2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071128

Ah-Q: Quantifying and Handling the Interference within a Datacenter from a System Perspective

Yuhang Liu[*†], Xin Deng[*†], Jiapeng Zhou[*†], Mingyu Chen[*†‡], Yungang Bao[*†]
[*]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[†]University of Chinese Academy of Sciences, Beijing, China
[‡]Zhongguancun Laboratory, Beijing, China
{liuyuhang, dengxin19g, zhoujiapeng22s, cmy, baoyg}@ict.ac.cn

***Abstract*—Interference among applications frequently occurs in a datacenter and significantly influences the cost-efficiency and the user experience. However, it is challenging for us to quantify the exact intensity of the interference that occurred in the overall system of a datacenter, because there are many concurrent applications in a datacenter, and their type can be either latency-critical (LC) and best-effort (BE). To address this issue, we present the Ah-Q which includes a theory and a strategy.**

First, we propose the "system entropy" ($\mathrm{E_S}$) theory to holistically and analytically quantify the interference in a datacenter to address this vital issue. The interference is caused by the scarcity of resources or/and the irrationality of scheduling. As more appropriate scheduling can compensate for resource scarcity, we derive the concept of "resource equivalence" to quantify the effectiveness of a resource scheduling strategy. We evaluate different resource scheduling strategies to validate the correctness and effectiveness of the proposed theory.

Moreover, using the theory to eliminate interference, we propose a new resource scheduling strategy; i.e., ARQ, which dynamically allocates the isolated resources and the shared resources to simultaneously harvest the benefits of isolation and sharing. Our results show that compared to the state-of-the-art strategies (PARTIES and CLITE), ARQ is more effective to reduce the tail latency of the LC applications and to increase the IPC of the BE applications. Compared with PARTIES and CLITE, ARQ increases the yield (the ratio of satisfactory LC applications) by 25% and 20%, respectively; when the load is low, ARQ increases IPC of BE applications by 63.8% and 37.1%, respectively; ARQ reduces $\mathrm{E_S}$ by 36.4% and 33.3%, respectively. The effectiveness of ARQ has saved resources significantly to achieve the same satisfactory overall user experience.

I. Introduction

In a typical datacenter, there are two types of applications. The first type is latency-critical (LC) applications, such as Redis [1] and Moses [24]. User experience of these LC applications is affected by tail latency and user expectation. The second type is best-effort (BE) applications, e.g., Spark [54] and Fluidanimate [3]. Performance of these BE applications is usually quantified in terms of instructions-per-cycle (IPC).

To improve resource efficiency in a datacenter, multiple applications are typically collocated on the same node. However, interference and contention in shared hardware resources negatively affect applications' performance [5, 7, 10, 20, 30, 41, 51, 52, 53]. For LC applications, interference can make an exceedingly destructive impact because the user experience is pretty sensitive to the tail latency. For BE applications, though not fatal effects, the drop of IPC due to interference is still desired to be as small as possible. The user experience of BE applications should not be overlooked. We use relative importance (RI) to term the importance difference between LC and BE applications.

As many different applications are concurrently running in a datacenter, we have an array of tail latency or instructions per cycle (IPC) values, which makes it challenging for us to tell the exact intensity of the interference that occurred in the overall system of a datacenter (a detailed example will be presented in Section II-C). The reason is that the tail latency and IPC are from an individual application perspective rather than the system perspective. Hence, how to quantify and reduce the interference collectively in a datacenter is a vital issue that needs to be addressed.

Prior work has used various methods to quantify interference, including the ratio of tail latency over instruction throughput [44], reduced service rate of a virtual machine (VM), and the duration of interference [47, 48]. These methods are effective and make sense in special cases. However, they are mainly ad hoc, and their units are not well defined, making it difficult to apply in different scenarios.

In this study, we propose system entropy ($\mathrm{E_S}$) to quantify the interference in a datacenter, following a three-step paradigm inspired by information entropy. Recall that Shannon has quantified uncertainty using information entropy in three steps [40]. Shannon first gave the required properties of information entropy, then proposed an analytical expression, and proved that the expression satisfies the required properties. $\mathrm{E_S}$ is fundamentally different from information entropy, but will also be proposed in a similar three-step manner. That is, we first itemize the required properties of $\mathrm{E_S}$, and then propose the analytical expression of $\mathrm{E_S}$, and finally validate that the expression has the required properties. Based on the analysis of the interference phenomenon of datacenters, we analyze the reasons for high tail latency, and then we distinguish and quantify three different types of interference.

$\mathrm{E_S}$ systematically quantifies the degree of interference in a datacenter. Specifically, $\mathrm{E_S}$ can be decomposed into LC entropy ($\mathrm{E_{LC}}$) and BE entropy ($\mathrm{E_{BE}}$). $\mathrm{E_{LC}}$ quantifies the interference that LC applications have received out of their tolerance. $\mathrm{E_{BE}}$ quantifies the interference that BE applications have suffered. If a datacenter only runs LC applications, $\mathrm{E_S}$ is just $\mathrm{E_{LC}}$. Similarly, if there are only BE applications on the node, $\mathrm{E_S}$ is simply $\mathrm{E_{BE}}$. If LC and BE applications co-exist, $\mathrm{E_S}$ is the linear combination of $\mathrm{E_{LC}}$ and $\mathrm{E_{BE}}$. $\mathrm{E_S}$ accommodates the performance degradation of all the collocated applications in the same datacenter, regardless of their characteristics and their possibly different performance metrics. In this way, we can use a single value to quantify interference of the overall system, and the metric is robust to various collocation scenarios.

We conducted a series of experiments and show how $\mathrm{E_S}$ changes with varying resource allocations and scheduling strategies. We prove that $\mathrm{E_{LC}}$ is correct and effective for guiding and evaluating different resource scheduling strategies. For instance, when $\mathrm{E_{LC}} = 0$, the tail latency requirements of all the LC applications have been satisfied; i.e., the yield (the ratio of satisfactory LC applications) is 100%. When $\mathrm{E_{LC}} > 0$, the absolute value of $\mathrm{E_{LC}}$ reflects how much of the overall user experience of the LC applications has not been satisfied.

Many state-of-the-art resource managers in prior studies [8, 12, 21, 22, 26, 27, 33, 34, 35, 36] used software and hardware resource isolation techniques to strictly isolated collocated applications, eliminating resource interference. Some researchers [7, 14, 37] shown that resource isolation may reduce resource utilization. However, their methods only focus on cache partitioning, and are only designed for BE applications. In this study, we find that strict isolation usually reduces resource utilization when interference among applications is not severe, and allowing resources to be flexibly isolated or shared among applications gives huge potential to mitigate the interference of LC and BE applications.

In this paper, we make the following main contributions:

❶ We propose the Ah-Q toolkit which includes a theory to quantify and a strategy to handle the interference within a datacenter.

❷ We propose the required properties and the analytical expression of system entropy, $\mathrm{E_S}$. $\mathrm{E_S}$ is a dimensionless single "figure of merit" of a datacenter, very useful for interference quantification and evaluation. Based on $\mathrm{E_S}$, we propose the concept of "resource equivalence" to evaluate the effectiveness of different scheduling strategies.

❸ Using the detected entropy as the feedback signal, to reduce interference, we design an associative scheduling strategy, ARQ, which allows partial resource sharing among BE and LC applications, and dynamically adjusts the size of isolated and shared resources. A space-time resource utilization model has been built to reveal the cause of interference and interpret the advantages of ARQ over previous strategies.

❹ We compare ARQ with the state-of-the-art strategies, CLITE [36] and PARTIES [8]. Our evaluation results show that, compared with PARTIES and CLITE, ARQ reduces $\mathrm{E_S}$ by 36.4% and 33.3% on average, respectively.

II. The System Entropy ($\mathrm{E_S}$)

Below, we propose the system entropy ($\mathrm{E_S}$) to quantify the interference occurring in a datacenter, following Shannon's information entropy paradigm [40]. First, we present the required properties of the measure (in sub-section II-A, then propose an analytical definition (in sub-sections II-B to II-B), and finally, validate their consistency (in section III).

TABLE I
List of symbol abbreviations.

Symbol	Description
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
$\mathrm{E_{LC}}$	LC entropy
$\mathrm{E_{BE}}$	BE entropy
$\mathrm{E_S}$	System entropy

A. The Required Properties of E_S

Considering the crucial influences of resource amount and resource scheduling strategy on interference, we require $\mathrm{E_S}$ to satisfy the following three properties.

