Qin: Unified Hierarchical Cluster-node Scheduling for Heterogeneous Datacenters

Wenkai Guan, *Member, IEEE* and Cristinel Ababei, *Senior Member, IEEE*

Abstract—Energy efficiency is among the most important challenges for computing. There has been an increasing gap between the rate at which the performance of processors has been improving and the lower rate of improvement in energy efficiency. This paper answers the question of how to reduce energy usage in heterogeneous datacenters. It proposes a unified hierarchical scheduling using a D-Choices technique, which considers interference and heterogeneity. Heterogeneity comes from servers' continuous upgrades and the integrated high-performance "big" and energy-efficient "little" cores. This results in datacenters becoming more heterogeneous and traditional job scheduling algorithms become suboptimal. To this end, we present a two-level hierarchical scheduler for datacenters that exploits increased server heterogeneity. It combines in a unified approach cluster and node level scheduling algorithms, and it can consider specific optimization objectives including job completion time, energy usage, and energy-delay-product (EDP). Its novelty lies in the unified approach and in modeling interference and heterogeneity. Experiments on a research cluster found that the proposed approach outperforms state-of-the-art schedulers by around 10% in job completion time, 39% in energy usage, and 42% in EDP. This paper demonstrated a unified approach as a promising direction in optimizing energy and performance for heterogeneous datacenters.

Index Terms—datacenters, heterogeneity, scheduling, collaborative filtering, kalman filtering

1 Introduction

"Our challenge is to figure out over the next decade what we think about computer efficiency as the No. 1 priority." [1]. Datacenters use an estimated 200 terawatt hours (TWh) per year, representing around 1% of global electricity demand and contributing around 3% of all global carbon emissions, thereby exceeding emissions from commercial flights (about 2.4%) and other existential activities that fuel our global economy [2], [3]. According to the U.S. Energy Information Administration, datacenters represent more than 7% of total commercial electric energy usage, and we project that this number will increase [4]. Therefore, reducing energy usage and improving efficiency in datacenters has never been more critical; it is important not only for the cost to companies but for the environmental footprint of these datacenters as this computing domain rapidly expands [5].

Previous work focused on optimizations at multicore processor/server and datacenter levels [6]. Among such solutions, job scheduling, workload placement, power budgeting, and server farm management policies are popular means to save energy and improve performance. Both performance - as the execution time of jobs from arrival time to completion time - and power consumption are directly affected by the scheduling algorithms of jobs arriving at the datacenter and then delegated to individual servers. Another recent optimization avenue is the exploitation of hardware heterogeneity.

However, the state-of-the-art work in the literature seemed to focus primarily on optimizations at individual cluster or node levels and typically addressed either performance or energy. A question that remains is how we do scheduling in a unified cluster-node approach to reduce energy without degrading performance. In this paper, we answer this question by presenting Qin - a hierarchical approach for scheduling in heterogeneous datacenters. At the cluster level, we propose a cluster manager that has a global view of the scheduling (application-to-server) by considering different server configurations on servers, whose global view would be used for server level scheduling (thread-to-core scheduling). At the server level, we proposed a server manager that combines task scheduling and migration with dynamic voltage and frequency scaling (DVFS). Finally, we combined server and cluster levels scheduling algorithms, providing a unified two-level hierarchical approach. We reported the preliminary results of the proposed scheduler in a recent conference paper [7]. In this paper, we extend that work. Our main contributions are as follows:

- Problem definitions. We present a multi-objective optimization formulation for the unified cluster-node scheduling in heterogeneous datacenters. More specifically, we focus on application-to-server (cluster level) and thread-to-core (node level) problems to optimize the objective of energy-delay-product (EDP).
- Unified hierarchical approach. We propose a unified hierarchical cluster-node scheduling solution that directly considers interference and heterogeneity. The proposed scheduling outperforms state-of-the-art scheduling approaches.
- D-choices greedy technique. We propose the D-choices greedy technique, which is at the heart of the scheduling algorithms, to help select servers and cores.

W. Guan is with the Division of Science and Mathematics at the University of Minnesota, Morris MN, 56267, USA. C. Ababei is with the Department of Electrical and Computer Engineering, Marquette University, Milwaukee WI, 53233, USA (e-mail: guan0210@morris.umn.edu; cristinel.ababei@marquette.edu)

- Dynamic admission control. We introduce a dynamic admission control protocol to shorten applications' waiting time, improve job completion time, and reduce energy usage.
- Experiments on a research cluster. We conduct extensive experiments of the proposed Qin scheduler on a research cluster and a simulated platform with real-world and synthetic workloads.

The remainder of this paper is organized as follows. Section 2 reviews related work on scheduling. Section 3 introduces basics about techniques used later to develop the proposed scheduling approach. Section 4 presents the collaborative filtering based energy usage, interference, and heterogeneity estimations. Section 5 presents the D-choices greedy scheduling method. Section 6 presents the dynamic admission control protocol. Section 7 presents results from the evaluation of the Qin scheduler. Finally, Section 8 concludes the paper.

2 Related Work

In this section, we review the literature on optimization techniques at the node (server or multicore) and cluster (datacenter) levels. Table 1 summarizes the most relevant previous works.

In the category of node level approaches, the study in [8] proposed an imitation learning based node level scheduling for heterogeneous chip-multiprocessors systems. This work is important to this paper because it inspires our node level scheduling. The study in [9] presented dynamic energy management under performance constraints in chip multiprocessors using dynamic voltage and frequency scaling (DVFS). This work relied on the default scheduler inside the Sniper simulator [68]. This work was further improved on by the study in [10], which proposed a node-level deep neural network (DNN) based energy optimization method under performance constraints but based on using the Sniper's scheduler. The work in [11]–[14] optimized power for latency-critical workloads: the study in [11] proposed Rubik, which used a statistical performance model for a fine-grain DVFS scheme; the work in [12] presented Adrenaline that leveraged finer granularity voltage boosting for tail latency control; the study in [13] proposed Gemini for DVFS based power management; the work in [14] presented ReTail, which learned simplicity for request-level power management. The work in [15] enhanced the default Linux scheduler for node-level scheduling. The study in [16] introduced the adaptive slow-to-fast scheduling framework that matches the workload's heterogeneity to the hardware's heterogeneity. The work in [17] demonstrated the potential of reconfigurable cores for servers running latency-critical applications. The work in [18] used hardware and software resource partitioning mechanisms for QoS-aware co-scheduled workloads. The study in [19], [20] laid the foundation for large-scale collaborative filtering recommender systems on multi-GPUs and significantly outperformed the state-of-the-art approaches on shared memory platforms. The work in [21] proposed a two-stage hybrid energy allocation method for parallel application scheduling.

In the category of cluster level approaches, the work in [22] proposed Paragon, an online and scalable datacenter scheduler that is heterogeneity and interference aware. The work in [23] exploited the characteristics of deep learning workloads and proposed an introspective cluster scheduling framework to improve the latency and efficiency of deep learning jobs. The study in [24] proposed Gavel, a heterogeneity-aware scheduler that systematically generalizes scheduling policies for heterogeneous datacenters. The work in [25] leveraged d choices technique to develop algorithms for better load balancing in MapReduce deployed on heterogeneous servers. None of the above cluster-level scheduling approaches considered energy usage in their optimization. The work in [26], [27] co-located multiple latency-critical jobs while meeting the QoS requirements and improving utilization and performance. The work in [28], [29] hybridized GPU and FPGA to improve the overall throughput scalability and energy proportionality while guaranteeing the QoS. The work in [30] used deep reinforcement learning for cooling optimization and compute-intensive job allocation in datacenters. The research in [31] leveraged approximate computing to boost the utilization of shared servers. The researchers in [32]–[34] managed resources for machine learning inference serving from the scheduler's perspective [33], system's perspective [34], and combined [32]. The researchers in [35] dealt with fluctuating workloads for machine learning inference serving. The work in [36], [37] laid the foundation for minimizing energy costs for heterogeneous datacenters by considering geographical workload distribution, renewable energy, cooling subsystem, etc., and significantly reduced the energy cost of datacenters compared with state-of-the-art work. The study in [38]–[40] also optimized energy in datacenters.

TABLE 1: Review of cluster and node level scheduling.

Methods	Level	Interference	Heterogeneity	Energy	Performance	Platform
[8], [16]	node		yes	yes	yes	both
[9], [10], [12]	node			yes	yes	simulated
[11]	node	yes		yes	yes	both
[14], [18]	node	yes		yes	yes	real
[13], [15]	node			yes	yes	real
[17]	node	yes	yes	yes	yes	simulated
[22]	cluster	yes	yes		yes	both
[23]	cluster	yes	yes		yes	real
[24], [29]	cluster		yes		yes	both
[26], [27], [31]	cluster	yes			yes	real
[28]	cluster		yes	yes	yes	real
[30]	cluster			yes	yes	simulated
[41]	hierarchical	yes	yes		yes	both
[42], [43]	hierarchical		yes		yes	real
Qin	**hierarchical**	**yes**	**yes**	**yes**	**yes**	**both**

In the category of hierarchical approaches, the work in [41] improved the Paragon scheduler by considering heterogeneity at both cluster and server levels and developed the Mage scheduler as a centralized scheduling approach that maps application-to-core directly using stochastic gradient descent (SGD) techniques. Despite the significant contribution to the hierarchical level approaches, [41] did not consider EDP and energy consumption and instead focused on optimizing server utilization. The study in [42] proposed Mesos, a distributed two-level scheduling mechanism for resource sharing in the datacenter. The work in [43] proposed a distributed scheduling approach for hierarchical scheduling. Similarly, [42], [43] improved significantly the resource utilization but again did not consider energy consumption.

Difference and Motivation. Table 1 indicates that (*i*) we have outstanding work at the node and cluster levels for optimizing energy consumption without degrading performance; (*ii*) we probably need more work at the hierarchical level, especially for optimizing energy usage and performance while considering interference and heterogeneity. Therefore, the novelty of the proposed work is the unified hierarchical approach for scheduling that combines cluster and node levels scheduling while modeling interference and heterogeneity and considering performance and energy usage as objectives. We take the application profiling idea from [22], [41] and combine it with our own ideas of D-Choices technique and the collaborative filtering based energy usage prediction and build a unified hierarchical datacenter scheduler based on the previous work. The application area of the statistical methods in datacenter energy consumption optimization should have a significant impact. We see this unified approach as a promising direction in optimization for energy and performance of heterogeneous servers and datacenters.

3 Background

In this section, we describe several concepts that will later be used in developing the proposed scheduling approach.

3.1 Collaborative Filtering

Collaborative filtering uses similarities between users and items simultaneously to provide recommendations. One popular application using collaborative filtering is recommendation systems (e.g., movie recommendation). In the recommendation systems, the input to collaborative filtering is a sparse matrix A - the utility matrix - with one row per user and one column per item. The elements of A are the movie ratings from users. Users rated a subset of the movies, and the collaborative filtering makes movie recommendations based on these ratings. Collaborative filtering uses singular value decomposition (SVD) and PQ-Reconstruction for movie recommendations. SVD is a matrix factorization method used for reducing dimensionality and identifying similarity. Matrix A is decomposed into matrices U, V, and Σ.

$$A_{m,n} = U \cdot \Sigma_{r\times r} \cdot V^T \tag{1}$$

where r is the rank of matrix A, and it represents the similarities identified by SVD. The matrix U represents the correlations between the row and similar concepts, such as to what degree users like fiction movies. Matrix V represents the correlations between the column and similar concepts, such as to what degree the movies fall in the fiction category. Matrix Σ represents similarity concepts.

Then, PQ-reconstruction is used to build matrix R (as an approximation of matrix A), where $R \approx Q \cdot P^T$. The decomposition of A is used to derive matrices P and Q as $P^T_{r\times n} = \Sigma \cdot V^T$, and $Q_{m\times r} = U$. Once matrix R is built, stochastic gradient descent (SGD) [44], [45] is employed to progressively improve the per-element estimations of matrix R via the following equations:

$$\epsilon_{ui} = r_{ui} - q_i \cdot p_u^T \tag{2}$$

$$q_i \leftarrow q_i + \eta(2 \cdot \epsilon_{ui} p_u - \lambda q_i) \tag{3}$$

$$p_u \leftarrow p_u + \eta(2 \cdot \epsilon_{ui} q_i - \lambda p_u) \tag{4}$$

This is done until $|\epsilon|_{L_2} = \sqrt{\Sigma_{u,i}|\epsilon_{ui}|^2}$ becomes marginal. r_{ui} is an element of the reconstructed matrix R, η is the learning rate, and λ is the regularization parameter. The final output matrix R from SGD represents the improved recovered elements that reflect strong similarities for accurate movie recommendations with high confidence.

