Unified Cross-layer Cluster-node Scheduling for Heterogeneous Datacenters

Wenkai Guan and Cristinel Ababei
Electrical and Computer Engr., Marquette University
E-mail: {wenkai.guan, cristinel.ababei}@marquette.edu

***Abstract*—In this paper, we present a two-level hierarchical scheduler for datacenters called Qin. The goal of the proposed scheduler is to exploit 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 real cluster demonstrate the proposed approach outperforms state-of-the-art schedulers by 10.2% in completion time, 38.65% in energy usage, and 41.98% in EDP.**

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

I. Introduction

One of the primary objectives in datacenters (DCs) is performance - as execution time of jobs from arrival time to completion time - in addition to power consumption. Both performance and power consumption are directly affected by the job scheduling algorithm. Another recent optimization avenue is the exploitation of hardware heterogeneity. Heterogeneity in datacenters comes from two different sources. First, servers become more heterogeneous among themselves due to continuous upgrades and maintenance. A second source of heterogeneity stems from the asymmetry of recent multicore proposals that integrate high-performance “big" and energy efficient “little" cores [1].

Previous work focused primarily on optimizations at individual cluster or node levels and typically addressed either performance or energy. In this paper, we present Qin - a cross-layer approach for scheduling in heterogeneous datacenters. Our main contributions are as follows: 1) We propose a unified hierarchical cluster-node scheduling considering interference and heterogeneity for multi-objetive optimizations. The proposed technique outperforms state-of-the-art scheduling techniques from industry and academia. 2) We present a multi-objective optimization formulation for the unified cluster-node scheduling for heterogeneous datacenters. More specifically, we focus on application-to-server (cluster-level) and the thread-to-core (node-level) problems. 3) We propose the D-choices greedy scheduling algorithms to solve these problems, and 4) We evaluate the proposed Qin scheduler on a real small cluster and on a simulated platform with real-world and synthetic workloads.

II. Related Work on Scheduling

In the category of *node level approaches*, the study in [2] proposed an imitation learning based scheduling framework for heterogeneous chip-multiprocessors systems capable of handling multiple applications exhibiting streaming behavior. The study in [3] 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 [24]. This work was further improved on by the study in [4], which proposed a node-level deep neural network (DNN) based energy optimization method under performance constraints, but, again based on using the Sniper’s scheduler. The work in [5] enhanced the default Linux scheduler for node-level scheduling in terms of the cores to which a new process should be assigned, and when one or more operating processes should be migrated to other cores. The study in [6] introduced an adaptive scheduling framework that matches the heterogeneity of the workload to the heterogeneity of the hardware. The work in [7] demonstrated the potential of reconfigurable cores for servers running latency-critical applications.

In the category of *cluster level approaches*, the work in [8] proposed Paragon, an online and scalable datacenter scheduler that is heterogeneity and interference aware. The work in [9] exploited the unique characteristics of deep learning workloads and proposed a new cluster scheduling framework to improve the latency and efficiency of training deep learning workloads. The study in [10] proposed Gavel, a heterogeneity-aware scheduler that generalizes a wide range of existing scheduling policies for heterogeneous datacenters. The work in [11] leveraged d choices technique to develop algorithms for better load balancing in MapReduce deployed on heterogeneous servers. However, none of the above scheduling approaches considered energy usage in their optimization.

In the category of *hierarchical approaches*, the Paragon scheduler was extended in [12] to consider heterogeneity at 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. The work in [13] proposed a distributed scheduling approach for hierarchical scheduling. The works in [14], [15] also studied hierarchical scheduling.

III. Prediction Methods and Problem Statement

A. Collaborative Filtering for Energy Usage Estimation at Cluster Level

To develop energy oriented scheduling algorithms, we use *collaborative filtering* techniques to predict the energy usage of workloads on different server configurations (SC). We use collaborative filtering for predicting the energy usage of an incoming application, which will run on different hardware platforms with selected V/F levels. We assume that a given server from a heterogeneous set can support several 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.

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 [17]. 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. Also, we need 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 2 seconds (fast profiling to avoid memory bursts) on any two server configurations with selected V/F levels. Then, we insert this application as a new row in matrix A. Lastly, we apply singular value decomposition (SVD) and PQ-reconstruction [16] that are used by the collaborative filtering technique to predict the missing scores of this application for all other server configurations.