① Dimensionless: $\mathrm{E_S}$ should have no dimension (e.g., its unit should not be a time or resource unit), and its value should be between 0 and 1. The closer the value is to 1, the greater the interference is.

② Resource amount sensitiveness: Given a set of co-running applications and a resource scheduling strategy, when the number of available resources in a datacenter increases, $\mathrm{E_S}$ should decrease or at least not increase.

③ Scheduling strategy sensitiveness: Given a set of co-running applications and a fixed number of available resources, when the scheduling strategy has reduced the resource contention among applications, $\mathrm{E_S}$ should decrease.

In the rest of this section, we introduce the analytical expression of $\mathrm{E_S}$ in three different scenarios in a datacenter. Table I lists the symbol abbreviations used in this paper.

B. The Analytical Expression of $\mathrm{E_S}$

The first scenario is that only N different LC applications are running, but no BE application exists in a datacenter. In this scenario, $\mathrm{E_S}$ is $\mathrm{E_{LC}}$, which is defined as follows.

There are three basic attributes for each LC application in a datacenter. For application i $(i = 1, 2, ..., N)$, TL_{i0} denotes application i's ideal tail latency (i.e., the tail latency when application i has not suffered any interference), TL_{i1} is the tail latency of application i when application i is under collocation, potentially suffering interference, and M_i is the maximum tail latency that application i can tolerate. Note that the ideal latency TL_{i0} can be obtained through resource isolation technology to temporarily allocate sufficient resources to application i. We can quantify the interference tolerance of application i as Eq. (1).

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

According to our observations, a user of an LC application determines M_i following two principles: (1) The more critical the application is, the smaller the tail latency threshold will be. (2) Users usually choose a value from the flat to small-slope region as M_i. Therefore, the user-defined target is a threshold affected by many factors and is only of a reference significance [42], and thus has some elasticity. In this study, we assume that the relative elasticity of M_i is 5%.

Since $TL_{i0} < M_i$, the range of A_i is $[0,1]$. The smaller M_i is, the closer A_i is to 0, and the smaller the application's interference tolerance is, and vice versa. We use R_i in Eq. (2) to quantify the interference that application i suffers.

$$R_i = 1 - \frac{TL_{i0}}{TL_{i1}} \tag{2}$$

Since $TL_{i0} < TL_{i1}$, the range of R_i is (0, 1). The smaller TL_{i1} is, the closer R_i is to 0, indicating that the interference suffered by the application is small, and vice versa. We use ReT_i in Eq. (3) to represent the remaining tolerance of application i after being interfered.

$$ReT_i = \left(A_i > R_i \ ? \ \ 1 - \frac{TL_{i1}}{M_i} : 0 \right) \tag{3}$$

We use Q_i in Eq. (4) to represent the interference that application i cannot tolerate. When the interference application i suffers (i.e., R_i) is larger than the interference tolerance (i.e., A_i), $Q_i = 1 - \frac{M_i}{TL_{i1}}$. Otherwise, $Q_i = 0$.

$$Q_i = \left(R_i > A_i \ ? \ \ 1 - \frac{M_i}{TL_{i1}} : 0 \right) \tag{4}$$

The A_i, R_i and Q_i above inspired us to develop a resource scheduling strategy (referred to as ARQ) which will be presented in Section IV. Moreover, we define $\mathrm{E_{LC}}$ as the interference that the LC applications cannot tolerate, which can be expressed as Eq. (5).

$$E_{LC} = \frac{1}{N} \sum_{i=1}^{N} Q_i \tag{5}$$

The second scenario is that only M different BE applications are running, but no LC application exists in a datacenter. In this scenario, $\mathrm{E_S}$ is the BE entropy ($\mathrm{E_{BE}}$). As shown in Eq. (6), we define $\mathrm{E_{BE}}$ as the slowdown incurred by the interference that the BE applications have suffered, 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.

$$E_{BE} = 1 - \frac{M}{\sum_{i=1}^{M} \frac{IPC_{solo}(i)}{IPC_{real}(i)}} \tag{6}$$

When none of the BE applications suffers any interference, $\mathrm{E_{BE}}$ is 0. The higher the interference occurring for application i is, the larger the ratio of $IPC_{solo}(i)$ over $IPC_{real}(i)$ is, and the closer the value of $\mathrm{E_{BE}}$ is to 1.

The third scenario is that LC and BE applications co-exist in a datacenter. In this scenario, as shown in Eq. (7), $\mathrm{E_S}$ is the linear combination of $\mathrm{E_{LC}}$ and $\mathrm{E_{BE}}$, where the relative importance (RI) is involved.

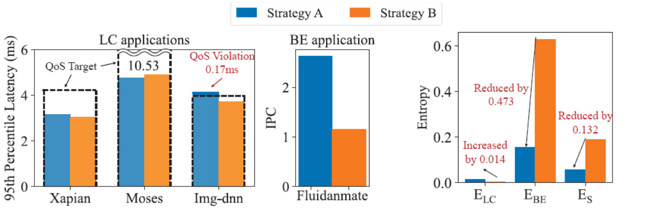


Fig. 1. Tail latency of the LC applications, IPC of the BE application and the entropy values under resource scheduling strategies A and B. The dotted box represents the QoS target of the LC applications.

$$E_S = RI \times E_{LC} + (1 - RI) \times E_{BE} \tag{7}$$

The rationale behind Eq. (7) is to eliminate $\mathrm{E_{LC}}$ and $\mathrm{E_{BE}}$ simultaneously to achieve the minimum $\mathrm{E_S}$. Generally, the range of RI is [0, 1]. However, when the resources are insufficient, reducing $\mathrm{E_{LC}}$ should take precedence over reducing $\mathrm{E_{BE}}$, which changes the range of RI into [0.5, 1].

Interestingly, Scenario 1 and 2 are the extreme cases of Scenario 3. Specifically, when only BE applications exist in a datacenter, only $\mathrm{E_{BE}}$ needs to be considered in system entropy, and therefore RI is chosen to be 0. This is typical in conventional high-performance computing. When only LC applications are running in the system, RI is assigned to be 1. The larger the value of RI, the higher the priority of the LC applications over that of the BE applications. Datacenter managers can determine the value of RI by considering several factors (e.g., the criticality of LC applications, the fairness among all the applications, and the economic benefit of the datacenter). In this study, without losing representativeness, we set RI to 0.8.

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. If necessary, the $\mathrm{E_S}$ model can be extended to involve different RI factors among the same type of applications.

C. Advantages of E_S

This section will show the advantages of the proposed $\mathrm{E_S}$ over tail latency and IPC with a simple example. Figure 1 shows 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 IPC and tail latency values shown in Figure 1, it is not straightforward for us to distinguish which strategy is better, yet we can precisely and reasonably do this with $\mathrm{E_S}$. The reason lies in the following advantages of $\mathrm{E_S}$.

First, $\mathrm{E_S}$ is concise and easy-to-use in practice. If IPC and tail latency are used, there will be many individual performance data that must be considered simultaneously. Assume N different LC applications and M different BE applications coexist in a datacenter. For each strategy, we need to examine $2N+M$ different performance values (i.e., the tail latency and the target threshold of each LC application, and the IPC of each BE application). In the example of Figure 1, even though there are only three LC applications and one BE application (the number of collocated applications is much larger in the real cloud [18]), for each strategy, we still need to examine 7 values at the same time (3 tail latency values, 3 tail latency thresholds and 1 IPC value), which is a challenging task.

Second, E_S reflects the overall user experience of many collocated applications more comprehensively. The change in the resource scheduling strategy may improve the performance of some applications and degrade the performance of others. In this example, for strategy B, although the tail latency of LC application Img-dnn is improved, the BE application's IPC has dramatically deteriorated. Therefore, with IPC and tail latency, it is difficult to determine whether the overall user experience of the datacenter is improved or not. QoS guarantee does not necessitate reducing E_{LC} to zero; that is, a small E_{LC} is tolerable. E_S reflects this observation in its definition. It is noteworthy that strategy A is not inferior to strategy B because the QoS violation in strategy A is tolerable. The LC application (Img-dnn) QoS violation 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%), so it is more reasonable to prefer strategy A over strategy B.