3.2 Kalman Filtering

Kalman filtering uses recursive equations and a feedback control mechanism to minimize the estimation error variance. We use the following state and output equations [46] to describe Kalman filtering:

$$x_n = Ax_{n-1} + Bu_{n-1} + w_{n-1} \tag{5}$$

$$z_n = Hx_n + v_n \tag{6}$$

where A is the state transition model, B is the optimal control input model, and H relates the state x to the measurement z. w_{n-1} and v_n are the random variables that have a Gaussian distributions.

Kalman filtering includes two phases: the predict phase and the update phase. We use the following equations to describe the predict phase, where the filter uses the previous state $\hat{x}_{n-1}$ and the input u_{n-1} to project the state. It also uses the error covariance of the posterior error P_{n-1} and the process noise covariance Q to project the error covariance P_n^- for the prior error.

$$\hat{x}_n^- = A\hat{x}_{n-1} + Bu_{n-1} \tag{7}$$

$$P_n^- = AP_{n-1}A^T + Q \tag{8}$$

We use the following equations to describe the update phase, where it first computes the Kalman gain K_n to minimize P_n and then updates the current state matrix $\hat{x}_n$ and P_n. R is the measurement noise covariance.

$$K_n = P_n^- H^T(HP_n^- H^T + R)^{-1} \tag{9}$$

$$\hat{x}_n = \hat{x}_n^- + K_n(z_n - H\hat{x}_n^-) \tag{10}$$

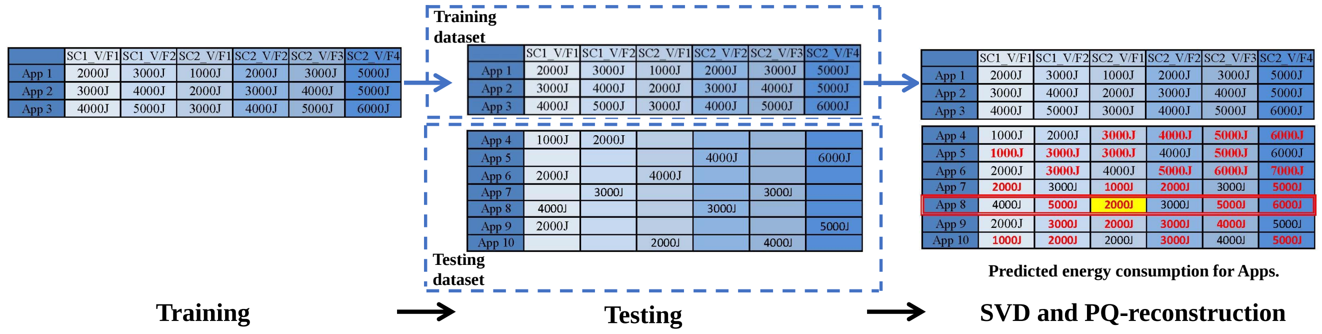
$$P_n = (1 - K_n H)P_n^- \tag{11}$$

4 Prediction Methods

In this section, we present prediction methods that will be employed by the proposed scheduling algorithms at cluster and node levels.

4.1 Collaborative Filtering for Energy Usage Estimation at Cluster Level

To develop energy oriented scheduling algorithms, we use collaborative filtering to predict the energy usage of an incoming application, which will run on different server configurations (SC) with selected Voltage/Frequency (V/F) levels that are set by the Dynamic Voltage and Frequency Scaling (DVFS). We assume that a given server (from a heterogeneous set) can support several different V/F levels. For example, a single server configuration in [22] corresponding to a given server hardware platform is replaced with four different server configurations as the processor can support four different V/F levels. In this way, we bring into the optimization process the DVFS (uses V/F levels) control knob that will help save energy.


Training
Training dataset
Testing dataset
Testing
Predicted energy consumption for Apps.
SVD and PQ-reconstruction

Training

	SC1_V/F1	SC1_V/F2	SC2_V/F1	SC2_V/F2	SC2_V/F3	SC2_V/F4
App 1	2000J	3000J	1000J	2000J	3000J	5000J
App 2	3000J	4000J	2000J	3000J	4000J	5000J
App 3	4000J	5000J	3000J	4000J	5000J	6000J

Testing

	SC1_V/F1	SC1_V/F2	SC2_V/F1	SC2_V/F2	SC2_V/F3	SC2_V/F4
App 1	2000J	3000J	1000J	2000J	3000J	5000J
App 2	3000J	4000J	2000J	3000J	4000J	5000J
App 3	4000J	5000J	3000J	4000J	5000J	6000J
App 4	1000J	2000J				
App 5				4000J		6000J
App 6	2000J		4000J			
App 7		3000J			3000J	
App 8	4000J			3000J		
App 9	2000J					5000J
App 10			2000J		4000J	

SVD and PQ-reconstruction

	SC1_V/F1	SC1_V/F2	SC2_V/F1	SC2_V/F2	SC2_V/F3	SC2_V/F4
App 1	2000J	3000J	1000J	2000J	3000J	5000J
App 2	3000J	4000J	2000J	3000J	4000J	5000J
App 3	4000J	5000J	3000J	4000J	5000J	6000J
App 4	1000J	2000J	**3000J**	**4000J**	**5000J**	**6000J**
App 5	**1000J**	**3000J**	**3000J**	4000J	**5000J**	6000J
App 6	2000J	**3000J**	4000J	**5000J**	**6000J**	**7000J**
App 7	**2000J**	3000J	**1000J**	**2000J**	3000J	**5000J**
App 8	4000J	**5000J**	**2000J**	3000J	**5000J**	**6000J**
App 9	2000J	**3000J**	**2000J**	**3000J**	**4000J**	5000J
App 10	**1000J**	**2000J**	2000J	**3000J**	4000J	**5000J**

Predicted energy consumption for Apps.

Fig. 1: Illustration of how collaborative filtering is employed to predict energy usage of applications executed on server configurations with different V/F settings.

The input to collaborative filtering is a sparse matrix A with one row per application and one column per server configuration with a selected V/F level. The matrix entries represent normalized application energy usage scores. Collaborative filtering requires offline training and online testing. In offline training, we select a small number of applications (around 10%) and profile them on all different server configurations for all V/F levels. We normalize the performance scores and fully populate the corresponding rows of matrix A. The energy usage model that we use is similar to the one in [48]. If a new server configuration is added to the heterogeneous cluster, we need to add columns in matrix A to represent the newly added server for its V/F level configurations and to profile the selected applications in the offline training mode on the newly added server configuration with selected V/F levels.

In the online testing mode, when a new application arrives, we first profile it for a period of 0.05 seconds (fast profiling to avoid memory bursts) on any two server configurations with selected V/F levels. Please note that the 0.05s time is a user defined parameter, which is passed to the *perf* tool for fast profiling. Therefore, the user can change the profiling time (e.g., set it in milliseconds but guarantee collecting enough profiled data) according to different types of workloads. Modern datacenter workloads mainly contain throughput-bound and latency-critical applications. We can set the user-defined application profile time at the millisecond level for the latency-critical applications and at the second level for the throughput-bound applications. We perform new application profiling application-per-core to minimize the intrusion. Then, we insert this application as a new row in matrix A. Lastly, we apply SVD and PQ-reconstruction [47] that are used by the collaborative filtering technique to predict the missing scores of this application for all other server configurations. Because SVD and PQ-Reconstruction are robust methods, missing entries and assumed relaxed sparsity constraints do not significantly affect SVD's accuracy [22].

When new applications come, instead of learning each new application, collaborative filtering leverages profiling data from history and combines a minimal profiling of the new application to identify similarities between new and known applications. Two applications can be similar in one characteristic (e.g., both benefit from a higher level of V/F) but different in others (e.g., one application benefits more from larger memory while the other does not). SVD uncovers the hidden similarities between applications and filters out the ones less likely to have an influence on the application's scores. Fig. 1 illustrates how collaborative filtering is applied to predicting energy usage for an application. In this example, the workload includes 10 applications and 2 server configurations. SC1 has two V/F levels and SC2 has 4 selected V/F levels. In the offline training, we profile App1 to App3 on SC1 and SC2 with the selected V/F levels; the profiling data is then used as training data for the collaborative filter. In the online testing mode, we first profile the testing applications (App4 to App10) on any two server configurations with selected V/F levels (to satisfy SVD's sparsity constraints). Then, we use SVD and PQ-reconstruction to predict the missing entries of energy consumption of applications App4-App10, shown in red color in Fig. 1. In this figure, App8 has the minimum predicted energy usage if scheduled to run on SC2 with V/F level 1.

Energy Usage Calculation: The elements of matrix A represent the estimated energy usage of the applications that run on different server configurations. We consider only the computational energy usage in this paper because it dominates the total energy usage in servers [48]. The following equations are used to calculate power, P, and energy usage, E, for one application that runs on a specific server configuration with a selected V/F level.

$$P = 1/2 \times C \times V^2 \times F \tag{12}$$

$$E = P \times T_{runtime} \tag{13}$$

where C is the total capacitive load, V and F represent the voltage and frequency at which the server is configured, and $T_{runtime}$ denotes the execution time of the application has on the specific server configuration.

4.2 Collaborative Filtering for Interference Estimation at Cluster Level

The work in [22], [49] found that interference in shared resources leads to higher performance loss. Therefore, in order to minimize the performance loss, one needs to minimize the interference in shared resources. Here, we propose to use the collaborative filtering technique from the previous section but to predict interference instead of energy usage scores.

Sources of interference (SoI): In this case, the rows of matrix A represent applications and the columns represent SoI, which include memory capacity, memory bandwidth, storage capacity, storage bandwidth, network bandwidth, last level cache (LLC) capacity, LLC bandwidth, L1 i-cache, L1 d-cache, translation lookhead buffer, integer processing units, floating point processing units, prefetchers, interconnection network, and vector processing units.

The elements of matrix A represent the normalized interference scores of applications against corresponding SoI. There are two types of interference we are interested in: interference that an application can tolerate from the pre-existing load on a server and interference the application will cause on that load. We detect interference due to contention on shared resources and assign a score to the sensitivity of an application to each type of interference. To derive sensitivity scores of applications run on contentious kernels, we use the iBench tool [50].

Interference Calculation: We define $Interf_1$ as the amount of interference that a new workload can tolerate from the interference caused by the pre-existing server load [22]:

$$Interf_1 = t_{newapp} - c_{server} \tag{14}$$

where t_{newapp} denotes the pressure (sensitivity score) the new application can tolerate from the pre-existing server load for a specific SoI; it can be estimated with the iBench tool, which runs a microbenchmark simultaneously with an application and progressively tunes up the microbenchmark's intensity until the application violates its quality-of-service (e.g., 95% of the performance achieved when running alone). c_{server} represents the caused pressure (sensitivity score) by the pre-existing server load; it can be calculated as a cumulative sensitivity score for a server as the sum of the sensitivity scores of individual applications running on the server. We define $Interf_2$ as the amount of interference that the new workload will cause on the pre-existing server load [22]:

$$Interf_2 = t_{server} - c_{newapp} \tag{15}$$

where t_{server} denotes the pressure the pre-existing server can tolerate; it can be calculated as the minimum sensitivity score among the sensitivities of individual applications running on the server. c_{newapp} represents the caused pressure by the new application for a specific SoI and can be estimated with the iBench tool, which runs the application concurrently with a microbenchmark and increases the application's intensity and records when the microbenchmark's performance degrades by 5% (compared to when running alone). Thus, we calculate the interference I of a new application when it runs on a server as the sum of the absolute values (L1 norm) of $Interf_1$ and $Interf_2$ for all SoIs:

$$I = |Interf_1| + |Interf_2| \tag{16}$$

The optimal server candidate is one whose interference is the smallest; ideally zero - which would indicate that there is no negative effect due to interference between the new application and the pre-existing applications running on servers.

Fig. 2 illustrates how collaborative filtering predicts the application's interference score. In the matrix from Fig. 2, each row represents an application, each column represents

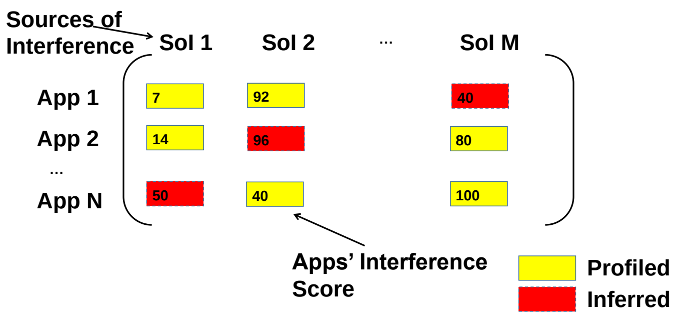


Fig. 2: Illustration of how collaborative filtering is employed to predict interference.

one source of interference (e.g., CPU, cache, network bandwidth, storage bandwidth, etc.), and each element in the matrix represents the application's interference score that is calculated as the sensitivity score estimated with the iBench tool. Collaborative filtering uses profiled applications' interference scores (in yellow) to predict the missing scores (in red). We use the collaborative filtering technique twice to predict the interference that an application can tolerate and the interference an application will cause.