When new applications arrive, 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.

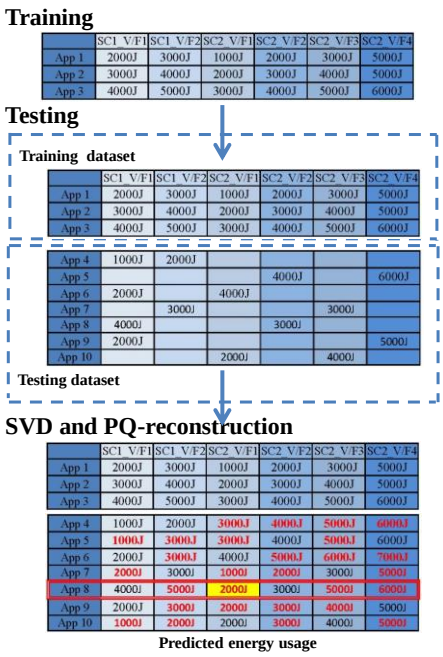


Fig. 1. Illustration of how collaborative filtering is employed to predict energy usage of applications executed on server configurations with different V/F settings.

B. Collaborative Filtering for Interference Estimation at Cluster Level

The work in [8], [18] 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. In this case, the rows of matrix A represent applications and the columns represent sources of interference (SoI), which include memory (capacity and bandwidth), cache hierarchy (L1/L2/L3 and translation look-aside buffers), and network and storage bandwidth. 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 [19].

C. 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 [8], 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. By using columns for all server configurations, we capture server heterogeneity within the cluster-level scheduling context.

D. 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 work in [3], 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.

E. 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, that minimize total interference, maximizes total heterogeneity, and minimizes total energy usage.

We minimize interference first because it was observed that interference can lead to higher performance loss than suboptimal server configurations [8]. From among the selected scheduling solutions that have minimum interference scores, we then find those that maximize heterogeneity scores. Lastly, from among those selected in the previous step, we identify the schedules that minimize energy usage scores - because our ultimate goal is to reduce energy usage under performance constraints.

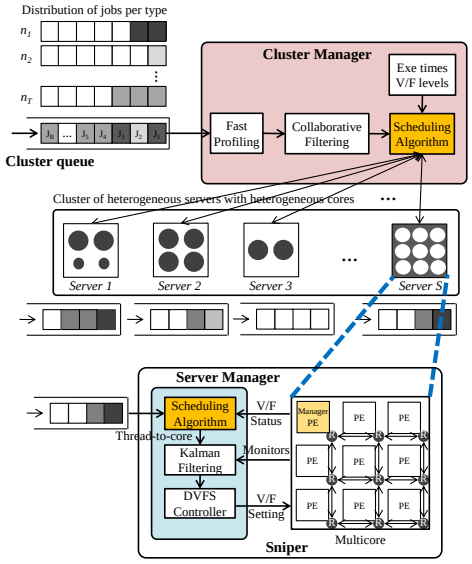


Fig. 2. Illustration of the proposed scheduling approach.

IV. Proposed Scheduling Approach

To solve the scheduling problem we propose Qin: a unified cross-layer cluster-node scheduling approach described in Fig. 2. 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 2 s to minimize the intrusion and avoid memory bursts); this is done by the function *BenchmarkProfiling*(m). Then, functions *EnergyPrediction*(m), *InterferencePrediction*(m), and *HeterogeneityPrediction*(m) use the collaborative filtering prediction technique to estimate the energy usage (Section III-A), interference (Section III-B), and heterogeneity (Section III-C) 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.

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

Algorithm 1: Cluster-level scheduling algorithm.

Inputs: Incoming jobs/applications M to cluster
Outputs: Application-to-server scheduling at cluster-level
Function ***CLUSTER-LEVEL-SCHEDULING()***
  for m *in* M **do**
    BenchmarkProfiling(m) // Fast online profiling
    E = EnergyPrediction(m) // Collaborative filtering based
    I = InterferencePrediction(m) // Collaborative filtering based
    H = HeterogeneityPrediction(m) // Collaborative filtering based
    D-ChoicesGreedyScheduling(E, I, H)
  end
end

Algorithm 2: Node-level scheduling algorithm.