Third, E_S can be used to define resource equivalence. Given the budget and power constraints, it becomes increasingly difficult to increase the available resources of a datacenter [49]. Therefore, it is crucial to focus on improving the usage and increasing the utilization of resources rather than increasing available resources. When evaluating the optimization of a datacenter, we can express the effectiveness in the following form: *when achieving the same "overall user experience", how many resources can be saved by a new strategy compared to the baseline strategy*. We can use E_S to evaluate the improvement of a scheduling strategy over another one in terms of resource saving. We say a scheduling strategy p_1 is inferior to p_2 if p_1 has to use more resources to achieve the same E_S as p_2. Suppose the amount of resources used by p_2 is R, and p_1 uses ΔR more sources, this means that $E_S(p_1,\ \mathrm{R}+\Delta\mathrm{R}) = E_S(p_2,\ \mathrm{R})$. The improvement from p_1 to p_2 is equivalent to increasing ΔR amount of resources, and ΔR is referred to as the *resource equivalence* of strategy p_2 relative to p_1.

III. Verification of E_S

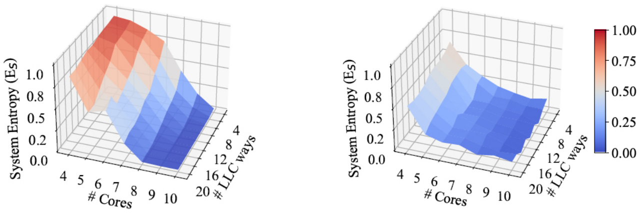
In this section, we verify that the analytical expression of E_S has satisfied all the required properties listed in Section II-A. It is easy to prove that E_S has the "dimensionless" property. We only need to focus on the other two properties.

A. Resource Amount Sensitiveness of E_S

To verify how E_S varies with the number of available resources, we run one BE application (Fluidanimate) and three LC applications (Xapian, Moses and Img-dnn with 20% of max load) concurrently in a datacenter.

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**



(a) Unmanaged

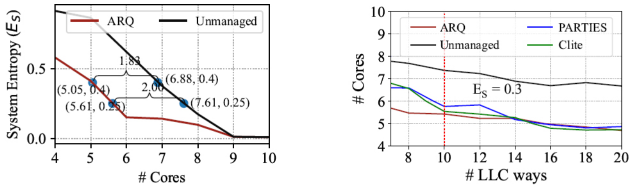
(b) ARQ

Fig. 2. Impact of the size of available resources on E_S (Xapian (20%), Moses (20%), Img-dnn (20%), Fluidmanate).

Table II shows E_{LC}, E_{BE} and E_S of Unmanaged when these applications run on 6-8 cores and all LLC ways. The max tail latency that an application can tolerate (i.e., M_i) and the ideal tail latency TL_{i0} are constant values. They are measured with enough resources, so the interference tolerance A_i does not depend on the number of available resources. When 6 processor cores are available, the real tail latency TL_{i1} of the three applications is higher than the M_i, so ReT_i is equal to 0. When more processor cores are available, ReT_i increases to 0.23, indicating that the remaining tolerance of the system is high at present, and there exist redundant resources that can be used to handle more requests.

When resources are scarce (the number of processor cores is 7), E_{LC} is large (i.e., 0.23). At this time, reducing the number of available processing units to 6, will make the tail latency deviate significantly from M_i and thus E_{LC} is increased to 0.64. However, if the number of processor cores increases to 8, the interference among applications will be reduced to an application-tolerable level ($\forall i, R_i < A_i$), and E_{LC} becomes 0 at this time.

Figure 2 shows E_S of Unmanaged and ARQ when the number of available processing units ranges from 4 to 10, and the number of LLC ways per set ranges from 4 to 20. When the number of available resources decreases for both strategies, E_S shows an increasing trend, verifying the second property of E_S. When the number of resources is sufficient (e.g., 10 processing units, 20 LLC ways), even if with the Unmanaged strategy, the interference among applications is small, with E_S only 0.006. When the number of resources is



(a) E_S varies with the number of processing units

(b) Isentropic lines of E_S

Fig. 3. An illustration of the concept of "resource equivalence".

insufficient (e.g., 6 processing units, 20 LLC ways), resource contention is severe, which causes E_S as high as 0.53. For the ARQ strategy, when the resources are sufficient (e.g., 10 processing units, 20 LLC ways), E_S is 0.008. However, when the number of resources is insufficient (6 processing units, 20 LLC ways), E_S of the ARQ strategy is 0.15.

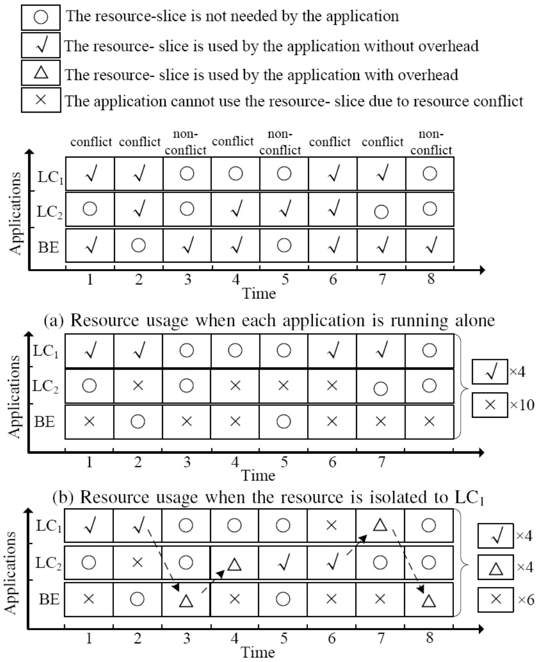
B. Scheduling Strategy Sensitiveness of E_S

Resource equivalence describes the difference in the number of resources between two different scheduling strategies when they reach the same E_S. The concept of "resource equivalence" is illustrated in different forms as shown in Figure 3(a) and (b). Figure 3(a) shows E_S of two strategies: Unmanaged and ARQ (the experimental setup will be described in Section V). The x-axis is the total number of available processing units, and the y-axis is the corresponding E_S. To make E_S reach 0.25, the Unmanaged strategy requires 7.61 cores, while the proposed ARQ strategy only requires 5.61 cores. The two-core resource that ARQ saved is the resource equivalence of the ARQ strategy compared to the Unmanaged strategy. Similarly, when E_S is 0.4, the resource equivalence is 1.83 cores.

Figure 3(b) shows the isentropic lines of different scheduling strategies when 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 (i.e., 0.3). As shown in Figure 3(b), 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 (i.e., resource equivalence R is close to 0). However, when the number of available LLC ways $<$ 10, the total amount of available resource is scarce and resource conflict is severe, and ARQ is able to achieve the same E_S with much fewer processing cores. For example, when 8 LLC ways are available, compared to PARTIES and CLITE, ARQ has saved 1 processing core for the data center, that is, the resource equivalence is 1 processing core (i.e., 12.5% processing cores). Similarly, using ARQ instead of the Unmanaged strategy brings a resource equivalence of 2 processing cores to the data center when 8 LLC ways are available (i.e., 25% processing cores have been saved).

IV. The ARQ Scheduling Strategy

In this section, we propose a scheduling strategy called ARQ to combine the advantages of resource sharing and resource isolation to reduce system entropy. The name of the strategy is to denote that A_i, R_i and Q_i in section II-B are three vital factors of an LC application.



○	The resource-slice is not needed by the application
√	The resource- slice is used by the application without overhead
△	The resource- slice is used by the application with overhead
×	The application cannot use the resource- slice due to resource conflict

Applications	conflict	conflict	non-conflict	conflict	non-conflict	conflict	conflict	non-conflict
LC_1	√	√	○	○	○	√	√	○
LC_2	○	√	○	√	√	√	○	○
BE	√	○	√	√	○	√	√	√
Time	1	2	3	4	5	6	7	8

(a) Resource usage when each application is running alone

Applications								
LC_1	√	√	○	○	○	√	√	○
LC_2	○	×	○	×	×	×	○	○
BE	×	○	×	×	○	×	×	×
Time	1	2	3	4	5	6	7	

√ ×4
× ×10

(b) Resource usage when the resource is isolated to LC_1

Applications								
LC_1	√	√	○	○	○	×	△	○
LC_2	○	×	○	△	√	√	○	○
BE	×	○	△	×	○	×	×	△
Time	1	2	3	4	5	6	7	8

√ ×4
△ ×4
× ×6

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

Fig. 4. An illustration of the space-time model (for brevity, only one resource slice and eight time-slices are considered).

A. Demonstrating the Key Insight via a Space-time Model

It is observed that, resource isolation can reduce performance uncertainty, and resource sharing can increase resource utilization and overall throughput. Therefore, we exploit the combination of resource isolation and resource sharing to make E_S as small as possible.