4.3 Collaborative Filtering for Heterogeneity Estimation at Cluster Level

To model heterogeneity, we again use collaborative filtering to identify how well an application runs on different server configurations. Similarly to [22], the input to the collaborative filtering technique is a matrix A whose rows represent applications while columns represent different server configurations. The matrix entries represent normalized application performance scores measured in millions instructions per second (MIPS). In the offline training, we first profile a few tens of selected applications on all the different server configurations to generate the sparse matrix A. In the online testing mode, when a new application arrives, we quickly profile it on any two server configurations, and add it as a new row in the matrix A. Lastly, we apply SVD and PQ-reconstruction to predict the missing performance scores for all other server configurations. In this way, we directly model the heterogeneity of the server configurations at the cluster level.

Heterogeneity Calculation: We define the heterogeneity score as a measure of performance of all possible different types of servers as well as of different configurations (e.g., V/F levels) of a given server. The numerical values to express heterogeneity scores are calculated based on MIPS. We measure MIPS of one application when it runs on one core using the expression:

$$H = \frac{IC}{T_{runtime}} \times 10^6 \tag{17}$$

where IC defines the instruction count of the application during its runtime $T_{runtime}$.

Fig. 3 illustrates how collaborative filtering predicts the application's performance score. In the matrix from Fig. 3, each row represents an application, each column represents a server configuration (different V/F levels), and each element in the matrix represents the application's performance score that is calculated based on MIPS. The collaborative

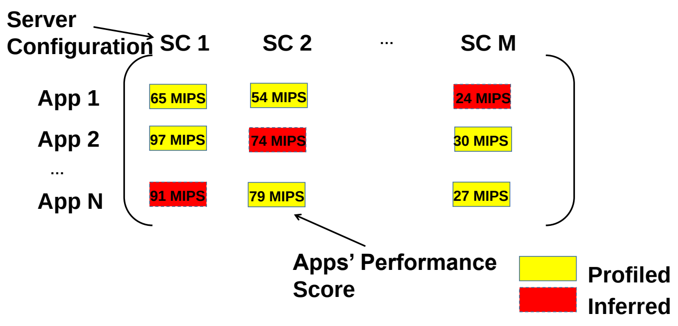


Fig. 3: Illustration of how collaborative filtering is employed to predict heterogeneity.

filtering uses existing profiled applications' performance scores to predict scores of the missing entries.

4.4 Kalman Filtering for Workload Prediction at Node Level

In this work, we combine node-level scheduling with thread migration with DVFS based energy reduction under performance constraints. The goal of the node-level scheduling is to map dynamically thread-to-core together with thread migration and DVFS for energy reduction without performance loss beyond a user set threshold. The actual scheduling algorithm will be described in a later section; threads migration is implemented through rescheduling. The DVFS based energy reduction technique - that is integrated with the scheduling approach - uses Kalman filtering to predict the workload. Similarly to previous literature [9], the workload is measured as average cycles per instruction and instruction count in the next control period for which V/F pairs should be selected and set to reduce energy consumption. This is under the assumption that the execution of a given application is split into consecutive control periods and that the DVFS algorithm is applied at the end of each such period.

Performance Loss Calculation: To estimate performance loss, we adopt the technique proposed in [9]:

$$pl = \sum_{p=1}^{N} \frac{T_{P_{delay}}}{T} \tag{18}$$

where pl denotes the performance loss of one node-level thread-to-core scheduling, p denotes the control period, N denotes the total number of control periods, T stands for the duration of the total length of the control periods, and $T_{P_{delay}}$ represents the extra time caused by the delayed instruction count (the instructions that should but not be executed due to core frequency degradation for energy reduction) during control period p. We calculate $T_{P_{delay}}$ with the following expression from [9]:

$$T_{P_{delay}} = I_{P_{Done}} \times \left(\frac{CPI_P(\frac{f_H}{f_P} - 1)}{f_H}\right) \tag{19}$$

where f_H stands for the highest available CPU frequency while f_P represents the CPU clock frequency in period p, $I_{P_{Done}}$ defines the number of instructions done in period index p, CPI_P represents the average CPU cycles per instruction in period p. $\frac{f_H}{f_P} - 1$ can be considered as slowness in frequency (or extra time in the execution) compared to the execution which would have been done at maximum possible frequency. Eq. 19 provides essentially an estimate of the extra delay in execution that is incurred due to frequency throttling, i.e., lowering the frequency to save energy. The logic behind Eq. 19 is that we are expecting delay based on changing the average CPU frequency while the average system frequency (which contains the stall time, cache misses, pipelining delay, etc.) won't be affected. A detailed explanation of its derivation can be found in [9].

4.5 Putting It All Together

Energy Usage: To calculate the total energy usage of all applications, E_{total}, the following equations are used:

$$E_{total} = \sum_{m=1}^{M} \sum_{i=1}^{N} E(m,i) \cdot x_{m,i} \tag{20}$$

$$E(m,i) = \sum_{j=1}^{S_i} E_{m,j} \cdot y_{m,j} \tag{21}$$

where M represents the total number of applications and N represents the total number of servers. $E(m,i)$ (calculated with eq. 21) defines the total energy usage of application m when it runs on server i. $x_{m,i}$ defines the indicator, which is 1 if application m runs on server i and 0 otherwise (we assume no cross-server migration in this paper). In eq. 21, S_i denotes the total number of server configurations (e.g., V/F levels) for server i. $E_{m,j}$ (calculated with eq. 13) defines energy usage of application m when it runs on server configuration j. $y_{m,j}$ represents the percentage of the execution time (value between $[0,1]$) when application m runs on server configuration j (we allow thread migration on one server in node-level, which will be discussed later).

Interference: To calculate the total interference of all applications running on all servers, I_{total}, the following equations are used:

$$I_{total} = \sum_{m=1}^{M} \sum_{i=1}^{N} I(m,i) \cdot x_{m,i} \tag{22}$$

$$I(m,i) = \sum_{j=1}^{S_i} I_{m,j} \cdot y_{m,j} \tag{23}$$

where M, N, $x_{m,i}$, and $y_{m,j}$ have the same definitions as before. $I(m,i)$ (calculated with eq. 23) defines the total interference of application m when it runs on the corresponding server i. In eq. 23, $I_{m,j}$ (calculated with eq. 16) defines the interference of a new application m when it runs on server configuration j.

Heterogeneity: Lastly, we derive the total heterogeneity score of all applications running on all server configurations, H_{total}, as follows:

$$H_{total} = \sum_{m=1}^{M} \sum_{i=1}^{N} H(m,i) \cdot x_{m,i} \tag{24}$$

$$H(m,i) = \sum_{j=1}^{S_i} H_{m,j} \cdot y_{m,j} \tag{25}$$

where $H(m, i)$ (calculated with eq. 25) defines the total heterogeneity score of application m when it runs on server i. In eq. 25, $H_{m,j}$ (calculated via eq. 17) denotes the heterogeneity score of application m when it runs on server configuration j.

Formulation of Scheduling Problem: The two-level hierarchical cluster-node scheduling problem is formulated as follows:

Given the input applications M and the server configurations S_N

Find the cluster and node levels scheduling functions $S()$ that map *application-to-server* at the cluster-level and *thread-to-core* at the node-level, which:

$$Step1 : min \quad I_{total} \tag{26}$$

$$Step2 : max \quad H_{total} \tag{27}$$

$$Step3 : min \quad E_{total} \tag{28}$$

Such that:

$$\forall s \in S() \qquad pl(s) \leq PL \tag{29}$$

where $pl(s)$ defines the performance loss of hierarchical cluster-node scheduling s, which is calculated with eq. 18. PL represents the performance loss threshold set by the user. I_{total} is calculated with eq. 22, H_{total} is calculated with eq. 24, and E_{total} is calculated with eq. 20.

We minimize interference first because it was observed that interference could lead to higher performance loss than suboptimal server configurations [22]. From among the selected scheduling solutions that have minimum interference scores, we then find those that maximize heterogeneity scores. Lastly, we minimize energy usage scores among those selected in the previous step because our ultimate goal is to reduce energy usage under performance constraints. We discuss application requirements, scalability, profiling overhead, and other aspects in Section 7.4. We assume no cross-server migration (parallel jobs will not be scheduled on multiple classes of servers), but we allow thread migration (implemented through re-scheduling in Sniper) on one server at the node-level (Section 5).

5 Proposed Scheduling Approach

5.1 Overview

To solve the scheduling problem formulated in Section 4.5, we propose Qin: a unified hierarchical cluster-node scheduling approach described in Fig. 4. At the cluster-level, the inputs to the Cluster Manager are the incoming jobs/applications while the output is the application-to-server scheduling. The Cluster Manager has a global view of the scheduling in that it considers different server configurations on the servers, which will be used for node-level scheduling (thread-to-core scheduling based on core V/F levels). Algorithm 1 describes the cluster-level scheduling algorithm. We first fast profile the incoming applications, both offline (around 10% of applications) and online (application-per-core profiling for 0.05 s to minimize the intrusion and avoid memory bursts); this is done by the function *BenchmarkProfiling(m)*. Then, functions *EnergyPrediction(m)*, *InterferencePrediction(m)*, and

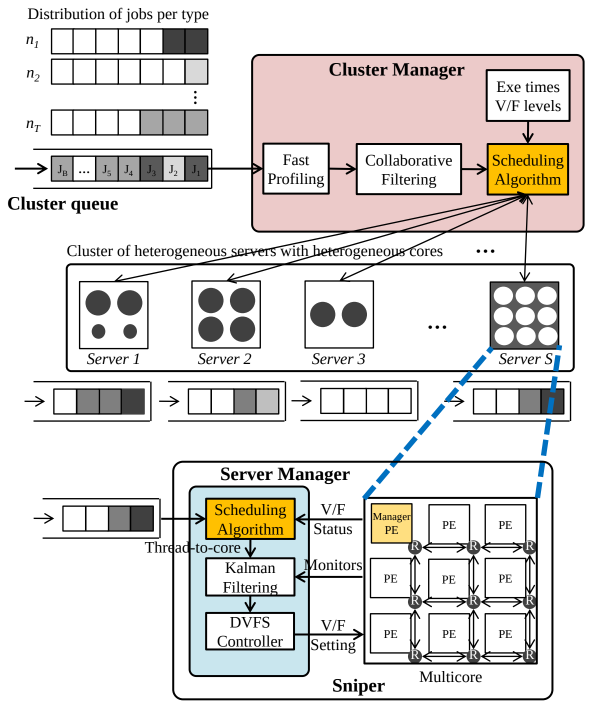


Fig. 4: Illustration of the proposed scheduling approach.

Algorithm 1: Cluster-level scheduling algorithm.

```
1  Inputs: Incoming jobs/applications $M$ to cluster.
2  Outputs: Application-to-server scheduling at
     cluster-level.
3  Function CLUSTER-LEVEL-SCHEDULING()
4      for $m$ in $M$ do
5          BenchmarkProfiling($m$); // Fast online
             profiling
6          E = EnergyPrediction($m$); // Collaborative
             filtering based
7          I = InterferencePrediction($m$); //
             Collaborative filtering based
8          H = HeterogeneityPrediction($m$); //
             Collaborative filtering based
9          D-ChoicesGreedyScheduling($E, I, H$);
10     end
11 end
```

HeterogeneityPrediction(m) use the collaborative filtering prediction technique to estimate the energy usage (Section 4.1), interference (Section 4.2), and heterogeneity (Section 4.3) of new incoming applications. Lastly, energy usage, interference, and heterogeneity estimations are used by *D-Choices Greedy Scheduling Algorithm* (described later) implemented as *D-ChoicesGreedyScheduling*(E, I, H) function, where E stands for the estimated energy usage from the collaborative filtering technique (Section 4.1), I denotes the estimated interference from the collaborative filtering technique (Section 4.2), and H represents the estimated heterogeneity (Section 4.3) from the collaborative filtering technique.