Inputs: Incoming remain unfinished tasks/threads T to nodes
Outputs: Thread-to-core scheduling and DVFS at node-level
Function ***NODE-LEVEL-SCHEDULING()***
  for *Every time_interval* **do**
    // For every default power update time interval 1ms
    for t *in* T **do**
      // For every remain unfinished thread t, do nothing if no thread remains
      CollectCoreV/FStatus() // Fast core V/F collection
      E = EnergyPrediction(t) // Same as cluster-level
      I = InterferencePrediction(t) // Calculated as extra delay
      H = HeterogeneityPrediction(t) // Calculated as instruction count
      D-ChoicesGreedyScheduling(E, I, H)
    end
    PerCoreWorkloadPrediction() // Kalman filtering based
    DVFS() // Set V/F levels for each core
  end
end

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. 2. It implements the DVFS scheme reported in [3], 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.

A. D-Choices Greedy Scheduling

The D-Choices concept applied to the problem of cross-layer scheduling is one of the main novel contributions in this paper. We define and frame the application-to-server and thread-to-core problems through the so-called ball-to-bin model: allocating m balls into n *heterogeneous* bins (assuming $m \geq n \ln n$). In this model, when we use a *single choice* approach (i.e., $D = 1$, always choose the best bin with the least load), 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 (i.e., $D \geq 2$, choose top D bins with the least load first, then randomly choose one from among these D bins), the maximum load of bins is $m/n + O(\ln \ln n) + O(1)$ with high probability [20]–[22]. When applying this model of "balls into heterogeneous bins" to the problem of scheduling on heterogeneous servers, we must consider the following differences: 1) "Balls" represent various applications at cluster-level and various threads at node-level. 2) 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, 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, 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. 1) at the cluster-level (denoted as $Im_c(t)$), and of the cores (calculated with eq. 2) at the node-level (denoted as $Im_n(t)$).

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

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

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 becomes: allocate M applications to N heterogeneous servers, where each application m has the size of α_m, to minimize cluster-level imbalance $Im_c(t)$. Similarly, the thread-to-core problem becomes: allocate S_M 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. This is essentially the idea of the proposed D-ChoicesGreedyScheduling(E, I, H).

Mathematical Analysis: At the cluster-level, assuming $M \geq N \ln N$, we derive the upper bound of the imbalance $Im_c(t)$ based on the studies in [20]–[22] and eq. 1 as follows:

$\boldsymbol{D = 1:}$

$$\begin{aligned} Im_c(t) &= max\{L_i(t)|i \in 1,2,...,N\} - \textstyle\sum_{i=1}^{N} L_i(t)/N \\ &= \frac{\sum_{m=1}^{M} \alpha_m}{N} + O(\sqrt{(\textstyle\sum_{m=1}^{M} \alpha_m \ln N)/N}) - \frac{\sum_{m=1}^{M} \alpha_m}{N} \\ &= O(\sqrt{(\textstyle\sum_{m=1}^{M} \alpha_m \ln N)/N}) \end{aligned}$$

$\boldsymbol{D \geq 2:}$

$$\begin{aligned} Im_c(t) &= max\{L_i(t)|i \in 1,2,...,N\} - \textstyle\sum_{i=1}^{N} L_i(t)/N \\ &= \frac{\sum_{m=1}^{M} \alpha_m}{N} + O(\ln \ln N) + O(1) - \frac{\sum_{m=1}^{M} \alpha_m}{N} \\ &= O(\ln \ln N) + O(1) \end{aligned}$$

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

$$Im_c(t) = \begin{cases} O(\sqrt{(\sum_{m=1}^{M} \alpha_m \ln N)/N}), & if \;\; D = 1 \\ O(\ln \ln N) + O(1), & if \;\; D \geq 2 \end{cases} \tag{3}$$

At the node-level, assuming $S_M \geq (S_N \ln S_N)$, we derive the upper bound of the imbalance $Im_n(t)$ based on the studies in [20]–[22] and eq. 2 as follows:

$\boldsymbol{D = 1:}$

$$\begin{aligned} Im_n(t) &= max\{L_j(t)|j \in 1,2,...,S_N\} - \textstyle\sum_{j=1}^{S_N} L_j(t)/S_N \\ &= \frac{\sum_{k=1}^{S_M} \beta_k}{S_N} + O(\sqrt{(\textstyle\sum_{k=1}^{S_M} \beta_k \ln S_N)/S_N}) - \frac{\sum_{k=1}^{S_M} \beta_k}{S_N} \\ &= O(\sqrt{(\textstyle\sum_{k=1}^{S_M} \beta_k \ln S_N)/S_N}) \end{aligned}$$