The state-of-the-art resource scheduling strategies [7, 8, 16, 27, 36] used resource isolation techniques to guarantee the QoS. That is, each application can only use the resources allocated to itself but cannot use the resources allocated to other applications. However, the resource isolation in these strategies leads to low resource utilization.

Take the processing unit resources as an example. We assume that only 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 we allocate only one core to the LC application, the tail latency of the LC applications will violate the QoS target, since the service rate is $0.8U$, which is less than U. However, if two cores are allocated to the LC application, the QoS target of the LC application can be met, but it would degrade the throughput of the BE application due to the waste of resources.

As shown in Figure 4, a space-time model is presented to illustrate different resource scheduling schemes. For brevity, we only consider two LC applications (i.e., LC_1 and LC_2) and one BE application (i.e., BE), and examine the usage of only one resource-slice (e.g., one processing unit or one LLC way) and eight time-slices. There are three different scenarios.

In scenario (a), each application is running alone, so we can know exactly the space-time resource requirement of each application. For a time-slice, when there exist two or more ticks, resource contention will occur. For instance, in time-slice 6, all the three applications need the same resource-slice, thus resource conflict occurs.

In scenario (b), the resource-slice is isolated, and is exclusively allocated to LC_1, so only LC_1 can use the resource-slice, guaranteeing the QoS 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, incurring resource waste.

In scenario (c), the resource is shared among all the applications although the LC applications take precedence over the BE applications. At the beginning of time-slice 3, the resource owner is changed from LC_1 to BE, increasing the throughput of BE. Meanwhile, it is noteworthy that the change of the resource ownership is not free, due to the context switching overhead and/or the cache pollution. The triangle represents that the resource-slide can boost the application performance with 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 (c) with (b), the number of crosses is reduced from 10 to 6 and there are four more triangles in scenario(c), and the resource utilization ratio has been almost doubled. The key insight is that, although resource isolation is an effective means for reducing performance uncertainty, resource sharing is crucial for improving system utilization. Therefore, in terms of the overall user experience, neither complete isolation nor sharing is the optimal strategy, and we need to simultaneously harvest the advantages of both isolation and sharing.

B. Design of the ARQ Strategy

A resource region includes a number of cores and cache ways. ARQ divides resources into shared and isolated regions based on the aforementioned key insight. Each LC application can use not only the resource of its own isolated region, but also the resources of the shared region, while the BE application can only run in the shared region. If an LC application running in the shared region can satisfy its QoS target, the resources of the isolated region will be reduced to 0, indicating that it can safely share resources with other applications. Once the QoS of an LC application is severely interfered with while running in the shared region, the ARQ strategy will detect this interference, and gradually increase the resources of its isolated region until the QoS target is satisfied.

Algorithm 1 shows the ARQ strategy. ARQ periodically (e.g., every 500ms [8], 1s [33] or 2s [36]) monitors the tail latency of each LC application and the IPC of each BE application to calculate ReT of each LC application and E_S. Then, ARQ adjusts resource allocation according to ReT and evaluates the effectiveness of the adjustment by E_S. If the adjustment increases E_S, we cancel the adjustment and try to take new adjustment action to avoid trapping in local optimum, that is, do not allow the old adjustment to occur again in the next 60s.

Algorithm 1 ARQ Resource Scheduling Algorithm.

```
function ARQ
    $isAdjust$ ←False, $E_S \leftarrow 1$
    while True do
        Monitor the tail latency values of the LC applications and the IPC values of BE applications periodically
        $E'_S \leftarrow E_S$
        $E_S \leftarrow$ computeEntropy()
        // $ReT$ is an array, the elements of which are the remaining tolerance of each LC application.
        $ReT \leftarrow$ computeRemainingTolerance()
        if $isAdjust$ and $E_S > E'_S$ then
            Cancel the last adjustment and do not allow the last victim region to be penalized in the next 60s.
            $isAdjust \leftarrow$ False
        else
            $isAdjust \leftarrow$ AdjustResource($ReT$)
        end if
    end while
end function

function AdjustResource
    $victimRegion \leftarrow$ findVictimRegion($ReT$)
    $beneficiaryRegion \leftarrow$ findBeneficiaryRegion($ReT$)
    // Choose one type of the resources (i.e., core, LLC, or memory bandwidth, etc) of $victimRegion$.
    $\Delta R \leftarrow$ findVictimResource($victimRegion$)
    Move one unit resource of type $\Delta R$ from the $victimRegion$ to the $beneficiaryRegion$
    return whether the resource has been actually adjusted
end function

function findVictimRegion
    for each $ReT_i$ in descending order do
        if $ReT_i > 0.1$ and application $i$ has isolated resource that allows to be penalized then
            return the isolated region of application $i$
        end if
    end for
    return the shared region
end function

function findBeneficiaryRegion
    Identify the application $i$ that has the smallest ReT.
    if $ReT_i < 0.05$ then
        return the isolated region of application $i$
    else
        return the shared region
    end if
end function
```

In the *AdjustResource* function, the goal is to move one slice of resource from a rich region to a poor region, hopefully decreasing E_S. We determine the victim and beneficiary regions by the *findVictimRegion* and the *findBeneficiaryRegion* functions according to the ReT array which records the ReT of each LC application. Then, using the *findVictimResource*, we determine which type of resources will be moved or penalized. Then, we move the selected resource from the victim region to the beneficiary region.

In the function *findVictimResource*, we maintain a finite state machine which is as same as that in [8] to determine the order of resource adjustment. Each state of the state machine represents a resource type (e.g., processing units, LLC capacity, and memory bandwidth). The function will turn to the next type when the current resource type cannot be penalized.

The function *findVictimRegion* takes the ReT array as input and outputs the victim region which donates resources to other regions. It traverses the ReT array in descending order to identify the application whose ReT is larger than 0.1. An application with a large ReT may not have isolated resources, so we need to traverse the ReT array in descending order to determine the victim region. 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 concern 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, the shared region will be the beneficiary region.

If the victim and beneficiary regions are both shared regions, no LC applications need more resources and no LC applications can donate resources, thus an equilibrium has been reached and the resources adjustment will not be enforced.

Monitoring interval is configured to be 500ms, which is consistent with that of PARTIES (see Section 4.3 in [8]). We find that smaller interval allows the scheduler to detect and react to QoS violation more timely, but tail latency becomes less stable, and increases the difficulty to accurately calculate the tail latency. Larger intervals ease tail latency calculations, but each QoS violation will last for a longer period. We find 500ms to be a practically suitable interval from evaluation.

C. Allocation Comparison

ARQ combines the benefits of resource sharing and isolation, sharing resources among the applications that have high ReT, and isolating the applications that have low ReT. In the following, we present two snapshots to illustrate and compare the allocation processes of ARQ and PARTIES, when the load of Xapian is low and high, respectively. The experimental setup will be described in Section V.

Figure 5 shows the resource allocation snapshot when the load of Xapian is 30%. Compared with PARTIES, ARQ makes the BE application (i.e., Stream) have more available resources. PARTIES allocates the isolated resources for each application to preferentially reduce E_{LC}. However, the user experience of the BE application is low because it can only use 10% processing unit and 30% LLC ways. ARQ finds that sharing resources among all the applications except Xapian can also make E_{LC} 0. Therefore, in ARQ strategy, Xapian was allocated isolated resources (10% processing unit and 25% LLC ways) to isolate interference, while Img-dnn and Moses shared the shared region resources with the BE application. Although E_{LC} of PARTIES and ARQ are both 0, ARQ is better since it achieves a much lower E_{BE}.

Figure 6 shows the resource allocation snapshot when the load of Xapian is 90%. Compared with PARTIES, ARQ makes the high-load LC application (i.e., Xapian) have more available resources, since the other LC applications can be satisfied only with the shared region resources. Given the high load of Xapian, both PARTIES and ARQ want to allocate more isolated resources for Xapian. However, to simultaneously satisfy the QoS targets of Moses and Img-dnn, PARTIES allocates 50% cores and 60% LLC ways while ARQ only allocates 30% cores and 35% LLC ways by sharing resources among applications. As a result, PARTIES can only allocate 50% cores and 40% LLC ways to Xapian which is not enough to satisfy the QoS target of Xapian, while ARQ allocates 70% cores and 65% LLC ways to Xapian which can significantly reduce E_{LC} and E_S. We will evaluate ARQ to compare it with the state-of-the-art strategies in detail in Section VI.

