIES had 105. The majority of these violations were caused by the resource adjustment process that occurred following the load fluctuations.

In the initial stage, when the load of all three LC applications is low, both PARTIES and ARQ are able to meet the QoS target of all LC applications. PARTIES assigns only 1 processing unit and 6 LLC ways to the BE application, while ARQ assigns 7 processing units and 15 LLC ways to the shared region. As a result, compared to PARTIES, ARQ significantly improves the user experience of BE applications by reducing E_{BE} by 22.3% (from 0.85 to 0.66).

During the period from 100s to 120s, Xapian's load increases to 70%. PARTIES fails to allocate resources that satisfy the QoS target, leading to high E_{LC}. In contrast, ARQ succeeds in exploring the allocation space and finding a solution that satisfies the QoS target. Although this causes a slight increase in E_{BE}, it is reasonable because it results in a significant improvement in the overall user experience as measured by E_S. During the period from 120s to 140s, Xapian's load is increased to 90%. Although neither PARTIES nor ARQ can completely eliminate QoS violations, ARQ has much lower values of E_{LC} and E_S than PARTIES. Compared to PARTIES, ARQ has improved the throughput of LC and BE applications by 36.4% on average.

In Fig. 17, there are spikes observed in the E_{LC} curve of PARTIES, which occur because PARTIES tentatively reduces the resources of an LC application to allocate more resources to the BE application. If the LC application fails to meet the QoS target after downsizing, it immediately recovers from the previous downsizing action. ARQ, on the other hand, effectively mitigates these spikes, despite having a more aggressive downsizing action than PARTIES.

ARQ achieves a smoother resource allocation by utilizing the shared region. As the load of the LC applications increases and the available resources become insufficient, ARQ quickly preempts the resources in the shared region from the BE applications to avoid a rapid rise in tail latency. This preemptive action may harm the throughput of the BE applications, but it ensures the QoS of the LC applications, making it a worthwhile tradeoff. On the other hand, PARTIES gradually allocates more resources to the LC applications to satisfy the QoS target, leading to the spiking phenomenon observed in Fig. 17.

X. Related Work

A. Decision Theorem and Interference Quantification

The DIP theorem complements the virtualization ability decision theorem [45] and the CAP theorem [22].

Scott et al. [58] characterize interference by considering the service rate under interference and the duration of time that interference persists. Other prior work, such as [9], [25], [38], [46], [51], [68], quantified interference by measuring the values of IPC or execution time before and after applications are interfered with. However, for LC applications, users concern about tail latency rather than IPC, and the change of IPC may be caused by interference from other applications or by fluctuations in their load. Therefore, quantifying interference using IPC alone is not appropriate.

Many researchers [15], [16], [37], [64], [70] have used tail latency before and after interference to quantify interference in LC applications. However, different applications have varying ideal tail latencies ranging from microseconds to seconds, so using tail latency to quantify interference for different applications is not practical. To address this issue, we propose E_S as a measure of interference that unifies the impact on different LC and BE applications. E_S is formal, reasonable, and systematical and has interpretability and measurability, making it a more appropriate metric than ad hoc metrics such as those proposed in [52], [57], [58]. In cases where applications care about both latency and IPC, we could choose a more critical performance metric or develop an aggregated metric that considers various metrics. This is a challenging scenario even without colocation and will be left as future work.

B. Resource Scheduling

Resource scheduling is a crucial issue in datacenter management, as it involves determining how to allocate resources

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

in a way that satisfies the QoS targets of different types of applications. In previous studies, software and hardware level resource isolation techniques were used to manage resources and eliminate interference on specific resources. Feedback-based resource managers have also been proposed, which detect and respond to QoS violations using application state information like tail latency and input load.

Heracles [34] uses a threshold-based method to safely colocate LC and BE applications and manage interference. PARTIES [13] dynamically adjusts the resource allocation of each application by monitoring tail latency to improve resource utilization. CLITE [43] uses Bayesian optimization to explore resource sensitivity to find an allocation with optimal performance. Sturgeon [41] uses decision trees and binary search to satisfy power consumption constraints and QoS targets. Twig [40] uses multi-agent deep reinforcement learning to improve energy efficiency when running multiple LC applications. Although CLITE, Sturgeon, and Twig can all coordinate scheduling of multiple resources in one step, they have limitations. Specifically, Sturgeon relies on prior application knowledge and offline pre-training, while CLITE and Twig involve a large amount of computations at runtime, potentially worsening application performance.

CuttleSys [30] evaluates the effect of the current allocation and adapts to changes in the behaviour of applications by collaborative filtering and dynamically search. Sinan [69] predicts end-to-end latency and QoS violation probability using historical system information. Stretch [36] statically partitions ROB and LSQ capacity of colocated applications.

Although these methods perform complete resource isolation for all applications, they have not explored the opportunities of sharing resources at the right time to maximize resource utilization and system throughput. Dunn [48] also uses CAT to partition the cache, but Dunn cares more about system fairness while ARQ focuses on both fairness between LC and BE applications and overall system performance.