$\boldsymbol{D \geq 2:}$

$$\begin{aligned} Im_n(t) &= max\{L_j(t)|j \in 1,2,...,S_N\} - \textstyle\sum_{j=1}^{S_N} L_j(t)/S_N \\ &= \frac{\sum_{k=1}^{S_M} \beta_k}{S_N} + O(\ln \ln S_N) + O(1) - \frac{\sum_{k=1}^{S_M} \beta_k}{S_N} \\ &= O(\ln \ln S_N) + O(1) \end{aligned}$$

Thus, at the node-level, when allocating S_M threads to S_N heterogeneous server configurations using D-Choices Greedy Scheduling method, the upper bound of the imbalance $Im_n(t)$ satisfies, with high probability:

$$Im_n(t) = \begin{cases} O(\sqrt{(\sum_{k=1}^{S_M} \beta_k \ln S_N)/S_N}), & if \;\; D = 1 \\ O(\ln \ln S_N) + O(1), & if \;\; D \geq 2 \end{cases} \tag{4}$$

We plot the relationship between the upper bound of the imbalance $Im_c(t)$ at the cluster-level and the number of heterogeneous servers N given by eq. 3 in Fig 3, with constant $\alpha_m = 1$ and different values for M (Note: the case of constant M and varying α_m leads to a similar plot). The relationship between the upper bound of the imbalance $Im_n(t)$ and the number of heterogeneous server configurations S_N at the node-level is similar. We observe 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 in imbalance reduction over traditional cluster-level best choice scheduling methods - which is the case of previous schedulers Paragon [8], Mage [12], and Kubernetes [23] 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 in imbalance reduction over traditional node-level best choice scheduling methods - which is the case of Sniper [24] scheduler - in load balancing. As mentioned earlier in Algorithms 1 and 2, we use the proposed D-Choices Greedy Scheduling method into the cluster-node scheduling approach described in Fig. 2.

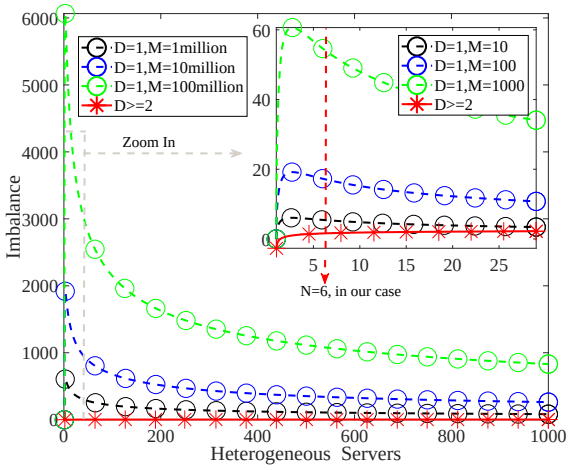


Fig. 3. Relationship between the upper bound of imbalance $Im_c(t)$ at cluster-level and number of heterogeneous servers N under the assumption $\alpha_m = 1$.

V. Experiments and Simulations

Server Systems: We implemented the cluster-level scheduling algorithm as a "plug-in" custom scheduler managed by the Kubernetes platform on an in-house real cluster built with six heterogeneous computers. The specific characteristics of the six computers are listed in Table I. 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 energy usage of CPU and memory.

Schedulers: At cluster-level, we compare the proposed Qin scheduler against the Kubernetes scheduler [23] (widely deployed on Amazon AWS, Google Cloud Platform, Microsoft Azure and IBM Cloud, assumed to be the best scheduling

TABLE I
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

TABLE II
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

approach from industry), and against Paragon [8] and Mage [12] schedulers (tested on major cloud computing services, assumed to be previous best schedulers from academia). At node-level, we compare the proposed Qin scheduler with the Sniper scheduler inside Sniper simulator v7.2 [24]. At the combined cluster-server levels, we compare the proposed unified cross-layer Qin scheduler versus the combination of Kubernetes scheduler at the cluster-level and Sniper scheduler at the node-level. Table II summarizes the compared schedulers in this paper.