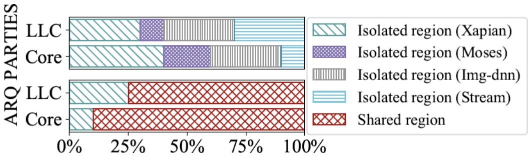


Fig. 5. A snapshot of the resource allocation of PARTIES and ARQ (Xapian (30%), Moses (20%), Img-dnn (20%) and Stream). Compared with PARTIES, ARQ makes the BE application (i.e., Stream) have more available resources (from the shared region).

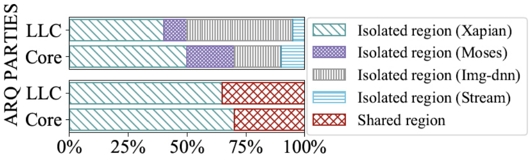


Fig. 6. A snapshot of the resource allocation of PARTIES and ARQ (Xapian (90%), Moses (20%), Img-dnn (20%) and Stream). Compared with PARTIES, ARQ makes the high-load LC application (i.e., Xapian) have more available resources, since the other LC applications can be satisfied only with the shared region resources.

D. Overhead Comparison

Like other QoS-aware scheduling strategies [8, 27], ARQ involves overhead from two parts: monitoring the system state (e.g., the tail latency and IPC of each application), and allocating system resources (e.g., cache, processor core) periodically. Unlike ML-based scheduling strategies [33, 36], ARQ does not require complex computations of resource allocations, so there is negligible overhead of computing the resource allocation. The monitoring overhead is negligible since we only read a few counters every 500ms (see the last paragraph in Section IV-B for discussion on the monitoring interval). The overhead of resource adjustment mainly comes from warmup of cache ways for cache re-partitioning and context switching for core re-assignment, and it depends on the frequency of resource re-adjustment.

When the contention is high, PARTIES can result in ping-ponging effects between severely resource-starved applications, which incurs overhead everytime resources are switched. Moreover, due to the long queues that have been built up in the system, core allocation in PARTIES may would need more than 500ms to take effect. Compared to PARTIES, ARQ has much less ping-ponging effects, because ARQ applies the shared region as a resource pool which provides more resources for LC applications, reducing the likelihood of QoS violations.

Our evaluations have involved the overhead of ARQ mentioned above, and the experimental results that will be presented in next sections show that ARQ has small overhead, achieving much less QoS violations than PARTIES (see Figure 8, 9 and 10).

V. Experimental Methodology

We conduct experiments on a real server of a datacenter. Table III shows our experimental platform. We use the *taskset* command to set the core affinity for each application and use Intel's Cache Allocation Technology (CAT) [17, 19] to allocate the LLC for each core. CAT allows for a given number of ways to be assigned to a specific application to limit the amount of LLC space the application can occupy. Consistent with previous studies [8], we disabled Hyper-Threading in our experiments. We evaluate the scheduling strategies with several application combinations. Each combination contains multiple LC and BE applications from different domains.

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)

Xapian is a search engine that is widely used in popular websites and software frameworks. In our experiments, the search index is built from a dump of the English version of Wikipedia, and query terms are chosen randomly, following a Zipfian distribution [2, 15]. **Moses** is a statistical machine translation system. We drive Moses using randomly chosen dialogue snippets from the English-Spanish corpus [45]. **Img-dnn** is a handwriting recognition application. We drive the application using randomly chosen samples from the MNIST database [13]. **Masstree** [28] is a scalable in-memory key-value store. We drive Masstree using a modified version of the Yahoo Cloud Serving Benchmark [9, 23]. **Sphinx** [50] is an accurate speech recognition system. **Silo** [46] is a in-memory transactional database. These LC applications are from Tailbench [23] and are instantiated with 4 threads.

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

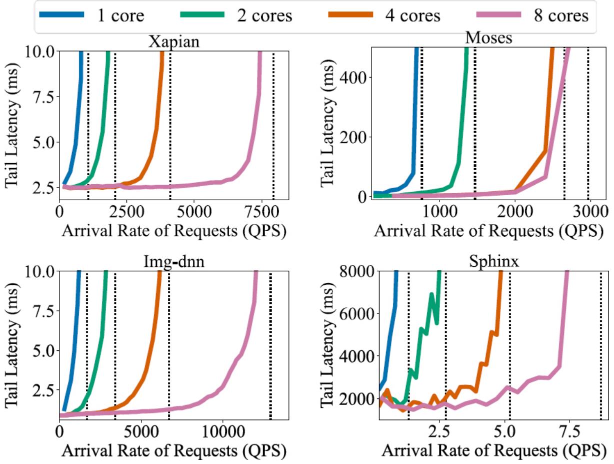


Fig. 7. The relationship between tail latency and arrival rate of requests with 1, 2, 4 and 8 processing units (the dashed lines denote the maximal service rate under varying core counts).

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 95^{th} percentile tail latency is used without losing generality. As shown in Figure 7, the lines of different colors correspond to the number of processor cores as 1, 2, 4, and 8. For each LC application, as the arrival rate of requests gradually increases, tail latency increases slowly at the beginning. When the arrival rate of requests exceeds a certain threshold, the tail latency increases exponentially. Similar to previous research [8, 36], 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 Eq. (1)), 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.

We run different BE applications in our experiments: Fluidanimate, Stream and Streamcluster, respectively. **Fluidanimate** and **Streamcluster** are taken from PARSEC benchmark suite [3]. Fluidanimate conducts a liquid simulation that uses a computational method to solve the Navier-Stokes equation. Streamcluster solves the online clustering problem. Like the LC applications, Fluidanimate and Streamcluster are both instantiated with 4 threads. **Stream** [32] is a well-known 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 units, LLC and memory bandwidth, we instantiate 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 relies on the real-time scheduling strategy of the operating system (i.e., round-robin). It sets the LC applications to the real-time priority. When the real-time process is ready, if the current core is running a non-real-time process, the real-time process immediately preempts the non-real-time process.

PARTIES [8]: This strategy leverages hardware and software resource partitioning technology to adjust resource allocations dynamically. It strictly partitions resources between collocated applications without resource sharing. It calculates the slack of multiple LC applications during a fixed time interval and determines whether resources need to be upsized or downsized according to the slack of each LC application. In this way, it ensures that the QoS targets of the LC applications are not violated.

CLITE [36]: This strategy is also based on resource isolation. It uses Bayesian optimization to identify or predict desirable resource allocations, and builds a predictive model for different resource partitioning configurations by sampling several points in large configuration space.

VI. 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.

A. The Case of Constant Load

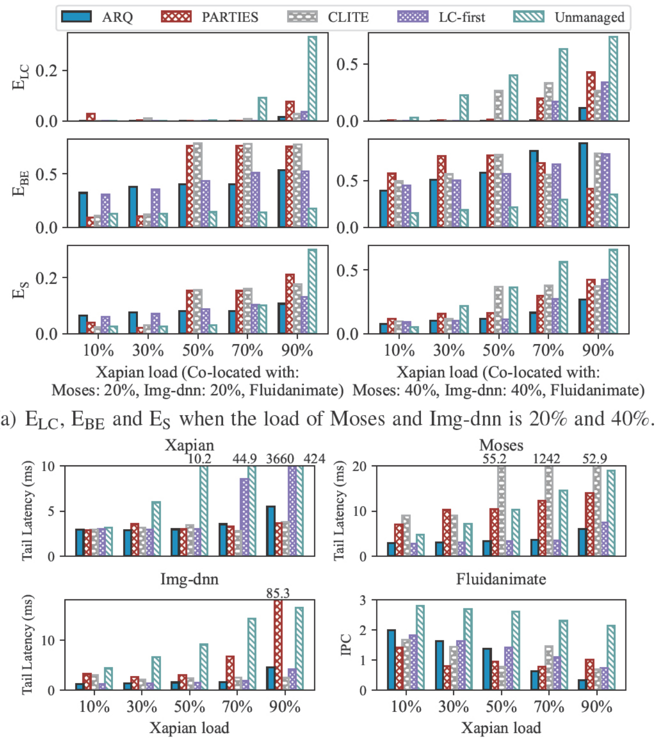
Collocated with Fluidanimate: In this experiment, we concurrently run three LC applications (i.e., Xapian, Moses, and Img-dnn) and one BE application (i.e., Fluidanimate), and the load of the LC applications is constant.