At the node-level, the inputs to the Server Manager are the incoming tasks/threads while the output is thread-to-

Algorithm 2: Node-level scheduling algorithm.

```
1  Inputs: Incoming remain unfinished tasks/threads
     $T$ to nodes.
2  Outputs: Thread-to-core scheduling and DVFS at
     node-level.
3  Function NODE-LEVEL-SCHEDULING()
4    for Every time_interval do
5      // For every default power update time
         interval 1ms
6      for $t$ in $T$ do
7        // For every remain unfinished thread $t$,
           do nothing if no thread remains
8        CollectCoreV/FStatus(); // Fast core V/F
           collection
9        E = EnergyPrediction($t$); // Same as
           cluster-level
10       I = InterferencePrediction($t$); //
           Calculated as extra delay
11       H = HeterogeneityPrediction($t$); //
           Calculated as instruction count
12       D-ChoicesGreedyScheduling($E, I, H$);
13     end
14     PerCoreWorkloadPrediction(); // Kalman
         filtering based
15     DVFS(); // Set V/F levels for each core
16   end
17 end
```

core scheduling with task migration and DVFS control. The Server Manager combines node-level scheduling and thread migration with DVFS to achieve dynamic energy reduction under performance constraints. Algorithm 2 describes the node-level scheduling algorithm. The function *CollectCoreV/FStatus()* collects core V/F levels information. Similarly to the cluster-level case, functions *EnergyPrediction(t)*, *InterferencePrediction(t)*, and *HeterogeneityPrediction(t)* leverage the collaborative filtering based prediction to estimate energy usage, interference (calculated as extra delay the thread causes to the mapped core), and heterogeneity (calculated as instruction count) of new incoming threads. Lastly, these estimations are fed into *D-ChoicesGreedyScheduling(E, I, H)*. The *DVFS()* function is implemented by the DVFS Controller from Fig. 4. It implements the DVFS scheme reported in [9], but, adapted to support thread migration; it is responsible for finding optimal V/F pairs for all cores inside the multicore processor on the server node. It uses as input Instruction Count and CPI values as predicted by the Kalman filter for the next control period implemented by PerCoreWorkloadPrediction() function.

Again, the main novelty of the proposed scheduling approach is the unified hierarchical cluster-node scheduling considering interference and heterogeneity for multi-objective optimizations in heterogeneous datacenters. By saying "unified", we mean that (*i*) the cluster-level and node-level scheduling use the same D-choices greedy scheduling method, and (*ii*) the cluster-level scheduling has a holistic view and interacts with the node-level scheduling. For example, the DVFS during node-level scheduling can affect (through fast profiling results) the cluster-level scheduling while the cluster-level scheduling can enhance (through allocating applications to servers that have available lower levels of V/F) the energy optimization of the node-level scheduling.

5.2 D-Choices Greedy Scheduling

The D-Choices concept applied to the problem of hierarchical scheduling is one of the main novel contributions in this paper. The key idea is that a small number of choices in greedy scheduling can lead to significantly better load balancing. This is in contrast with previous work where the single choice greedy scheduling approach allocates many more jobs to high-performance servers than regular servers, which results in load imbalance among servers that undermines the overall jobs completion time. On the other hand, we found that trading-off load balancing and local-optimal greedy scheduling (resulting in load imbalance) can improve overall job completion time and energy usage.

We define and frame the application-to-server problem and the thread-to-core problem through the so-called ball-to-bin model: allocating m balls into n *heterogeneous* bins (assuming $m \geq n \ln n$) [52]. In this model, when we use a *single choice* approach, the upper bound of the maximum load of bins is $m/n + O(\sqrt{(m \ln n)/n})$ with high probability. However, when we use a *multiple choices* paradigm, the maximum load of bins is $m/n + O(\ln \ln n) + O(1)$ with high probability. When applying this model of "balls into heterogeneous bins" [52] to the problem of scheduling on heterogeneous servers, we must consider the following differences: (*i*) "Balls" represent various applications at cluster-level and various threads at node-level. (*ii*) The objective is to minimize *imbalance*, which translates into minimization of applications/threads queuing time, thereby improving performance.

To address these differences, we first define the load at both cluster and node levels. At the cluster-level, at time t, the load of a server i is the total size (total instructions count) of the unprocessed applications assigned to the server up to t is denoted as $L_i(t)$. At the node-level, at time t, the load of a core j is the total size (total instructions count) of the unfinished threads assigned to the core up to t is denoted as $L_j(t)$. We then define the *imbalance* at both cluster-level and node-level. At time t, the imbalance is the difference between the *maximum* and the *average* load of the servers (calculated with eq. 30) at the cluster-level (denoted as $Im_c(t)$), and of the cores (calculated with eq. 31) at the node-level (denoted as $Im_n(t)$).

$$Im_c(t) = max\{L_i(t) | i \in 1, 2, ..., N\} - \sum_{i=1}^{N} L_i(t)/N \quad (30)$$

$$Im_n(t) = max\{L_j(t) | j \in 1, 2, ..., S_N\} - \sum_{j=1}^{S_N} L_j(t)/S_N \quad (31)$$

where i and j are the indices for servers and server configurations (core V/F levels), while N and S_N represent the numbers of servers and server configurations.

To this end, the application-to-server scheduling problem is defined as: Allocate M applications to N heterogeneous servers, where each application m has the size of α_m, to minimize cluster-level imbalance $Im_c(t)$. Also, the thread-to-core problem is defined as: Allocate S_M various threads to S_N heterogeneous server configurations (core V/F levels), where each thread k has the size of β_k, to minimize node-level imbalance $Im_n(t)$. To solve these problems, the D-Choices Greedy Scheduling method is proposed. Compared to traditional approaches where only one optimal candidate server or server configuration is selected, here, we select the *top* D ($D \geq 2$) candidates and then randomly pick one from among these D candidates.

Approximate Upper Bound: At the cluster-level, assuming $M \geq N \ln N$, when allocating M applications to N heterogeneous servers using D-Choices Greedy Scheduling method, we approximate the upper bound of the D-Choices Greedy Scheduling as follows. More details of the approximation is shown in APPENDIX A.

Thus, at the cluster-level, when allocating M applications to N heterogeneous servers using D-Choices Greedy Scheduling method, the approximate upper bound of the imbalance $Im_c(t)$ satisfies, with high probability:

$$Im_c(t) = \begin{cases} O(\sqrt{(\sum_{m=1}^{M} \alpha_m \ln N)/N}), & if \quad D = 1 \\ O(\ln \ln N) + O(1), & if \quad D \geq 2 \end{cases} \tag{32}$$

Similarly, at the node-level, when allocating S_M threads to S_N heterogeneous server configurations using D-Choices Greedy Scheduling method, the approximated upper bound is as follows. More details of the approximation is shown in APPENDIX A.

$$Im_n(t) = \begin{cases} O(\sqrt{(\sum_{k=1}^{S_M} \beta_k \ln S_N)/S_N}), & if \quad D = 1 \\ O(\ln \ln S_N) + O(1), & if \quad D \geq 2 \end{cases} \tag{33}$$

At the cluster level, if we assume each incoming application has the same application size, where α_m is a constant. Under this assumption, the approximate upper bound of the single choice ($D = 1$) $O(\sqrt{(\sum_{m=1}^{M} \alpha_m \ln N)/N})$ is at the same order as $O(\sqrt{(M \ln N)/N})$. Where the approximate upper bound of the D-Choices ($D \geq 2$) remains $O(\ln \ln N)$. Similar approximation can also happen at the node level. We also plot the relationship between the approximate upper bound of the imbalance $Im_c(t)$ at the cluster-level and the number of heterogeneous servers N in APPENDIX A. Thus, we conclude that: At the cluster-level, when the number of applications M is large (which is realistic in todays datacenters such as those of Google that process millions of applications per day) and the size of each application α_m is large (which is also realistic, especially for deep learning jobs), the proposed D-Choices Greedy Scheduling method leads to significant improvement over traditional cluster-level best choice scheduling methods - which is the case of previous schedulers Paragon [22], Mage [41], and Kubernetes [67] schedulers - in load balancing. At the node-level, when the number of threads S_M and the size of each thread β_k are large, the proposed D-Choices Greedy Scheduling method achieves significant improvement over traditional node-level best choice scheduling methods - which is the case of Sniper [68] scheduler - in load balancing.

Algorithm 3: D-choice greedy scheduling algorithm, cluster level, focus: performance.

```
1  Inputs: Estimated energy, interference, heterogeneity
     scores E, I, H.
2  Outputs: Selected server for the new incoming
     application.
3  Function
     D-CHOICES-GREEDY-SCHEDULING(E,I,H)
4      Sorted_I = Sort_Interference(I);
5      P =
         SelectTop_D_InterferenceCandidates(Sorted_I);
6      Flist.AddCandidates(P);
7      Subset_I = SelectSubsetServers(Sorted_I); //
         Select top subset server candidates in
         interference scores
8      Sorted_H = Sort_Heterogeneity(H[Subset_I]);
9      Q = Select-
         Top_D_HeterogeneousCandidates(Sorted_H);
         // Select top D server candidates in
         heterogeneity scores from the subset
10     Flist.AddCandidates(Q);
11     RandomlySelectOne(Flist);
12 end
```

Algorithm 4: D-choice greedy scheduling algorithm, node level, focus: energy usage.

```
1  Inputs: Estimated Energy, interference,
     heterogeneity scores E, I, H.
2  Outputs: Selected core for the incoming thread.
3  Function
     D-CHOICES-GREEDY-SCHEDULING(E,I,H)
4      CollectCoreV/FStatus();
5      Sorted_I = Sort_Interference(I);
6      P =
         SelectTop_D_InterferenceCandidates(Sorted_I);
7      Flist.AddCandidates(P);
8      Subset_I = SelectSubsetCores(Sorted_I); //Select
         top subset core candidates in interference scores
9      Sorted_E = Sort_Energy(E[Subset_I]);
10     Q = SelectTop_D_EnergyCandidates(Sorted_E);
         //Select top D core candidates in energy scores
         from the subset
11     Flist.AddCandidates(Q);
12     RandomlySelectOne(Flist);
13 end
```

As described earlier in Algorithms 1 and 2, we use the proposed D-Choices Greedy Scheduling method into the cluster-node scheduling approach described in Fig. 4. More specific details of how this method is implemented are presented in Algorithms 3 and 4. Algorithm 3 shows the pseudocode description for the specific implementation where the focus of the optimization is performance (i.e., completion time). In this case, the inputs to *D-CHOICES-GREEDY-SCHEDULING(E,I,H)* include estimated energy usage, interference, and heterogeneity scores. We do not use energy scores E when scheduling focuses on performance. The algorithm sorts and selects first the top $D1$ server configurations - for any given server - in terms of interference and adds them to the finalist server candidates. This way, interference is minimized first because interference usually results in higher performance loss than suboptimal server configurations. Then, we select the top P percent of server configurations according to interference scores as a subset. It is from among this subset of server configurations, that finally the top $D2$ server configurations with the best heterogeneity scores are selected and added to the overall finalist candidates list; and finally, one of them then is randomly picked from the list as the server to which the incoming application is to be scheduled. Algorithm 4 shows the pseudocode description for the D-Choices Greedy Scheduling method at node-level where the objective focus is energy usage minimization. Its idea is similar that of the cluster-level case. The difference is that in this case we work with voltage and core frequency scores instead. The specific versions of this algorithm when the objective focus is performance or EDP are similar - not included here in the interest of space; the difference is in the types of scores that are employed: interference and frequency for performance; interference, frequency and voltage for EDP.

5.3 Setting d for D-Choices

How many choices are needed for the D-Choices Greedy Scheduling described in Algorithms 3, 4? Are two choices ($d = 2$) enough? One could heuristically set d by starting from $d = 2$ and increase it until the best load balancing is achieved. But this method is time-consuming and not practical. Instead, we select d for D-Choices based on the analysis presented earlier, when the balls-into-bins model was discussed - where we allocated m balls into n heterogeneous bins under the assumption that $m \geq n \ln n$. According to that model, one can achieve a constant load balance as long as $d = \Omega(\log n)$ [53], [54], where n is the number of bins. Therefore, we set $d = \lceil \log N \rceil$, where N is the number of heterogeneous servers, as the trade-off between the setting complexity, the load balancing, and the memory overhead (aggregation cost). For example, in our case where $N = 6$ (we have six heterogeneous computers in the cluster, as it will be discussed later on), we set $d = 2$ (two choices) for the D-Choices greedy scheduling algorithms.