Workloads: We conduct our evaluation using both real-world and synthetic application workloads. Similarly to the study in [8], we use Splash-2 benchmarks [25] 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 [12], we use the mutilate load generator [26] 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 applications and latency-critical applications, we use Parsec 3.0 benchmarks as well [27].

A. Experiments at Cluster Level

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 because we observed that it resulted in better load balancing. A summary of the comparison of the Qin scheduler against the other schedulers is presented in Table III for three types of applications: 100 real-world workloads, 100 latency-critical synthetic workloads, and 100 throughput-bound workloads.

Performance: Fig. 4.a shows the comparison in terms of normalized jobs completion time of the performance aware Qin scheduler vs. Kubernetes, Paragon, Mage schedulers for 100 real-world application workloads on 6-server heterogeneous cluster. 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*. Overall, the proposed performance aware Qin scheduler outperforms all other schedulers on average by 9.43% (Kubernetes), 32.39% (Paragon), and 26.15% (Mage), respectively. The improvement in jobs completion time gets even better as the number of workloads increases, which demonstrates a good scalability of the proposed scheduler.

Energy usage: Fig. 4.b shows the comparison of the normalized energy usage (of both CPU + memory combined, as shown in the y-axis) of the proposed energy aware Qin scheduler against Kubernetes, Paragon, Mage schedulers. Again, 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: Fig. 4.c shows the comparison in terms of normalized energy delay product (EDP, as shown in the y-axis). Again, the proposed EDP aware Qin scheduler outperforms the other schedulers on this dimension too.

Server Utilization: Fig. 5 shows the heat maps of the server utilization - calculated as average CPU utilization and collected by *Metrics API*) vs. time for Qin and Kubernetes schedulers for 100 synthetic application workloads on 5-worker + 1-master cluster. Fig. 5.a indicates that Qin scheduler achieves high and balanced servers utilization during the jobs completion time, while Fig. 5.b indicates that Kubernetes scheduler achieves good servers utilization in the middle period but shows *long tail* imbalanced servers utilization during the end 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 leads to load imbalance among servers. In contrast, the D-Choice Greedy scheduling method (used by Qin) results in better load balancing among servers and thus avoids this *long tail* phenomenon.

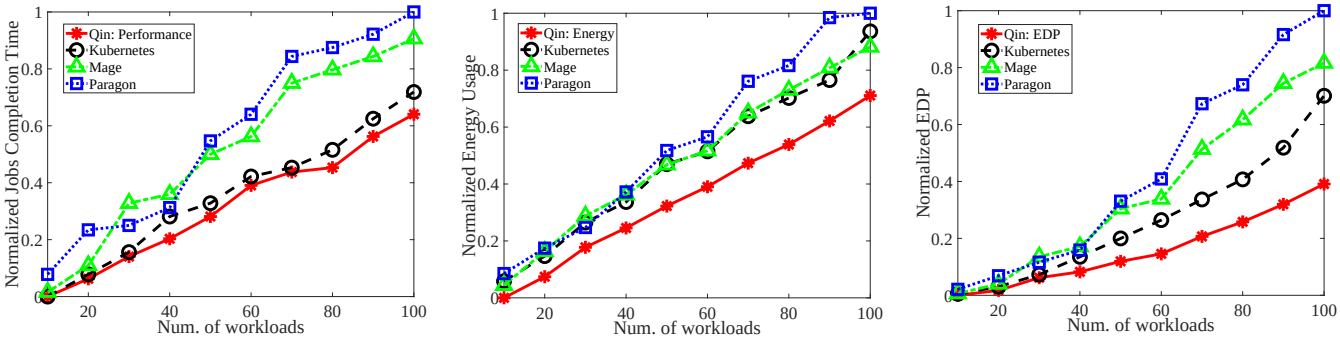
Scheduling Overheads: Fig. 6 shows the execution time breakdown of the Qin scheduler for 100 synthetic application workloads on the 6-server heterogeneous cluster. The overheads of the fast profiling and classification step and the D-Choices Greedy scheduling step are 2.23% and 1.27%, respectively. Overall, the Qin scheduler performs fast profiling, classification, and scheduling.