Figure 8 shows E_{LC}, E_{BE} and E_S of different strategies when the load of Moses and Img-dnn is 20% (left) and 40% (right) of the max load, respectively, and Xapian's load varies from 10% to 90%.

When the load of the LC applications is low, the Unmanaged strategy achieves the lowest E_S among all the strategies, showing the benefits of resource sharing. The reason is that the interference between applications is not severe at this time, and resource sharing can achieve higher resource utilization than other strategies. However, when the load is high, despite low E_{BE}, the rapid increase in E_{LC} makes E_S also increase rapidly, since the Unmanaged strategy does not take any action to guarantee the QoS of the LC applications.

Compared with the Unmanaged strategy, the LC-first strategy allows the LC applications, to preempt the resources of the BE applications if needed. Although 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 use complete resource isolation to mitigate interference among applications and satisfy the QoS of LC applications. When the load of the LC applications is low, many resources are allocated for the BE application with the premise of guaranteeing the QoS of the LC applications, which leads to low E_{BE} and E_S. When the load is high (e.g., the load of Moses and Img-dnn is 20%, respectively, the Xapian's load is larger than 50%), they allocate most resources to the LC applications but few resources to the BE application, which incurs high E_{BE} and E_S.



(a) E_{LC}, E_{BE} and 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%.

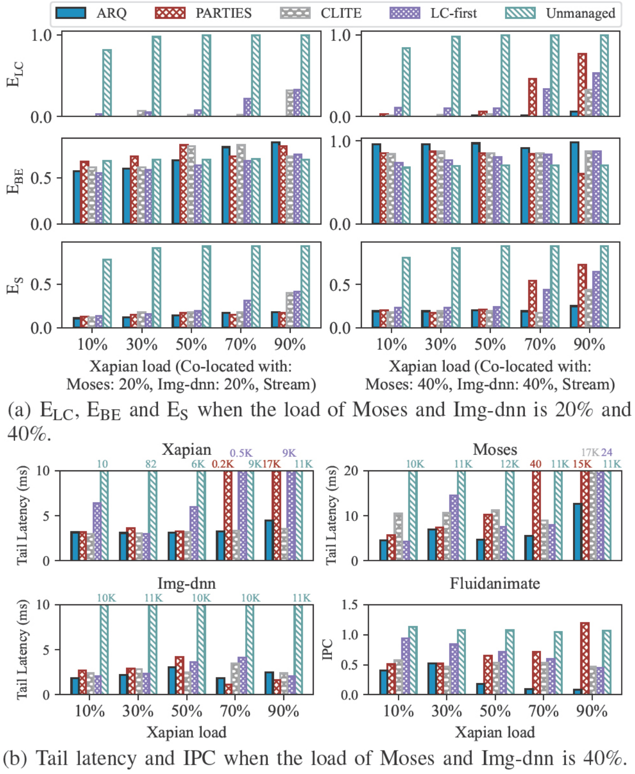
Fig. 8. Results when Xapian, Moses, Img-dnn and Fluidanimate are collocated.

As shown in Figure 8(a), ARQ achieves the lowest E_S among all the strategies. ARQ reduces E_{LC} more significantly than other strategies, implying that QoS of the LC applications has been guaranteed preferentially. ARQ has the lowest E_{BE} during most time among all the strategies based on resource isolation. When the load is extremely high, it is reasonable that ARQ has higher E_{BE} than other strategies, because ARQ lets 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 of all applications.

Figure 8(b) shows more detailed data regarding the tail latency and IPC for one scenario (i.e., when the load of Moses and Img-dnn is 40%). Taking the Unmanaged as the baseline, ARQ reduces the tail latency by 66.5% on average, while CLITE reduces by 43.6% and PARTIES reduces by 37.2%. When the load is low (i.e., Xapian's load $\leqslant$ 50%), compared with PARTIES and CLITE, ARQ increases IPC by 63.8% and 37.1%, respectively. When the load is pretty high (i.e., Xapian's load $\geqslant$ 70%), ARQ preferentially optimizes tail latency rather than IPC, and allocates resources to guarantee the QoS of the LC applications.

Collocated with Stream: In this experiment, we instantiate Stream with 10 threads to represent another type of severe interference among applications. Figure 9 shows E_{LC}, E_{BE} and E_S of each strategy and detailed tail latency and IPC.

Neither the Unmanaged nor the LC-first strategy can satisfy the QoS of the LC applications even if the load is low, resulting



(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. 9. Results when Xapian, Moses, Img-dnn and Stream are collocated.

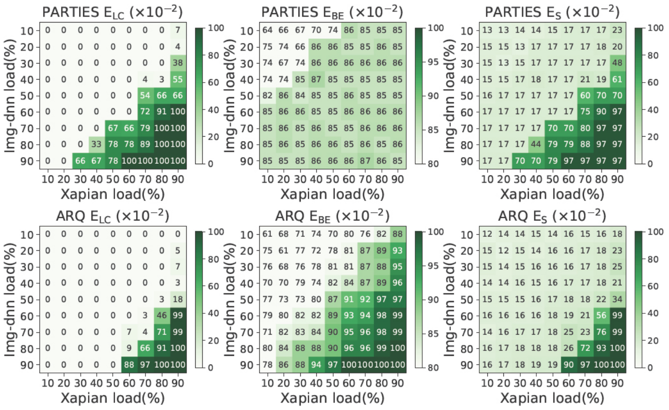


Fig. 10. $\mathrm{E_{LC}}$, $\mathrm{E_{BE}}$ and $\mathrm{E_S}$ when Xapian, Moses, Img-dnn and Stream are collocated (the load of Moses is fixed to 20%).

in high $\mathrm{E_{LC}}$ and $\mathrm{E_S}$. When the load of the LC applications is low (e.g., Xapian's load ⩽ 50%, and the load of Moses and Img-dnn is 20%, respectively), the other four strategies except the Unmanaged strategy can maintain low $\mathrm{E_{LC}}$. When the load of Xapian is 70%, and the load of Moses and Img-dnn is 40%, respectively, only CLITE and ARQ can eliminate $\mathrm{E_{LC}}$ and $\mathrm{E_S}$ to an ideal level. When the load of LC applications is pretty high (i.e., the load of Xapian is 90%, and the load of Moses and Img-dnn is 40%, respectively), only ARQ can achieve low $\mathrm{E_{LC}}$ and $\mathrm{E_S}$.

When the load of Img-dnn and Moses is 40%, respectively, and the load of Xapian is 90%, neither PARTIES nor CLITE can find a feasible resource allocation to satisfy the QoS of the LC applications. Nevertheless, ARQ has reduced $\mathrm{E_{LC}}$ to nearly zero (i.e., 0.06). Moreover, ARQ reduces $\mathrm{E_S}$ by 73.4% (i.e., from 0.94 to 0.25), while CLITE and PARTIES reduce $\mathrm{E_S}$ by 53.2% and 22.3%, respectively.

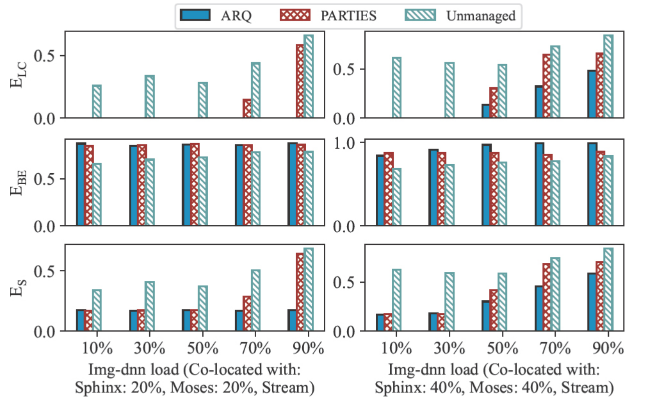


Fig. 11. $\mathrm{E_{LC}}$, $\mathrm{E_{BE}}$ and $\mathrm{E_S}$ when Img-dnn, Moses, Sphinx and Stream are collocated (the load of the LC applications is constant).

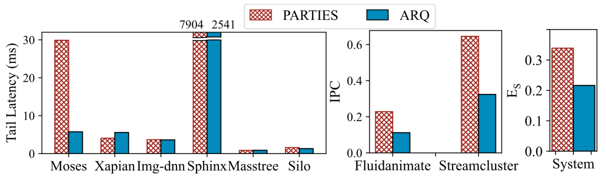
In the experiments above, ARQ achieves the highest yield (the ratio of satisfactory LC applications) and the lowest system entropy. Compared with PARTIES and CLITE, ARQ increases the yield by 25% and 20% respectively (from 60% and 65% to 85%). ARQ reduces $\mathrm{E_S}$ by 36.4% and 33.3% respectively (from 0.22 and 0.21 to 0.14).