6 Dynamic Admission Control

Major cloud service companies deploy admission control protocols to shorten applications queue time, prevent machine overload, enhance resiliency, and enable authentication and authorization [60]–[63]. A traditional fixed admission control protocol ignores that often humans are the users who submit jobs to the datacenter. For example, Raj high-performance computer cluster (at Marquette University) [64] has many submitted jobs during the school year (job arrival rate increases even more at the end of each semester), but sparsely arriving jobs during the summer period. Similarly, one can see an increase in online shopping behavior before holidays. This type of job arrival behavior motivates the proposal of dynamic admission control (DAC) protocol that considers "user behavior" (jobs arrival rates)

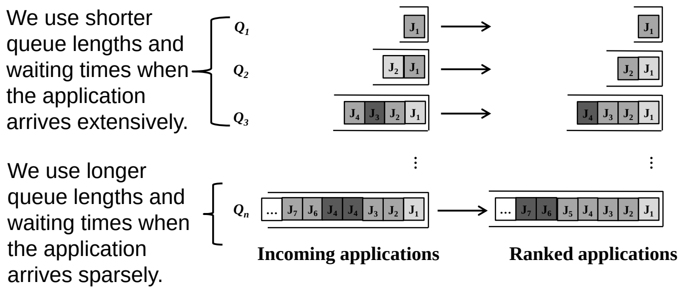

We use shorter queue lengths and waiting times when the application arrives extensively.
Q_1
Q_2
Q_3
J_1
J_2
J_3
J_4
We use longer queue lengths and waiting times when the application arrives sparsely.
Q_n
J_5
J_6
J_7
Incoming applications
Ranked applications

Fig. 5: Illustration of dynamic admission control (DAC). DAC adjusts applications queue length and waiting time dynamically to achieve the best trade-off between applications' waiting times and their scheduling results.

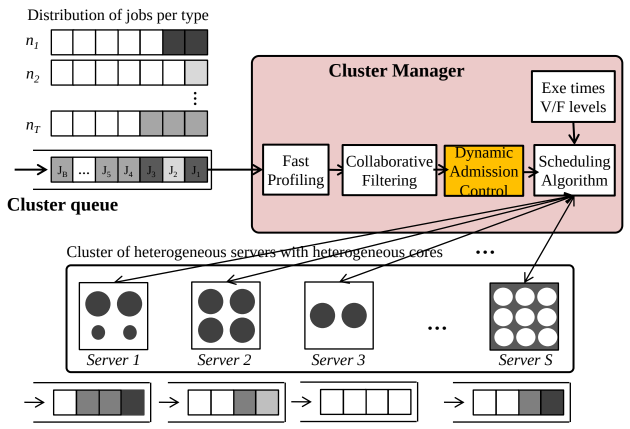


Fig. 6: Dynamic admission control is integrated in the cluster-level scheduling; this figure is a changed version of Fig. 4.

in a way that shortens applications execution time or energy consumption.

The key novelty of DAC is to achieve the best trade-off between applications queue length (applications' waiting time) and their scheduling results (scheduled to a preferred/suitable server) based on the observation that different incoming application orderings result in different application queue times, jobs completion times, and energy usages. For example, Fig. 5 illustrates that when application arrives extensively (e.g., such as the jobs submitted to Raj high-performance computer cluster during the academic year), DAC uses shorter queue lengths and waiting times - at the cost that not all the applications in the queue can be scheduled to their best corresponding suitable servers. On the other hand, when the applications arrive sparsely (e.g., jobs submitted during summer time), DAC uses longer queue lengths and waiting times - in this way ensuring that applications in the queue have a higher chance of being scheduled to a preferred/suitable server.

The proposed dynamic admission control is integrated inside the cluster-level scheduling approach as shown in Fig. 6. Effectively, it is implemented after the profiling and prediction steps (described in Section 4), but, before the D-Choices Greedy Scheduling process (described in Section 5.2). In our experiments, we observed that given the same workloads scheduled by the same scheduler to run on the same cluster, different incoming application orderings result in different application queue times, which impact the total jobs completion time. We use this observation to design a DAC strategy. However, in a practical deployment, we need to consider the following challenges: (*i*) We have infinite continuous incoming applications to the datacenter; in other words, we know the index of the first application but do not the last one, and (*ii*) We want to benefit more from DAC than pay its overheads; in other words, we want DAC's overhead to be constrained.

To address the challenge of the infinite workload, we use the power of the "double" concept. After completing the fast online profiling and prediction steps, we get the predicted energy usage, interference, and heterogeneity scores for each application running on all server configurations. DAC ranks applications according to their interference, heterogeneity, and energy usage scores. For example, inter-server data movement is projected to double every 12-15 months and thus increases its interference to the servers and applications that running on the servers. The interference scores predicted and profiled by the iBench [50] considered compute (e.g., CPU), memory (e.g., cache), and network (e.g., network bandwidth) in a holistic way. Thus, ranking applications according to their interference scores potentially mitigates the interference caused by data movement across servers. Similarly, future datacenter become heterogeneous due to server upgrading, and thus ranking applications according to their heterogeneity scores mitigates the performance loss caused by datacenter heterogeneity. Ranking applications according to their energy usage scores can also reduce the application's total energy usage. For example, when ranking using applications' average heterogeneity scores, we rank the applications from index low to index $high$ (initially $low = high = 1$) in descending order according to their average heterogeneity scores (the higher the heterogeneity score, the lower the application queue time it brings). After completing each ranking, we set index $high = high{*}2$ and re-rank all the applications between low and $high$. We repeat this "double-and-rank" process until the overhead threshold is reached. After that, we feed these ranked applications into the D-Choices Greedy Scheduling process described in Section 5.2 and repeat the "double-and-rank" process for the continuously arriving new applications. The "double-and-rank" process takes *logarithmic* iterations to find the maximum length of the ranked applications queue to feed the D-Choices Greedy Scheduling while guaranteeing that DAC's overhead is constrained.

To address DAC's overhead challenge - the trade-off between applications queue length (applications' waiting time) and their scheduling results, we set the threshold of the DAC's overhead σ_{DAC} to be associated with the jobs arrival rate υ_{jobs}. If the users submit jobs to the datacenter extensively (υ_{jobs} is large), we set a tight threshold (a small σ_{DAC} translates into spending less time in ranking applications). Otherwise, we set a relaxed threshold.

Discussion: What is the overhead of implementing DAC inside the Qin Scheduler? DAC's overhead comes mainly from the "double-and-rank" process, which needs *logarithmic* iterations to find the right applications queue length to rank the applications while guaranteeing applications' ranking times are within the threshold (set by the user). In other words, when we have infinite incoming applications, the threshold constrains DAC's overhead. How much optimization does the DAC bring to the Qin Scheduler? We propose DAC based on the observation that different incoming application orderings result in different application queue times, job completion times, and energy usage. Our experimental results from 7.1 demonstrate that DAC improves Qin Scheduler in performance and energy usage. How to adjust application queue length and waiting time based on workload arrival rate? We need to consider the applications' types in the workload and the free-server probability distributions in the datacenter. First, inspired by the work in [65], we assume the job arrival follows the Poisson distribution with parameter λ. If the incoming workload is interactive applications (which are the cases in [11]–[14], [18]), then the threshold of the DAC's overhead σ_{DAC} should be within 99th percentile (μsec) of the tail latency of the interactive application; if the incoming workload is throughput-bound applications (e.g., Parsec 3.0 benchmarks [71]), then we can set the threshold of the DAC's overhead σ_{DAC} to be $1/\lambda$. Second, we also need to consider the free-server probability distributions in the datacenter. For example, the work in [66] statistically analyzed the free-server probability distributions in a 1000 server experiment on Amazon EC2, and they found that within 56 sec when the application arrived at the queue, there was a 60% probability that there would be at least one free server for this arrived application.

7 Experiments and Simulations

Please recall that this paper focuses on how we can reduce energy usage without degrading performance in heterogeneous datacenters via scheduling. Section 5 aims to answer this question by proposing unified hierarchical scheduling using a D-Choices technique. In this section, we implemented the proposed scheduling method on a real cluster to demonstrate the proposed unified approach as a promising direction in optimizing the energy and performance of heterogeneous servers and datacenters.

Real Cluster for Experiments: We implemented the cluster-level scheduling algorithm as a "plug-in" custom scheduler managed by the Kubernetes platform on a real in-house cluster built with six heterogeneous computers. Our complete implementation will be made publicly available on a GitHub repository, which will also include a video demonstration. The specific characteristics of the six computers are listed in Table 2. We implemented the cluster with Kubernetes v1.14.0 and virtual networking layer *flannel*. We used the Linux *perf* tool v5.4.148 for fast profiling of instructions count and CPU and memory energy usage.

TABLE 2: Characteristics of nodes in the Kubernetes cluster.

Server Type	Role	GHz	Cores	L1(KB)	Mem(GB)
Xeon E5-1620	master	3.60	8	32	16
Intel i5-6600	worker	3.30	4	32	16
Intel i7-4790	worker	3.60	8	32	16
Intel i5-7600	worker	3.50	4	32	8
Intel i5-4690	worker	3.50	4	32	8
Intel i5-4670	worker	3.40	4	32	8

Schedulers: At the cluster-level, we compare the proposed Qin scheduler against the Kubernetes scheduler [67] (widely deployed on Amazon AWS, Google Cloud Platform, Microsoft Azure and IBM Cloud) and against Paragon [22] and Mage [41] schedulers (tested on major cloud computing services). At node-level, we compare the proposed Qin scheduler with the Sniper scheduler inside Sniper simulator v7.2 [68]. At the combined cluster-server levels, we compare the proposed unified hierarchical Qin scheduler versus the combination of Kubernetes scheduler at the cluster-level and Sniper scheduler at the node-level. Table 3 summarizes the compared schedulers in this paper.

TABLE 3: Summary of the compared schedulers.

Scheduler	Method	Metrics
Kubernetes	Best Node	Multiple (resource, constraints, ...)
Paragon	Greedy and Statistical	Server utilization and Interference
Mage	Stochastic Gradient Descent	Server utilization and Performance
Sniper	Least Loaded	Performance, Energy, EDP
Qin	D-choices Greedy	Performance, Energy, EDP

Workloads: We conduct evaluations using both real-world and synthetic application workloads. Similarly to the study in [22], we use Splash-2 benchmarks [69] as real-world applications, randomly replicated with equal likelihood and randomized interleaving to generate up to 100 real-world application workloads. Similarly to the study in [41], we use the mutilate load generator [70] to generate synthetic latency-critical workloads, and again up to 100 synthetic application workloads with uniform, normal, and exponential distributions. In addition, to study modern datacenter workloads, which contain throughput-bound and latency-critical applications, we use Parsec 3.0 benchmarks as well [71]. In our future work, we will test the fast indirect profiling on abstract parameters for energy usage with more computation-heavy applications.

7.1 Experiments at Cluster Level

7.1.1 *Qin Scheduler Outperforms State-of-the-art When Optimizing Solely on Performance, Energy, and EDP*

Comparison: To avoid scheduling overheads, we implement each scheduler with a different objective as an independent custom scheduler managed by the Kubernetes platform. Then, these schedulers could be switched between as necessary. If workloads submitted to the cluster are mainly latency-critical, the performance aware Qin scheduler can be used to focus on jobs completion time. If workloads are energy-hungry, the energy aware Qin scheduler can be used; otherwise, the EDP aware Qin scheduler is used. For all cluster-level versions of the Qin scheduler, we use $d = 2$ inside the D-choice greedy scheduling algorithm - as it was explained earlier. A summary of the comparison of the Qin scheduler against the other schedulers is presented in Table 4 for three types of applications: 100 real-world workloads, 100 latency-critical synthetic workloads, and 100 throughput-bound workloads.

Performance: To compare the performance aware Qin scheduler with Kubernetes, Paragon, Mage schedulers regarding performance, we measure the normalized jobs completion time of these schedulers with the increased number of 100 real-world application workloads on 6-server heterogeneous cluster (Fig. 7.a). In Fig. 7.a, the x-axis represents the number of workloads for which scheduling is done and the y-axis shows the performance measured as normalized jobs completion time. Each line corresponds to the performance gained by different schedulers. We found that the proposed performance aware Qin scheduler outperforms all other schedulers on average by around 9% (Kubernetes), 26% (Mage), and 32% (Paragon) respectively. Furthermore, the improvement in jobs completion time gets even better as the number of workloads increases, which demonstrates a good scalability of the proposed scheduler. Thus, we conclude that the performance aware Qin scheduler - integrated with D-Choices Greedy scheduling - results in better load balancing and reduces the normalized jobs completion time.