B. Simulations at Node Level

At the node level, we implement the proposed performance, energy usage, and EDP aware Qin schedulers as independent alternative schedulers inside Sniper simulator v7.2 [24]. We perform simulations using Splash-2 benchmarks [25]. For the energy aware Qin scheduler, we set the maximum acceptable performance loss threshold from DVFS control in Fig. 1 to

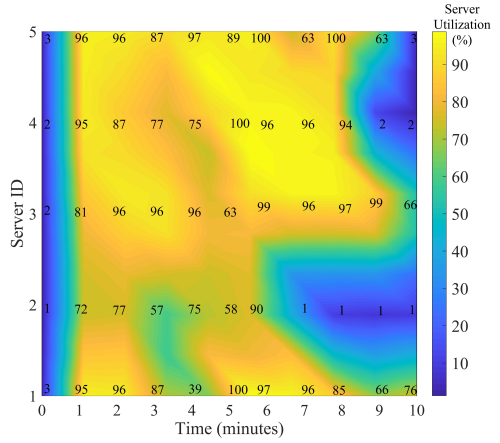
TABLE III
IMPROVEMENT ACHIEVED BY QIN SCHEDULER OVER THREE STATE-OF-THE-ART SCHEDULERS AT CLUSTER-LEVEL.

	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%

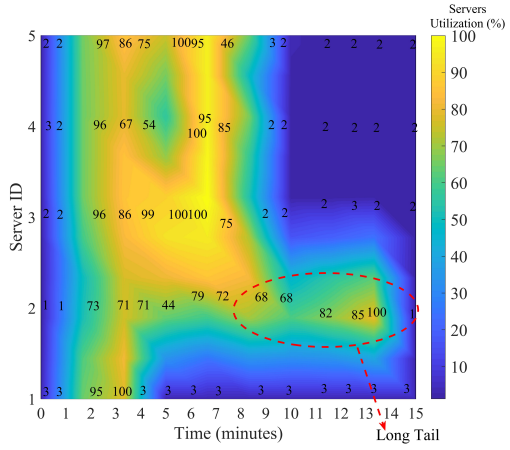


(a) (b) (c)

Fig. 4. 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.



(a)



(b)

Fig. 5. Server utilization vs. time: (a) Qin scheduler and (b) Kubernetes schedulers for 100 synthetic application workloads at the cluster-level.

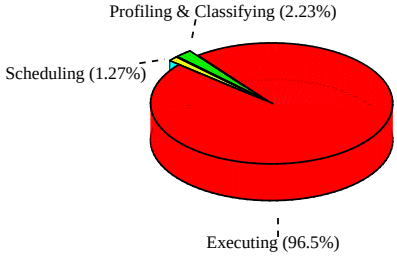


Fig. 6. Execution time breakdown for Qin scheduler for 100 synthetic application workloads at the cluster-level.

be 50%, and for the EDP aware Qin scheduler 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. 7 shows the comparison of the results obtained with the proposed Qin scheduler vs. the Sniper default scheduler. Fig. 7.a shows the comparison in terms of performance. The x-axis shows Splash-2 applications and the y-axis represents normalized application runtime measured as cycles reported by the Sniper simulator. We observe that the proposed performance aware Qin scheduler outperforms Sniper scheduler for most Splash-2 benchmark applications. Fig. 7.b and Fig. 7.c show similar improvement of the proposed Qin scheduler over Sniper scheduler in energy usage and EDP. The results from Fig. 7 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.

C. Cross-layer Cluster-Node Hierarchical Scheduling

In this section, we present the combined cluster-node levels hierarchical scheduling results - as the combination of the

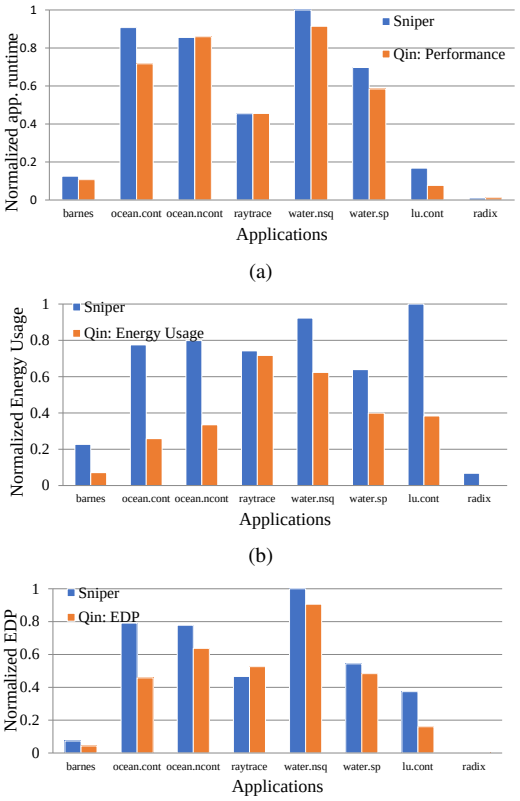


Fig. 7. Comparison of results at node level.