XI. Conclusion

As workloads are diverse and rapidly changing, it is challenging to achieve the optimal match between applications and the underlying architecture in a datacenter. However, it is essential to simultaneously achieve high application concurrency and high QoS to ensure maximum resource utilization and user satisfaction. This study presents Ah-Q, which includes a pair of theorems (DIP and TLT), the metric (E_S) and the strategy (ARQ) to effectively address this challenge.

DIP theorem provides the necessary and sufficient conditions to determine whether a datacenter can guarantee QoS and achieve high-throughput. While previous studies have recognized the issue of severe interference in modern datacenters, their methods are heuristic. TLT theorem formulates the relationship between throughput and tail latency. These theorems serve as reminders to datacenter designers that high concurrency does not always lead to high throughput. In addition to the total amount of available resources, the ability to manage those resources is crucial for achieving high throughput in a datacenter.

E_S is a comprehensive and analytical approach to quantifying interference caused by resource contention in a datacenter. ARQ leverages both resource isolation and sharing to optimize performance. We validate the correctness and effectiveness of E_S and ARQ on the platform of a real datacenter, demonstrating significant improvements in overall user experience. Our experiments show that E_S and ARQ are robust and easy-to-use across diverse scenarios.

Acknowldgment

The authors would like to thank the anonymous reviewers for their insightful comments. The authors would also like to thank Professor Zhiwei Xu and Ninghui Sun for their valuable suggestions. The first author thanks Professor Mingfa Zhu for his guidance.

References

[1] memtier benchmark website, 2024. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[2] Nginx official website, 2024. [Online]. Available: http://nginx.org

[3] Redis official website, 2024. [Online]. Available: https://redis.io

[4] Sysbench: A system performance benchmark, 2024. [Online]. Available: http://sysbench.sourceforge.net

[5] wrk2: A constant throughput, correct latency recording variant of wrk, 2024. [Online]. Available: https://github.com/giltene/wrk2

[6] Intel 64 and IA-32 architectures software developer's manual, *Volume 3B: Syst. Program. Guide, Part*, vol. 2, no. 11, pp. 1–64, 2011.

[7] R. Baeza-Yates, "Applications of web query mining," in *Proc. Eur. Conf. Inf. Retrieval*, Springer, 2005, pp. 7–22.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[9] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. IEEE/ACM Annu. Int. Symp. Microarchit.*, 2008, pp. 318–329.

[10] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2000, Art. no. 7.

[11] X. Bu, J. Rao, and C.-Z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proc. Int. Symp. High Perf. Parallel Distrib. Comput.*, 2013, pp. 227–238.

[12] Q. Chen et al., "Alita: Comprehensive performance isolation through bias resource management for public clouds," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–13.

[13] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 107–120.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[15] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[16] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[17] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 599–613, 2017.

[18] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[19] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 104–117.

[20] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge, U.K.: Cambridge Univ. Press, 2015.

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

[21] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 281–297.

[22] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[23] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. ACM Int. Symp. Qual. Service*, 2019, pp. 1–10.

[24] Intel, *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Santa Clara, CA, USA: Intel Corporation, Apr. 2015.

[25] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–12.

[26] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 598–610.

[27] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 729–742, 2014.

[28] H. Kasture and D. Sanchez, "TailBench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–10.

[29] P. Koehn et al., "Moses: Open source toolkit for statistical machine translation," in *Proc. 45th Annu. Meeting ACL Interactive Poster Demonstration Sessions*, 2007, pp. 177–180.

[30] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores," in *IEEE/ACM Annu. Int. Symp. Microarchit.*, 2020, pp. 650–664.

[31] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations Res.*, vol. 9, no. 3, pp. 383–387, 1961.

[32] Y. Liu, X. Deng, J. Zhou, M. Chen, and Y. Bao, "Ah-Q: Quantifying and handling the interference within a datacenter from a system perspective," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 471–484.

[33] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proc. IEEE/ACM Int. Symp. Comput. Archit.*, 2014, pp. 301–312.

[34] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 450–462.

[35] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[36] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing QoS and throughput for colocated server workloads on SMT cores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 15–27.

[37] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. IEEE/ACM Annu. Int. Symp. Microarchit.*, 2011, pp. 248–259.

[38] J. Mars, L. Tang, and M. L. Soffa, "Directly characterizing cross core interference through contention synthesis," in *Proc. 6th Int. Conf. High Perform. Embedded Archit. Compilers*, 2011, pp. 167–176.

[39] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," in *Proc. IEEE Comput. Soc. Tech. Committee Comput. Archit. Newslett.*, vol. 2, no. 19–25, 1995.

[40] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 167–179.

[41] P. Pang et al., "Sturgeon: Preference-aware co-location for improving utilization of power constrained computers," in *Proc. IEEE Int. Parallel Distribut. Process. Symp.*, 2020, pp. 718–727.

[42] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proc. EuroSys Conf.*, 2019, pp. 1–16.

[43] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 193–206.

[44] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, "Phase-aware cache partitioning to target both turnaround time and system performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2556–2568, Nov. 2020.

[45] G. J. Popek, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 4, 1974, Art. no. 121.