Energy usage: Similarly, to compare the proposed energy aware Qin scheduler with Kubernetes, Paragon, Mage schedulers regarding energy, we measure the normalized energy usage (of both CPU + memory combined) with the increased number of 100 real-world application workloads on 6-server heterogeneous cluster (Fig. 7.b). We found that, again, when optimizing solely on energy usage, the proposed scheduler outperforms the state-of-the-art schedulers, because it directly considers the energy usage through the collaborative filtering based energy usage estimation.

Performance-energy tradeoff: Lastly, to compare the proposed EDP aware Qin scheduler outperforms the other schedulers regarding energy efficiency, we measure normalized energy delay product (EDP) with the increased number of 100 real-world application workloads on 6-server heterogeneous cluster (Fig. 7.c). We found that, again, the proposed EDP aware Qin scheduler outperforms the other schedulers on this dimension too.

7.1.2 *Qin Scheduler Increases Server Utilization But Remains Unvaried For Memory Utilization*

Server Utilization: To test if the Qin scheduler increases server utilization, we generated the heat maps of the server utilization - calculated as average CPU utilization and collected by *Metrics API*) - with time for Qin and Kubernetes schedulers for 100 synthetic application workloads on 5-worker + 1-master cluster (Fig. 8). Fig. 8.a indicates that Qin scheduler achieves high and balanced servers utilization during the jobs completion time, while Fig. 8.b indicates that Kubernetes scheduler achieves good servers utilization in the middle period but shows long tail imbalanced servers utilization during the ending period (Paragon and Mage schedulers have similar long tail phenomenon). The long tail phenomenon increases the delay in jobs completion time and is caused by the best choice scheduling methods (used by Kubernetes, Paragon, and Mage) that lead to load imbalance among servers. Thus, we conclude that the D-Choice Greedy scheduling method - used by Qin - results in better load balancing among servers, avoids this long tail phenomenon, and thus increases server utilization.

Memory Utilization: To determine if the Qin scheduler increases memory utilization, we generate the heat maps of the average memory utilization vs. time for Qin and Kubernetes schedulers for 100 synthetic application workloads on 5-worker + 1-master cluster (Fig. 9). We observed (*i*) unvaried memory utilization during scheduling and (*ii*) imbalanced memory distribution among servers for both schedulers. The memory utilization is unvaried because both Qin and Kubernetes schedulers leave the memory the way it is during scheduling. The surprising imbalanced memory distribution happens because of different memory sizes and allocated jobs in each server. Thus, we found that

TABLE 4: Improvement achieved by Qin scheduler over three state-of-the-art schedulers at cluster-level.

	Real-world apps			Latency-critical apps			Throughput-bound apps		
Cluster-level Scheduler	**Performance**	**Energy**	**EDP**	**Performance**	**Energy**	**EDP**	**Performance**	**Energy**	**EDP**
Kubernetes	9.43%	21.23%	43.53%	33.52%	4.01%	26.36%	18.31%	4.70%	22.16%
Paragon	32.39%	25.69%	60.23%	37.57%	11.8%	36.46%	30.95%	4.71%	34.20%
Mage	26.15%	16.85%	51.39%	22.29%	3.51%	21.24%	19.44%	3.98%	22.65%

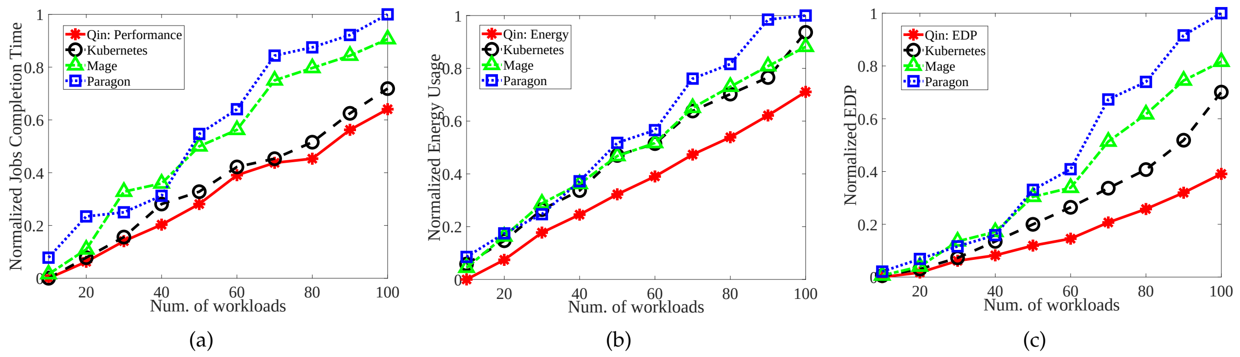
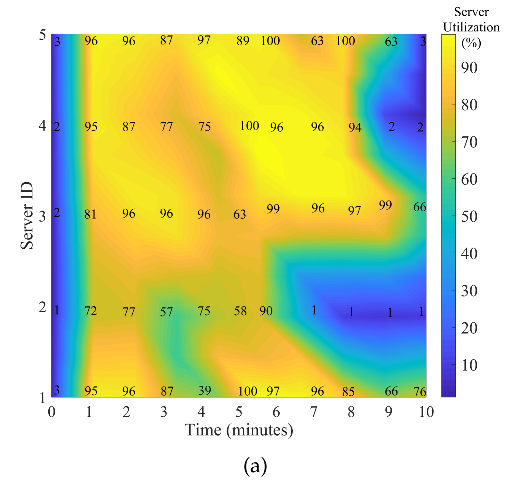


Fig. 7: Comparison of the proposed Qin scheduler against state-of-the-art schedulers at cluster level: a) jobs completion time, b) energy usage, and c) EDP.



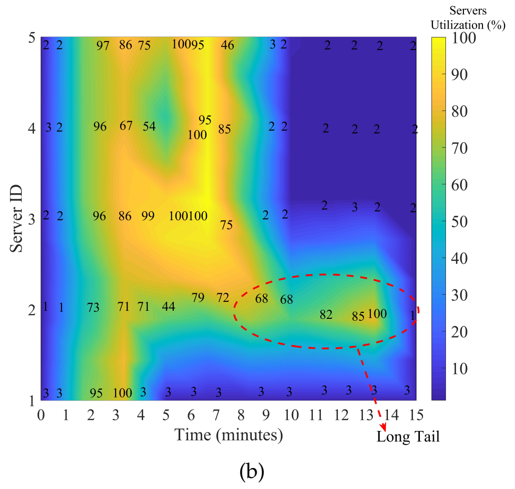


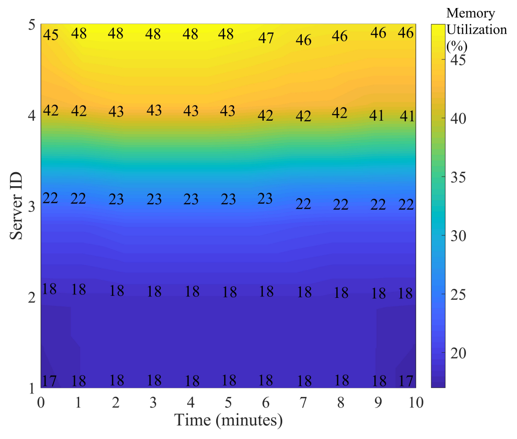
Fig. 8: Server utilization vs. time: (a) Qin scheduler and (b) Kubernetes schedulers for 100 synthetic application workloads at the cluster level.

the Qin scheduler does not improve memory utilization, and there is room for improvement in scheduling by exploiting factors that affect the memory system.

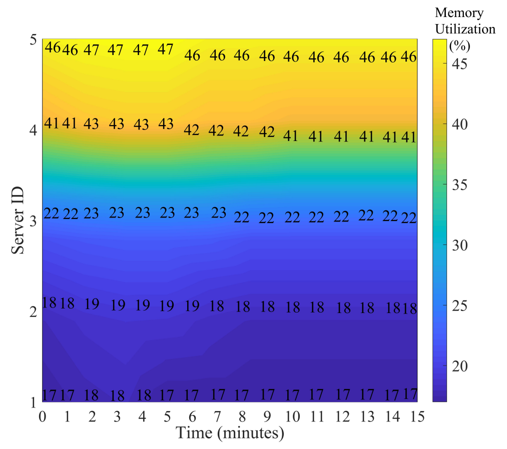
7.1.3 Qin Scheduler Has Low Scheduling Overheads, Good Scalability, and Good Parameter Setting For D-Choices

Scheduling Overheads: Fig. 10.a shows the execution time breakdown of the Qin scheduler for 100 synthetic application workloads on the 6-server heterogeneous cluster. We find that the fast profiling and classifying step and the D-Choices Greedy scheduling step are around 2% and 1%, respectively. We conclude that the Qin scheduler provides a good trade-off in performance, energy usage, and energy-delay-product despite acceptable overheads. When comparing the Qin scheduler with the state-of-the-art schedulers, as shown in Table 3, Qin, Mage, and Paragon may have similar scheduling overheads because of the used profiling-based prediction methods. Kubernetes' default schedule has the pros of supporting response time in milliseconds for production usage, but it has the cons of mainly optimizing performance and resources without considering energy usage, interference, and heterogeneity.

Scalability: We test scalability in scale-out and scale-up categories with six heterogeneous computers. Scale-out (horizontal) test focuses on the performance of Qin Scheduler when the number of application workloads increases; scale-up (vertical) test explores Qin Scheduler's performance when the size (execution time) of applications increases. Fig. 11.a shows the normalized jobs completion time of Qin Scheduler when the number of synthetic application workloads scale-out to 200. Fig. 11.b shows Qin Scheduler's performance when the execution time of each synthetic application doubled. We conclude from Fig. 11 that Qin Scheduler scales well in both cases. Cluster scale-out dominated performance gains by improving around 300x in computing recently [72]. In our future work, we will (*i*) use

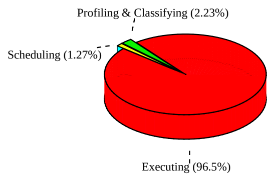


(a)

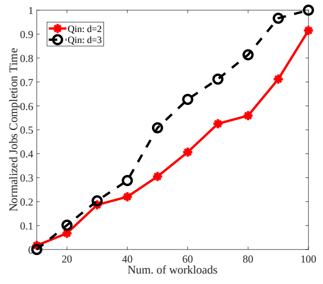


(b)

Fig. 9: Memory utilization vs. time: (a) Qin scheduler and (b) Kubernetes schedulers for 100 synthetic application workloads at the cluster level.



(a)

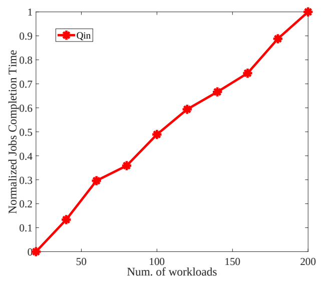


(b)

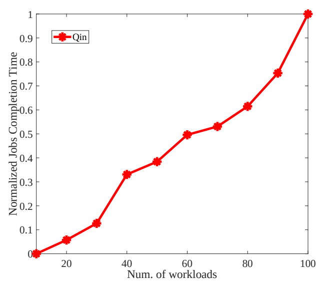
Fig. 10: (a) Execution time breakdown for Qin scheduler for 100 synthetic application workloads at the cluster level. (b) Comparison of d values for D-Choices Greedy scheduling method on real-world application workloads.

the simulator to simulate the situation when the number of servers increases and (*ii*) deploy the Qin scheduler on a concrete large-scale data center when everything is all set.

Setting d for D-Choices: Fig. 10.b evaluates d for D-Choices Greedy scheduling method using 100 real-world application workloads. We observe that (*i*) increasing d does not improve the load balancing and the jobs completion time because applications differ in size and execution time and (*ii*) setting d as the minimum optimal value (i.e., 2 in our case) results in good load balancing and jobs completion time.



(a)



(b)

Fig. 11: Scalability tests: (a) scale-out (horizontal) test (b) scale-up (vertical) test at the cluster-level.

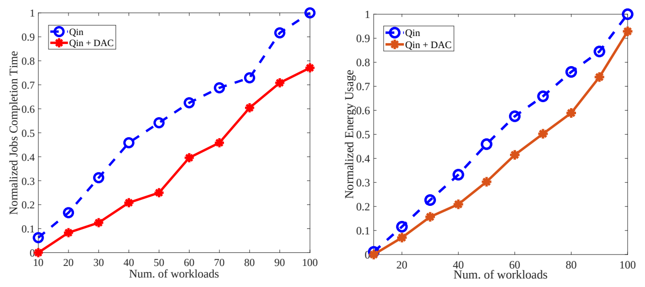
7.1.4 Dynamic Admission Control Reduces Total Jobs Completion Time And Energy Usage

Dynamic Admission Control: We investigate the impact of implementing the proposed dynamic admission control (DAC). Fig. 12.a (ranks applications according to descending average heterogeneity scores) and Fig. 12.b (ranks applications according to ascending average energy usage scores) show the comparison of normalized jobs completion time and energy usage for Qin scheduler with and without the DAC protocol for 100 real-world application workloads on the 6-server heterogeneous cluster. We set the application queue length to 100 and the application waiting time to 0. We do not set the Poisson distribution with parameter λ because 100 applications are relatively small, and we pass all the input applications at once to the Qin Scheduler on the Kubernetes platform. We do not set a threshold σ_{DAC} in this experiment because DAC's overhead is small when ranking all applications. We observe that the Qin scheduler with DAC improves jobs completion time and energy usage over the one without on average by around 21% and 6%.