Collocation with diverse loads: In this experiment, while the load of Moses is fixed to 20%, the load of Xapian and Img-dnn both varies from 10% to 90% when collocated with Stream to comprehensively show the benefits brought by ARQ. Figure 10 shows the entropy heatmap of $\mathrm{E_{LC}}$, $\mathrm{E_{BE}}$ and $\mathrm{E_S}$ when either PARTIES or ARQ scheduling strategy is used.

By adopting the shared region, ARQ allocates more resources for BE applications when the load of LC applications is low (e.g., $\mathrm{E_{LC}}$ equals 0), thus leading to lower $\mathrm{E_{BE}}$. When the load is low (i.e., for the top-left region of each subgraph), more resources are allocated to the shared region since the QoS target of LC applications can be easily satisfied with small private regions. BE applications can obtain more resources from the shared region compared to PARTIES and achieve lower $\mathrm{E_{BE}}$ (details are shown in the two subgraphs in the middle of Figure 10). When the load is high (i.e., for the bottom-right region of each subgraph), LC applications can obtain more resources from the shared region, thus leading to lower $\mathrm{E_{LC}}$ at the expense of higher $\mathrm{E_{BE}}$. The detailed reasons are described in Figure 5 and Figure 6 in Section IV-C.

Another application collocation: We use another combination of applications (Img-dnn, Moses and Sphinx as the LC applications and Stream as the BE application) to further evaluate the scheduling strategies.

Figure 11 shows $\mathrm{E_{LC}}$, $\mathrm{E_{BE}}$ and $\mathrm{E_S}$ of different strategies when the load of Moses and Sphinx is 20% (left) and 40% (right) of the max load, respectively, and Img-dnn's load varies from 10% to 90%. When the load is low, $\mathrm{E_S}$ of ARQ is almost as the same as that of PARTIES. When the load is high,



(a) LC applications' performance (b) BE applications' performance (c) E_S

Fig. 12. Tail latency, IPC and E_S when 6 LC applications and 2 BE applications are collocated (the load of the LC applications is 20%).

compared with the PARTIES, ARQ guarantees the QoS target of the LC applications, reducing E_S by 40.93% on average.

Collocation of even larger number of applications: In this experiment, to further evaluate the effectiveness and robustness of the strategies, we double the number of collocated applications. We concurrently run 6 different LC applications (Moses, Xapian, Img-dnn, Sphinx, Masstree and Silo from Tailbench) and 2 different BE applications (Fluidanimate and Streamcluster from PARSEC). Figure 12 shows tail latency and IPC when the load of each of the LC applications is 20%.

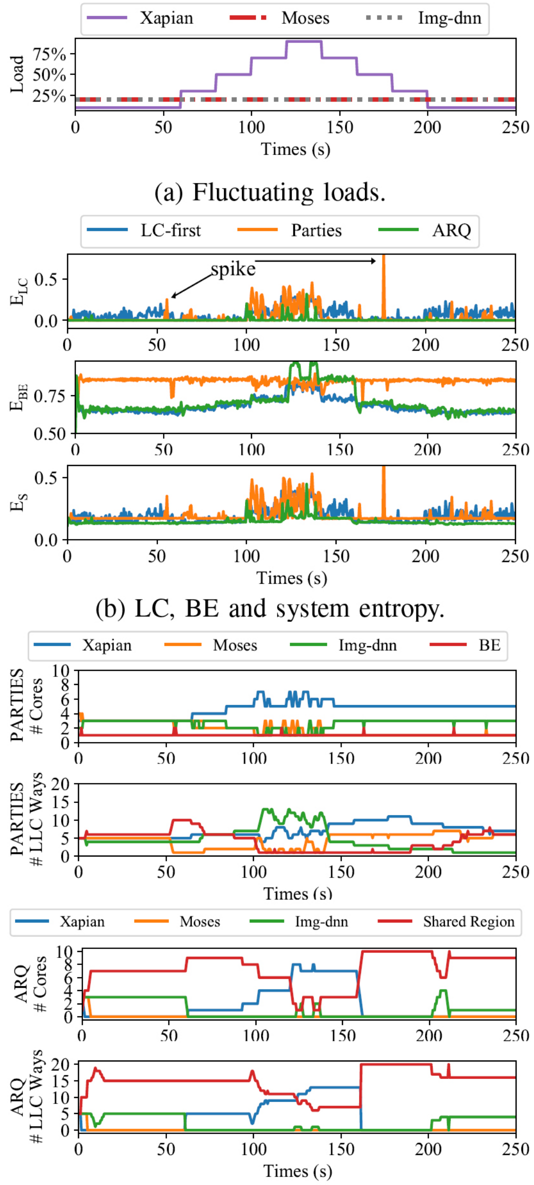
As shown in Figure 12, the number of collocated applications has been doubled on the same datacenter, and resource contention becomes even more severe. Compared with PARTIES, ARQ drastically reduces the tail latency of Moses and Sphinx (from 29.88 to 5.75 ms, and from 7904 to 2514 ms, respectively) at the cost of a slight increase on the tail latency of Xapian (from 4.06 to 4.17 ms, still satisfying the QoS target according to Table IV), thus reducing E_S significantly. Compared to PARTIES, ARQ reduces E_S by 36.4% (from 0.33 to 0.21). Considering Figure 8 and 12, we can conclude that the scalability of ARQ is very well.

B. The Case of 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 [6], in this section, we evaluate different strategies with a fluctuating load. We still choose Xapian, Moses and Img-dnn as LC applications, and Stream as BE applications. We set the load of Moses and Img-dnn as 20% and vary the load of Xapian from 10% to 90%. Figure 13(a) shows how Xapian's load fluctuates. Figure 13(b) shows the changes of E_{LC}, E_{BE} and E_S for LC-first, PARTIES and ARQ strategies. Figure 13(c) shows how ARQ and PARTIES dynamically schedule resources to adapt to load fluctuations.

Figure 13 shows the data during 250 seconds (i.e., 500 data points). During this process, ARQ has 59 tail latency violations, while PARTIES has 105 tail latency violations. These tail latency violations are mainly due to the resource adjustment after the load fluctuations.

In the beginning, the load of all the three LC applications is low, and thus both PARTIES and ARQ can satisfy the QoS target of all the LC applications. PARTIES only allocates 1 processing unit and 6 LLC ways to the BE application, while ARQ allocates 7 processing units and 15 LLC ways to the shared region. Consequently, compared with PARTIES, ARQ has reduced E_{BE} by 22.3% (i.e., from 0.85 to 0.66), and thus the user experience of the BE applications has been improved significantly.



(a) Fluctuating loads.

(b) LC, BE and system entropy.

(c) The resource allocation of PARTIES and ARQ.

Fig. 13. E_{LC}, E_{BE} and E_S and the corresponding scheduling process of LC-first, PARTIES, and ARQ (Xapian's load is fluctuating).

During the 100s-120s, Xapian's load is increased to 70%. PARTIES fails to find an allocation to satisfy the QoS target, leading to high E_{LC}. ARQ succeeds in the exploration to find an allocation that satisfies the QoS target. Although it causes some increase in E_{BE}, it deserves and is reasonable because the overall user experience in terms of E_S has been improved significantly. During 120s-140s, Xapian's load is increased to 90%; although neither PARTIES nor ARQ can reduce E_{LC} to 0, ARQ has much lower E_{LC} and E_S than PARTIES.

In Figure 13, there are some spikes in the E_{LC} curve of PARTIES, because PARTIES tentatively downsizes the resources of an LC application to maximize the resources of the BE application. If the LC application no longer satisfies the QoS target after downsizing, it would immediately recover from the previous incorrect downsize action. As shown in Figure 13, ARQ effectively mitigates the spiking phenomenon, even though it has a downsize action that is more aggressive than that of PARTIES.

ARQ eliminates the spiking phenomenon by occupying the resources of the shared region. When the load of the LC applications increases and the available resource is insufficient, PARTIES gradually allocate more resources to satisfy the QoS target. In ARQ, to avoid the rapid rise of the tail latency, the LC applications quickly preempt the resources in the shared region from the BE applications. Although this would harm the throughput of the BE applications, it is worthwhile because it guarantees the QoS of the LC applications.