[46] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki, "Fact: A framework for adaptive contention-aware thread migrations," in *Proc. 8th ACM Int. Conf. Comput. Front.*, 2011, pp. 1–10.

[47] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 423–432.

[48] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with Intel's cache allocation technology," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn.*, 2017, pp. 194–205.

[49] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, 1948.

[50] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," in *Proc. Int. Conf. Architectural Support Program. Lang. operating Syst.*, 2000, pp. 234–244.

[51] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. IEEE/ACM 48th Annu. Int. Symp. Microarchit.*, 2015, pp. 62–75.

[52] N.-H. Sun, Y.-G. Bao, and D.-R. Fan, "The rise of high-throughput computing," *Front. Inf. Technol. Electron. Eng.*, vol. 19, no. 10, pp. 1245–1250, 2018.

[53] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proc. ACM/IEEE Conf. Supercomput.*, Washington, DC, USA, 1990, pp. 324–333.

[54] J. Tiedemann, "Parallel data, tools and interfaces in OPUS," in *Proc. 8th Int. Conf. Lang. Resour. Eval.*, Istanbul, Turkey, ELRA, 2012, pp. 2214–2218.

[55] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. ACM Symp. Operat. Syst. Princ.*. ACM, 2013, pp. 18–32.

[56] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. Math*, vol. 58, no. 345–363, 1936, Art. no. 5.

[57] S. Votke, J. A. Jaleel, A. Suresh, M. Delasay, S. Doroudi, and A. Gandhi, "Optimal Markovian dynamic control of interference-prone server farms," in *Proc. IEEE 27th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2019, pp. 295–308.

[58] S. Votke, S. A. Javadi, and A. Gandhi, "Modeling and analysis of performance under interference in the cloud," in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2017, pp. 232–243.

[59] M. M. Waldrop, "The chips are down for Moore's law," *Nat. News*, vol. 530, no. 7589, 2016, Art. no. 144.

[60] W. Walker et al., "Sphinx-4: A flexible open source framework for speech recognition," *Sun Microsystems*, 2004.

[61] C. Wang, B. Urgaonkar, G. Kesidis, A. Gupta, L. Y. Chen, and R. Birke, "Effective capacity modulation as an explicit control knob for public cloud profitability," *ACM Trans. Auton. Adaptive Syst.*, vol. 13, no. 1, pp. 1–25, 2018.

[62] M. Widenius, D. Axmark, and K. Arno, *MySQL Reference Manual: Documentation From the Source*. Sebastopol, CA, USA: O'Reilly Media, 2002.

[63] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, "Small is better: Avoiding latency traps in virtualized data centers," in *Proc. ACM Symp. Cloud Comput.*, 2013, pp. 1–16.

[64] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 607–618, 2013.

[65] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, Art. no. 10.

[66] S. M. Zahedi and B. Lee, "Ref: Resource elasticity fairness with sharing incentives for multiprocessors," *ACM Sigplan Notices*, vol. 49, no. 1, pp. 145–160, 2014.

[67] S. Zhang, H. Shan, Q. Wang, J. Liu, Q. Yan, and J. Wei, "Tail amplification in n-tier systems: A study of transient cross-resource contention attacks," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1527–1538.

[68] X. Zhang et al., "CPI2: CPU performance isolation for shared compute clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 379–391.

[69] Y. Zhang et al., "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 167–181.

[70] L. Zhao et al., "Rhythm: Component-distinguishable workload deployment in datacenters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

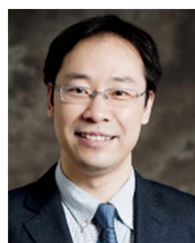
Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.

Yuhang Liu (Member, IEEE) received the PhD degree in computer science from Beihang University, Beijing, in 2013. He is an associate professor with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS). He has been a postdoctoral researcher with the Computer Science Department of Illinois Institute of Technology (IIT), Chicago. He is a member of ACM. His research interests include computer architecture and high performance computing.

Xin Deng (Student Member, IEEE) received the ME degree from ICT, CAS, Beijing, in 2022. He currently works with Tencent as a research and development engineer. His primary research interests include computer architecture and resource scheduling in datacenters.

Jiapeng Zhou (Student Member, IEEE) received the BS degree from the School of Software Engineering, Huazhong University of Science and Technology, Wuhan, China, in 2022. He is currently working toward the ME degree with ICT, CAS, Beijing. His primary research interests include computer architecture and memory system optimization.

Mingyu Chen (Member, IEEE) received the PhD degree in computer system architecture from ICT, CAS, Beijing, in 2000. He is currently a professor with ICT, CAS. He is a member of ACM. His main research interests include computer architecture, operating system, and algorithm optimization for high performance computers.



Yungang Bao (Member, IEEE) received the PhD degree in computer science from ICT, CAS, Beijing, in 2008. He is a professor with ICT, CAS, Beijing. From 2010 to 2012, he was a postdoctoral researcher with the Department of Computer Science, Princeton University. His current research interests include computer architecture, operating system, and system performance modeling and evaluation. He is a member of ACM.

Authorized licensed use limited to: ShanghaiTech University. Downloaded on March 28,2024 at 15:04:44 UTC from IEEE Xplore. Restrictions apply.