Such improvement is expected because the DAC protocol shortens application queue time, thereby reducing the total jobs completion time and energy usage. DAC aims to achieve the best trade-off between applications' waiting time and their scheduling results (scheduled to a preferred/suitable server). For example, DAC ranks application that has a higher heterogeneity score (higher MIPS) with a lower index (to be scheduled earlier). Because scheduling takes time, the lower index application may have been executed already before the next application is scheduled, and thus, DAC shortens the application waiting time. DAC reduces energy usage because it ranks energy-efficient application with a lower index, and thus reduce the waiting time of the energy-hungry applications, which turns into extra saved energy usage.

7.1.5 Visualizing Qin Scheduler in Realtime

One More Thing: Visualizing Schedulers. We use Prometheus tool [73] to generate the data plotted in Fig. 13. This tool allows us to visualize and compare Kubernetes and Qin Schedulers in realtime when running 100 synthetic application workloads separately on 5-worker + 1-master



(a) (b)

Fig. 12: (a) Comparison of proposed Qin scheduler with and without dynamic admission control for jobs completion time. (b) Comparison of proposed Qin scheduler with and without dynamic admission control for energy usage.

cluster. The x-axis denotes the system local time, the y-axis represents the node load (collected Prometheus), and each plot-line represents a realtime worker load monitored by Kubernetes and Qin schedulers. From local time $\sim 18:23$ to $\sim 18:41$ Kubernetes Scheduler monitors the 5-worker while from $\sim 18:42$ to $\sim 18:55$ we plug and play Qin scheduler. We observe imbalanced peak loads among workers during the Kubernetes period, but improved balanced peak loads during the Qin period. Fig. 13 demonstrates that the "plug and play" Qin Scheduler is **useful** in the realtime on a practical cluster.

7.1.6 Discussion

Why does the proposed Qin scheduler outperform state-of-the-art schedulers? From a mathematical perspective, eq. 32 and Fig. 1 in APPENDIX A show that the imbalance $Im_c(t)$ at the cluster-level increases when the number of applications M and the size of each application α_m increase for the best choice scheduling methods. On the other hand, the imbalance $Im_c(t)$ is independent of the M and the α_m for the D-Choices Greedy scheduling method. The best choice scheduling method "lost memory" requires more steps to compensate for the imbalance compared to the D-Choice Greedy scheduling method; therefore, it deviates significantly from the optimal scheduling with increased M and α_m. In contrast, the D-Choices Greedy scheduling method "recalls memory" requires fewer steps to compensate for the imbalance and achieves stochastically undistinguishable imbalance after fewer iterations of applications allocation; thus, it is independent of the number of applications M and the size of each application α_m [51]. From a system perspective, the proposed Qin scheduler incorporates the D-Choices Greedy scheduling method and considers energy usage, interference, and heterogeneity directly, while the state-of-the-art schedulers (Kubernetes, Paragon, and Mage) use the best choice method and do not consider the energy usage of the CPU and memory. This indicates that the proposed Qin scheduler outperforms state-of-the-art schedulers: (*i*) in performance (focus solely on jobs completion time) because the incorporated D-Choices Greedy scheduling method leads to better load balancing among servers and thus reduces the applications queuing time in servers; (*ii*) in energy usage (focus solely on energy usage of CPU and memory) because of the integrated collaborative filtering based energy usage estimation (Kubernetes, Paragon, and Mage do not consider energy usage); and (*iii*) in EDP (focus solely on EDP) because of the combination of the D-Choices Greedy scheduling method and the collaborative filtering based energy usage, interference, and heterogeneity estimations.

7.2 Simulations at Node Level

7.2.1 Qin Scheduler Outperforms Sniper Scheduler When Optimizing Solely on Performance, Energy, and EDP

Comparison: At node level, we implement the proposed performance, energy usage, and EDP aware Qin schedulers as independent alternative schedulers inside Sniper simulator v7.2 [68] to avoid switching scheduling overhead (runtime configuring). We perform simulations using Splash-2 benchmarks [69]. For the energy aware Qin scheduler (optimizing solely on energy usage reduction), we set the maximum acceptable performance loss threshold from DVFS control in Fig. 1 to be 50%, and for the EDP aware Qin scheduler (optimizing EDP solely) we set the maximum acceptable performance loss threshold to be 10% (user can change the performance loss threshold). We select $d = 2$ for the node level D-Choices Greedy scheduling algorithm.

Fig. 14 shows the comparison of the obtained results with the proposed scheduling algorithm and the Sniper default scheduler in each of the three different focus optimizations. Fig. 14.a shows the comparison in terms of performance (normalized application runtime). The x-axis denotes the applications in the Splash-2 benchmarks, and the y-axis represents normalized application rumtime measured as cycles generated by the Sniper simulator. We normalize the application runtime, energy usage, and EDP according to $f_{norm} = (f - f_{min})/(f_{max} - f_{min})$, where f_{min} and f_{max} are the minimum and the maximum values of the respective objective cost function f (e.g., application runtime, energy usage, or EDP). We observe that the proposed performance aware Qin scheduler outperforms Sniper scheduler for most Splash-2 benchmark applications. Fig. 14.b and Fig. 14.c show similar improvement of the proposed Qin scheduler over Sniper scheduler in energy usage and EDP. These results indicate that the proposed node-level Qin scheduler can generate thread-to-core scheduling better than the Sniper scheduler in terms of application runtime (performance), energy usage, and EDP.

7.2.2 Discussion

Why does the proposed Qin scheduler outperform Sniper scheduler at the node-level? From a mathematical perspective, eq. 33 reveals that the proposed D-Choices Greedy scheduling method (used by Qin scheduler) achieves better load balancing over best choice scheduling methods (used by Sniper scheduler). Thus, it reduces the threads CPU queuing time and improves the application runtime on the node. From a system perspective, the proposed Qin scheduler combines the node-level scheduling and threads migration with DVFS based energy reduction, and thus improves the energy usage and the EDP over Sniper scheduler.

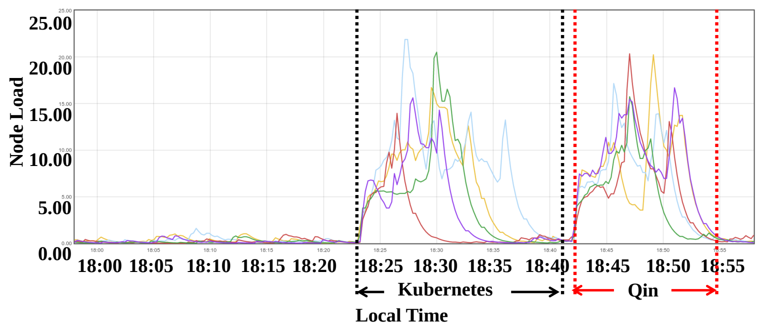


Fig. 13: Visualizing Kubernetes and Qin Schedulers in real time by the Prometheus tool [73].

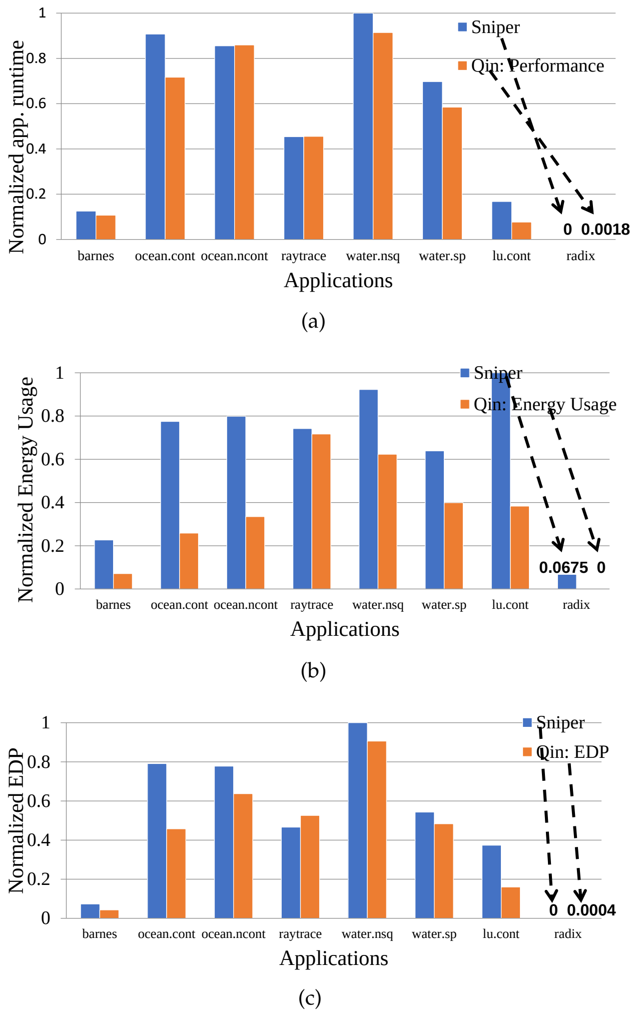


Fig. 14: Comparison of results at node level.

7.3 Hierarchical Cluster-Node Scheduling

7.3.1 Unified Hierarchical Scheduling Approach Outperforms State-of-the-art and Benefits The Literature

Comparison: In this section, we present the combined cluster-node levels hierarchical scheduling results - as the combination of the results obtained with the scheduling algorithms from the previous two sections. The workloads are first scheduled at the cluster level to generate the application-to-server scheduling. On each of the real cluster servers, we installed the Sniper simulator inside the docker container. Applications that were scheduled to these servers will arrive as inputs to the Sniper simulator instances, where the node level scheduling inside Sniper processes them. We use the longest jobs completion time (of all jobs scheduled to a given server) of all servers to represent the jobs completion time of the entire cluster. The sum of energy usage of all the servers gives the total energy usage of the entire cluster. The improvements of the proposed two-level hierarchical Qin scheduler over the combination of Kubernetes scheduler at cluster level + Sniper default scheduler at node level for 100 real-world application workloads are listed in Table 5. These results indicate that conducting a hierarchical integrated scheduling may provide benefits over the cluster and node level scheduling conducted in isolation separately. In our future work, we plan to deploy the node level Qin scheduler on the real servers (as opposed to inside the Sniper simulator) and thus achieve a better hierarchical integration of the proposed scheduling algorithms.

Discussion: How does the proposed unified hierarchical scheduling approach benefit the literature? This work looks at scheduling as the combination of cluster and node levels in a unified approach - which outperforms state-of-the-art scheduling methods that focus separately on cluster or node levels. The proposed Qin scheduler has a global view of the cluster-node scheduling and provides a new angle (through the hierarchical approach) for design objectives optimization in heterogeneous datacenters.

TABLE 5: Improvement achieved by the proposed hierarchical Qin scheduler vs. Kubernetes + Sniper default scheduler for 100 real-world application workloads.

Cluster-Node levels Schedulers	Performance	Energy	EDP
Kubernetes + Sniper	10.2%	38.65%	41.98%

7.4 Discussion

New Applications: Qin scheduler performs online scheduling without prior knowledge of the new incoming applications. When a new application arrives, the Qin scheduler uses the profiling data it already has (previously profiled applications) and conducts a minimal fast profiling (0.05 seconds profiling on 2 random server configurations) of the new application to identify similarities between the new and known applications.

Profiling Overhead: We mitigate profiling overhead by (*i*) profiling application workloads periodically (when application workloads differ significantly from the previously profiled workloads) during the offline training stage and (*ii*) shortening the user-defined online profiling time.

Application Requirements: Qin scheduler can be configured to focus on performance, energy, or EDP for different application requirements (minimize execution time, minimize energy usage, performance-energy trade-off). Our future work will incorporate industry-level application requirements, such as priority, latency sensitivity, and affinity to specific cores.