results obtained with the scheduling algorithms from the previous two sections. In collecting these results, 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 on a given server) of all servers to represent the jobs completion time of the entire cluster. The summation 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: 10.2% in terms of performance, 38.65% in energy use, and 41.98% in EDP. These results indicate that conducting a cross-layer 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 cross-layer integration of the proposed scheduling algorithms.

VI. Conclusion

We presented a unified cluster-node level hierarchical scheduling approach for heterogeneous datacenters that considers multiple design objectives. More specifically, we presented a cross-layer scheduling approach that models interference and heterogeneity while being able to focus the optimization on jobs completion time, energy usage, or EDP. Experiments using both real-world and synthetic workloads on a real six-node in-house cluster demonstrated the effectiveness of the proposed scheduling approach, which outperformed state-of-the-art schedulers from industry and academia.

VII. Acknowledgement

The first author is grateful to Qin Zhang, after whom the Qin scheduler is named in this paper.

References

[1] ARM big.LITTLE technology. [Online]. Available: http://www.arm.com/products/processors/technologies/biglittleprocessing.php
[2] A. Krishnakumar et al., "Runtime task scheduling using imitation learning for heterogeneous many-core systems," *IEEE TCAD*, 2020.
[3] M.G. Moghaddam and C. Ababei, "Dynamic Energy Management for Chip Multi-processors Under Performance Constraints," *Microprocessors and Microsystems*, 2017.
[4] M.G. Moghaddam et al., "Dynamic Energy Optimization in Chip Multiprocessors Using Deep Neural Networks," *IEEE TMSCS*, 2018.
[5] G. Papadimitriou et al., "Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs," *HPCA*, 2019.
[6] M.E. Haque et al., "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," *MICRO*, 2020.
[7] N. Kulkarni et al., "CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores," *MICRO*, 2020.
[8] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ASPLOS*, 2013.
[9] W. Xiao et al., "Gandiva: Introspective Cluster Scheduling for Deep Learning," *OSDI*, 2018.
[10] D. Narayanan et al., "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," *OSDI*, 2020.
[11] A. Moaddeli et al., "The Power of d Choices in Scheduling for Data Centers with Heterogeneous Servers," *arXiv*, 2019.
[12] F. Romero and C. Delimitrou, "Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems," *PACT*, 2018.
[13] Z. Wang et al., "Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler," *SoCC*, 2019.
[14] A.A. Bhattacharya et al., "Hierarchical Scheduling for Diverse Datacenter Workloads," *SoCC*, 2013.
[15] C. You et al., "Low Complexity Hierarchical Scheduling for Diverse Datacenter Jobs," *IEEE Communications Letters*, 2019.
[16] A. Rajaraman et al., "Textbook on Mining of Massive Datasets," 2011.
[17] M. McKeown et al., "Power and Energy Characterization of an Open Source 25-core Manycore Processor," *HPCA*, 2018.
[18] J. Mars et al., "Bubble-Up: Increasing Utilization in Modern Warehouses Scale Computers via Sensible Co-locations," *MICRO*, 2011.
[19] C. Delimitrou et al., "iBench: Quantifying Interference for Datacenter Applications," *IISWC*, 2013.
[20] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE TPDS*, 2001.
[21] U. Wieder, "Balanced Allocations with Heterogeneous Bins," *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.
[22] P. Berenbrink et al., "Balanced Allocations: The Heavily Loaded Case," *SIAM Journal on Computing*, 2006.
[23] Kubernetes, 2021. [Online]. Available: http://k8s.io
[24] T.E. Carlson et al. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," *Int. Conf. for HPC, Networking, Storage and Analysis*, 2011.
[25] S.C. Woo et al., "The SPLASH-2 programs: characterization and methodological considerations," *ISCA*, 1995.
[26] J. Leverich et al., "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," *EuroSys*, 2014.
[27] PARSEC 3.0-Beta, 2015, [Online]. Available: http://parsec.cs. princeton.edu