VII. Related Work

Interference Characterization: Scott et al. [48] proposed using the service rate under interference and the duration of time interference lasts to characterize interference. Prior work [4, 20, 31, 38, 43, 55] used the values of IPC or execution time before and after the applications have been interfered with to quantify the interference. However, for the LC applications, users do not concern about IPC, and the change of IPC may be caused by interference from other applications or by fluctuations in their load. Therefore, it is not appropriate to use IPC to quantify the interference for LC applications. Many researchers [10, 11, 30, 53, 57] use the tail latency before and after the interference to quantify the interference of the LC applications. However, the ideal tail latency for different applications varies greatly (from the microsecond level to the second level). Hence, we propose E_S to unify the interference of different LC and BE applications. As a measure of interference, E_S has interpretability and measurability, and its analytical expression has all the required properties. Therefore, E_S is more formal, reasonable, and systematical than the ad hoc metrics [44, 47, 48]. There may be applications that care about both latency and IPC. In that case, we cloud either choose a more critical performance metric, or come up with an aggregated metric that takes various metrics into account. It is a challenging scenario even without colocation, and we will leave it as future work.

Resource Scheduling: How to schedule resources to satisfy the QoS target of different types of applications in a datacenter is a vital issue. Previous studies used software and hardware level resource isolation techniques to manage resources to eliminate interference on specific resources. Many feedback-based resources managers have been proposed to detect and respond to QoS violations using application state information (such as tail latency and input load). Heracles [27] collocates the LC applications and the BE applications safely using a threshold-based method to manage interference. PARTIES [8] dynamically adjusts the resource allocation of each application by monitoring the tail latency to further improve the resource utilization. CLITE [36] uses Bayesian optimization to explore resource sensitivity to find an allocation with optimal performance. Sturgeon [34] uses decision trees and binary search to find out an allocation that can satisfy power consumption constraints and QoS targets. Twig [33] uses multi-agent deep reinforcement learning to improve the energy efficiency of multiple LC applications. Although CLITE, Sturgeon and Twig can all coordinately schedule multiple resources in one step, they have limitations. Specifically, Sturgeon relies on prior application knowledge and offline pre-training; CLITE and Twig involve large amount of computations at runtime to find the best allocation among a large pool of candidate allocations, incurring more overhead and potentially worsening applications' performance. Besides, CuttleSys [25] regularly evaluates the effect of current allocation and makes decisions to adapt to the changes of the applications by collaborative filtering and dynamically dimensioned search. Sinan [56] uses CNN and Boosted Tree to predict end-to-end latency and QoS violation probability based on historical system information. Stretch [29] proposes a method to statically partition the ROB and LSQ capacity resources of collocated tasks. They all perform complete resource isolation for all the applications, but have not explored the opportunities of sharing resources at the right time to maximize resource utilization and system throughput. Dunn [39] also uses CAT to partition the cache. However, Dunn cares more about system fairness while ARQ focuses on both fairness (between LC and BE applications) and overall system performance.

VIII. Conclusions

As cloud workloads are rapidly changing, it is challenging to achieve the perfect match between applications and the underlying architecture in a datacenter. However, it is crucial for a datacenter to simultaneously achieve high task concurrency (for high resource utilization) and high QoS (in terms of yield and IPC). In this study, we present the Ah-Q which includes a theory and a strategy to address this issue. Specifically, we have proposed system entropy, E_S, a holistic and analytical solution to the problem of quantifying the interference incurred by resource contention in a datacenter. We have proposed the ARQ algorithm to harvest the benefits of resource isolation and sharing. We demonstrate the correctness and effectiveness of E_S and ARQ on the platform of a real datacenter. Extensive experiments validated that E_S is correct and useful, and the associated ARQ strategy has improved the overall user experience significantly. We also show that E_S and ARQ are easy-to-use and robust in diverse scenarios.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. We 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. This work is supported in part by the National Natural Science Foundation of China (No. 62090023, 61772497) and National Key RD Program of China (No. 2016YFB1000201).

References

[1] "Redis," April 2022. [Online]. Available: Redis.io

[2] R. Baeza-Yates, "Applications of web query mining," in *Proceedings of the European Conference on Information Retrieval*. Springer, 2005, pp. 7–22.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[4] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 318–329.

[5] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for mapreduce applications in virtual clusters," in *Proceedings of the 22nd International symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013, pp. 227–238.

[6] M. C. Calzarossa, M. L. Della Vedova, L. Massari, D. Petcu, M. I. Tabash, and D. Tessera, "Workloads in the clouds," in *Principles of Performance and Reliability Modeling and Evaluation*. Springer, 2016, pp. 525–550.

[7] Q. Chen, S. Xue, S. Zhao, S. Chen, Y. Wu, Y. Xu, Z. Song, T. Ma, Y. Yang, and M. Guo, "Alita: comprehensive performance isolation through bias resource management for public clouds," in *SC20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[8] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

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

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

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

[13] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[14] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 104–117.

[15] D. G. Feitelson, *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[16] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[17] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, 2011.

[18] J. Guo, Z. Chang, S. Wang, H. Ding, and Y. Bao, "Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces," in *the International Symposium*, 2019.

[19] Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.

[20] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[21] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 598–610.

[22] 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.

[23] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[24] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens *et al.*, "Moses: Open source toolkit for statistical machine translation," in *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, 2007, pp. 177–180.

[25] 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 *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 650–664.

[26] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 301–312.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[28] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 183–196. [Online]. Available: https://doi.org/10.1145/2168836.2168855

[29] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing qos and throughput for colocated server workloads on smt cores," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 15–27.

[30] 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 *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 248–259.

[31] J. Mars, L. Tang, and M. L. Soffa, "Directly characterizing cross core interference through contention synthesis," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011, pp. 167–176.

[32] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19–25, 1995.

[33] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 167–179.

[34] P. Pang, Q. Chen, D. Zeng, C. Li, J. Leng, W. Zheng, and M. Guo, "Sturgeon: Preference-aware co-location for improving utilization of power constrained computers," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 718–727.

[35] 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 *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[36] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[37] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, "Phase-aware cache partitioning to target both turnaround time and system performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2556–2568, 2020.

[38] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki, "Fact: a framework for adaptive contention-aware thread migrations," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011, pp. 1–10.

[39] 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 *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 194–205.

[40] C. E. Shannon, “A mathematical theory of communication,” *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[41] A. Snavely and D. M. Tullsen, “Symbiotic job scheduling for a simultaneous multithreaded processor,” in *Proceedings of the ninth International conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 234–244.

[42] A. Sriraman, “Unfair data centers for fun and profit,” *Wild and Crazy Ideas (ASPLOS)*, 2019.

[43] 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 *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 62–75.

[44] N.-H. Sun, Y.-G. Bao, and D.-R. Fan, “The rise of high-throughput computing,” *Frontiers of Information Technology and Electronic Engineering*, vol. 19, no. 10, pp. 1245–1250, 2018.

[45] J. Tiedemann, “Parallel data, tools and interfaces in opus.” in *Lrec*, vol. 2012, 2012, pp. 2214–2218.

[46] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, “Speedy transactions in multicore in-memory databases,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 18–32. [Online]. Available: https://doi.org/10.1145/2517349.2522713

[47] S. Votke, J. A. Jaleel, A. Suresh, M. Delasay, S. Doroudi, and A. Gandhi, “Optimal markovian dynamic control of interference-prone server farms,” in *Proceedings of the 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 295–308.

[48] S. Votke, S. A. Javadi, and A. Gandhi, “Modeling and analysis of performance under interference in the cloud,” in *Proceedings of the 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2017, pp. 232–243.

[49] M. M. Waldrop, “The chips are down for moore's law,” *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[50] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Wölfel, “Sphinx-4: A flexible open source framework for speech recognition,” *Sun Microsystems*, 12 2004.

[51] 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 Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 13, no. 1, pp. 1–25, 2018.

[52] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, “Small is better: Avoiding latency traps in virtualized data centers,” in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[53] H. Yang, A. Breslow, J. Mars, and L. Tang, “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[55] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi2: Cpu performance isolation for shared compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.

[56] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: Ml-based and qos-aware resource management for cloud microservices,” ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 167–181. [Online]. Available: https://doi.org/10.1145/3445814.3446693

[57] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao, “Rhythm: component-distinguishable workload deployment in datacenters,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–17.