Switching Scheduling Overhead: The proposed Qin scheduler can dynamically switch its focus between performance, energy, and EDP. To minimize the switching overhead, we implement each scheduler with a different focus as an independent custom scheduler managed by the Kubernetes platform to avoid configuring the Qin scheduler at runtime. In other words, we have produced a set of three Qin schedulers implemented on the Kubernetes platform.

Scalability: Fig. 1 in APPENDIX A indicates that when the number of servers increases, although the imbalance of the traditional best choice scheduling methods decreases, their imbalance remains in order of factor more significant than the proposed D-Choices Greedy Scheduling method. Practically, Fig. 11 reveals that Qin Scheduler scales well when the number and size of application workloads increase. Therefore, Qin Scheduler scales well in larger systems theoretically and practically.

Open Source: "If you want to go fast, go alone. If you want to go far, go together." - African Proverb [55]. Thanks to the open-source Kubernetes project, Qin scheduler has the potential to be integrated into Kubernetes (as an alternative scheduler), which has been widely deployed on Amazon AWS - Amazon Elastic Kubernetes Service [56], Google Cloud Platform - Google Kubernetes Engine [57], Microsoft Azure - Azure Kubernetes Service [58], and IBM Cloud - IBM Cloud Kubernetes Service [59]. We will make the entire implementation of the Qin scheduler publicly available so that it can be integrated into the Kubernetes project. This will enable further research and ease of results duplication and comparison.

8 Conclusion

This paper proposed a solution to the problem of reducing energy usage without degrading performance in heterogeneous datacenters. The proposed solution does that through a novel hierarchical scheduling approach that models interference and heterogeneity while being able to focus the optimization on jobs completion time, energy usage, or EDP separately. The unified cluster-node scheduling was formulated as a multi-objective optimization problem. Specifically, we focused on application-to-server (cluster-level) and thread-to-core (node-level) problems. We developed a novel D-choices based greedy scheduling algorithms to solve these problems. To further improve the scheduling performance and energy usage, we developed a dynamic admission control protocol to shorten application queue time, thereby improving jobs completion time and energy usage. Experiments using both real-world and synthetic workloads on a real six-node in-house cluster demonstrated the superiority of the proposed scheduling approach, which outperformed state-of-the-art schedulers from industry and academia by around 10% in completion time, 39% in energy usage, and 42% in EDP. This paper demonstrated a unified approach as a promising direction in optimization for energy and performance of heterogeneous servers and datacenters.

References

[1] L. Su, "Innovation For the Next Decade of Compute Efficiency," *ISSCC*, 2023.

[2] N. Jones, "How to stop data centers from gobbling up the world's electricity," *nature*, 2018.

[3] H. Lavi, "Measuring greenhouse gas emissions in data centres: the environmental impact of cloud computing," *climatiq*, 2022.

[4] Annual Energy Outlook, U.S. Energy Information Administration (EIA), 2016. [Online]. Available: http://www.eia.gov/forecasts/aeo/data.cfm#enconsec

[5] United States Environmental Protection Agency, "Report to Congress on server and data center energy efficiency," *Report*, 2007.

[6] L.A. Barroso and U. Holzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines," *Morgan and Claypool Synthesis Series on Computer Architecture*, 2009.

[7] W. Guan and C. Ababei, "Unified Cross-layer Cluster-node Scheduling for Heterogeneous Datacenters," *Int. Green and Sustainable Computing Conference (IGSC)*, 2022.

[8] A. Krishnakumar et al., "Runtime task scheduling using imitation learning for heterogeneous many-core systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, 2020.

[9] M.G. Moghaddam and C. Ababei, "Dynamic Energy Management for Chip Multi-processors Under Performance Constraints," *Elsevier Microprocessors and Microsystems*, 2017.

[10] M.G. Moghaddam et al., "Dynamic Energy Optimization in Chip Multiprocessors Using Deep Neural Networks," *IEEE TMSCS*, vol. 4, no. 4, 2018.

[11] H. Kasture et al., "Rubik: Fast Analytical Power Management for Latency-Critical Systems," *MICRO*, 2015.

[12] C.H. Hsu et al., "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," *HPCA*, 2015.

[13] L. Zhou et al., "Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines," *MICRO*, 2020.

[14] S. Chen et al., "ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud," *HPCA*, 2022.

[15] G. Papadimitriou et al., "Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs," *HPCA*, 2019.

[16] M.E. Haque et al., "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," *MICRO*, 2020.

[17] N. Kulkarni et al., "CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores," *MICRO*, 2020.

[18] S.Chen et al., "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," *ASPLOS*, 2019.

[19] H. Li et al., "MSGD: A Novel Matrix Factorization Approach for Large-Scale Collaborative Filtering Recommender Systems on GPUs," *TPDS*, 2018.

[20] H. Li et al., "CuSNMF: A Sparse Non-negative Matrix Factorization Approach for Large-Scale Collaborative Filtering Recommender Systems on Multi-GPU," *ISPDP*, 2017.

[21] J. Peng et al., "HEA-PAS: A hybrid energy allocation strategy for parallel applications scheduling on heterogeneous computing systems," *Journal of Systems Architecture*, 2022.

[22] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ASPLOS*, 2013.

[23] W. Xiao et al., "Gandiva: Introspective Cluster Scheduling for Deep Learning," *OSDI*, 2018.

[24] D. Narayanan et al., "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," *OSDI*, 2020.

[25] A. Moaddeli et al., "The Power of d Choices in Scheduling for Data Centers with Heterogeneous Servers," *arXiv*, 2019.

[26] Q. Chen et al., "Prophet: Precise QoS Prediction on Non-Preemtive Accelerators to Improve Utilization in Warehouse-Scale Computers," *ASPLOS*, 2017.

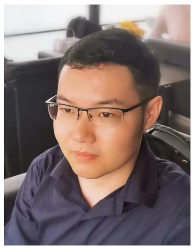
[27] T. Patel et al., "CLITE: Efficient and QoS-Aware Co-location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," *HPCA*, 2020.

[28] S. Wang et al., "Poly: Efficient Heterogeneous System and Application Management for Interactive Applications," *HPCA*, 2019.

[29] T.N. Le et al., "AlloX: Compute Allocation in Hybrid Clusters," *EuroSys*, 2020.

[30] D. Yi et al., "Efficient Compute-Intensive Job Allocation in Data Centers via Deep Reinforcement Learning," *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 6, pp. 1474-1485, 2020.

[31] N. Kulkarni et al., "Pliant: Leveraging Approximation to Improve Datacenter Resource Efficiency," *HPCA*, 2019.
[32] F. Romero et al., "INFaas:Automated Model-less Inference Serving," *ATC*, 2021.
[33] D. Mendoza et al., "Interference-Aware Scheduling for Inference Serving," *EuroMLSys*, 2021.
[34] N. J. Yadwadkar et al., "A Case for Managed and Model-less Inference Serving," *HotOS*, 2019.
[35] J. Zhang et al., "Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems," *HotCloud*, 2020.
[36] K. Haghshenas et al., "Infrastructure Aware Heterogeneous-Workloads Scheduling for Data Center Energy Cost Minimization," *IEEE Trans. on Cloud Computing*, 2022.
[37] N. Hogade et al., "Minimizing Energy Costs for Geographically Distributed Heterogeneous Data Centers," *IEEE Trans. on Sustainable Computing*, 2018.
[38] Z. Zhou et al., "An Adaptive Energy-Aware Stochastic Task Execution Algorithm in Virtualized Networked Datacenters," *IEEE Trans. on Sustainable Computing*, 2022.
[39] A. M. Al-Qawasmeh et al., "Power and Thermal-Aware Workload Allocation in Heterogeneous Data Centers," *IEEE Trans. on Computer*, 2015.
[40] H. Sun et al., "Energy-efficient and thermal-aware resource management for heterogeneous datacenters," *Sustainable Computing*, 2014.
[41] F. Romero and C. Delimitrou, "Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems," *PACT*, 2018.
[42] B. Hindman et al., "Mesos: a platform for fine-grained resource sharing in the data center," *NSDI*, 2011.
[43] Z. Wang et al., "Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler," *SoCC*, 2019.
[44] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," *COMPSTAT*, 2010.
[45] M. Lin and et. al., "Online algorithms for geographical load balancing," *IGSC*, 2012.
[46] S. Bang and et. al., "Run-time adaptive workload estimation for dynamic voltage scaling," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 9, pp. 1334-1347, 2009.
[47] A. Rajaraman et al., "Textbook on Mining of Massive Datasets," 2011.
[48] M. McKeown et al., "Power and Energy Characterization of an Open Source 25-core Manycore Processor," *HPCA*, 2018.
[49] J. Mars et al., "Bubble-Up: Increasing Utilization in Modern Warehouses Scale Computers via Sensible Co-locations," *MICRO*, 2011.
[50] C. Delimitrou et al., "iBench: Quantifying Interference for Datacenter Applications," *IISWC*, 2013.
[51] P. Berenbrink et al., "Balanced Allocations: The Heavily Loaded Case," *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing*, vol. 35, no. 6, 2006.
[52] U. Wieder, "Balanced Allocations with Heterogeneous Bins," *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.
[53] P.B. Godfrey, "Balls and bins with structure: balanced allocations on hypergraphs," *SODA*, 2008.
[54] C. Greenhill et al., "Balanced Allocation on Dynamic Hypergraphs," *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, vol. 176, 2020.
[55] V. J. Reddi et al., "Machine Learning Systems with TinyML," 2023. [Online]. Available: https://harvard-edge.github.io/cs249r_book/
[56] Amazon Elastic Kubernetes Service (EKS), 2023. [Online]. Available: https://aws.amazon.com/eks/
[57] Google Kubernetes Engine (GKE), 2023. [Online]. Available: https://cloud.google.com/kubernetes-engine
[58] Azure Kubernetes Service (AKS), 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/kubernetes-service
[59] IBM Cloud Kubernetes Service, 2023. [Online]. Available: https://www.ibm.com/cloud/kubernetes-service
[60] Amazon AWS Admission Control, 2022. [Online]. Available: https://aws.amazon.com/cn/blogs/big-data/enhance-resiliency-with-admission-control-in-amazon-opensearch-service/
[61] Google Cloud Platform Admission Control, 2022. [Online]. Available: https://cloud.google.com/service-infrastructure/docs/admission-control
[62] Microsoft Azure Admission Control, 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/secure/access-control
[63] IBM Cloud Admission Control, 2022. [Online]. Available: https://cloud.ibm.com/docs/containers?topic=containers-security
[64] Marquette Raj cluster, 2021. [Online]. Available: https://www.marquette.edu/news-center/2021/marquette-s-high-performance-computing-cluster-to-officially-open-june-2.php
[65] Y. Zhang et al., "HPC Data Center Participation in Demand Response: An Adaptive Policy With QoS Assurance," *IEEE Tran. on Sustainable Computing*, vol. 7, no. 1, 2022.
[66] C. Delimitrou et al., "QoS-aware scheduling in heterogeneous datacenters with Paragon," *ACM Trans. on Computer Systems*, vol. 31, no. 4, 2013.
[67] Kubernetes, 2021. [Online]. Available: http://k8s.io
[68] T.E. Carlson et al. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.
[69] S.C. Woo et al., "The SPLASH-2 programs: characterization and methodological considerations," *ISCA*, 1995.
[70] J. Leverich et al., "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," *EuroSys*, 2014.
[71] PARSEC 3.0-Beta, 2015, [Online]. Available: http://parsec.cs.princeton.edu
[72] J.P.Fricker, "The Cerebras CS-2: Designing an AI Accelerator Around the World's Largest 2.6 Trillion Transistor Chip," *DAC*, 2023.
[73] Prometheus, 2022, [Online]. Available: https://prometheus.io/



Wenkai Guan – received Ph.D. and M.S. degrees from Marquette University and a B.S. degree from Wuhan University of Technology in China. He was a research assistant at the Huazhong University of Science and Technology in China. He is a tenure-track assistant professor at the University of Minnesota, Morris, USA. His research passions include environmental sustainability in computing, machine learning and datacenter, and multicore embedded systems. Dr. Guan strives to be an advocate for gender equity – especially for women in computing/engineering.

Cristinel Ababei - received the Ph.D. degree in electrical and computer engineering from the Univ. of Minnesota, Minneapolis, in 2004. He is an associate professor in the Dept. of ECE, Marquette Univ. Prior to that, from 2012 to 2013, he was an assistant professor in the Dept. of EE, SUNY at Buffalo. Between 2008 to 2012, he was an assistant professor in the Dept. of ECE, North Dakota State University. From 2004 to 2008, he worked for Magma Design Automation, Silicon Valley. His current research interests include electronic design automation of systems-on-chip with emphasis on reliability and energy consumption, datacenters, parallel computing, and FPGAs.