Intelligent middleware for HPC systems to improve performance and energy cost efficiency

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

INTELLIGENT MIDDLEWARE FOR HPC SYSTEMS TO IMPROVE PERFORMANCE AND ENERGY COST EFFICIENCY

by

YIJIA ZHANG

B.S., Peking University, China, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2021

Approved by

First Reader

Ayse K. Coskun, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Ioannis Ch. Paschalidis, PhD
Professor of Electrical and Computer Engineering
Professor of Systems Engineering
Professor of Biomedical Engineering

Third Reader

Martin Herbordt, PhD
Professor of Electrical and Computer Engineering

Fourth Reader

Vitus J. Leung, PhD
Principal Member of Technical Staff
Sandia National Laboratories

Wir müssen wissen.
Wir werden wissen. – David Hilbert (1930)

Acknowledgments

First, I would like to express my gratitude to my advisor, Prof. Ayse Coskun. This work would not have been possible without her guidance and help. I am also thankful to Prof. Ioannis Paschalidis, who advised me and provided substantial guidance to my research. I also appreciate the help from Prof. Martin Herbordt, Dr. Vitus J. Leung, Prof. Manuel Egele, Prof. Katzalin Olcoz, and Jim Brandt for their guidance in the HPC monitoring and resource management project. I thank Dr. Taylor Groves for providing me internship opportunity at the Lawrence Berkeley National Laboratory, which led to a joint work between the Berkeley Lab and BU.

It goes without saying, without the support of my parents, Binquan Zhang and Guorong Wang, this dissertation would not have been possible.

I also want to thank my collaborators, Hao Chen, Ozan Tuncer, Ata Turk, Fulya Kaplan, Daniel C. Wilson, Emre Ates, Burak Aksar, and Athanasios Tsiligkaridis. Owing to their collaboration, I was able to finish my projects and publish them in the past few years. I would like to thank Boston University and Sandia National Laboratories for funding most of my research. I also appreciate my colleagues in our research group: Onur, Tiansheng, Aditya, Zihao, Prachi, Mert, and Anthony. I will never forget the life with them, all the sharing and all the happiness. This goes to the ICSG and CAAD labs at BU as well.

Some contents in Chapter 2 are reprints of the material from the following paper:

- Yijia Zhang, Ioannis Ch. Paschalidis, and Ayse K. Coskun. Data Center Participation in Demand Response Programs with Quality-of-Service Guarantees. In ACM International Conference on Future Energy Systems (e-Energy), pp. 285-302, Jun. 2019.

Some contents in Chapter 3 are reprints of the material from the following papers:

- Yijia Zhang, Ozan Tuncer, Fulya Kaplan, Katzalin Olcoz, Vitus J. Leung, and Ayse K. Coskun. Level-Spread: A New Job Allocation Policy for Dragonfly Networks. In Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1123-1132, May. 2018.

- Yijia Zhang, Taylor Groves, Brandon Cook, Nicholas J. Wright, and Ayse K. Coskun. Quantifying the impact of network congestion on application performance and network metrics. In IEEE International Conference on Cluster Computing (Cluster), pp. 162-168, Sept. 2020.

INTELLIGENT MIDDLEWARE FOR HPC SYSTEMS TO IMPROVE PERFORMANCE AND ENERGY COST EFFICIENCY

YIJIA ZHANG

Boston University, College of Engineering, 2021

Major Professor: Ayse K. Coskun, PhD
Associate Professor of Electrical and Computer Engineering

ABSTRACT

High-performance computing (HPC) systems play an essential role in large-scale scientific computations. As the number of nodes in HPC systems continues to increase, their power consumption leads to larger energy costs. The energy costs pose a financial burden on maintaining HPC systems, which will be more challenging on future extreme-scale systems where the number of nodes and power consumption are expected to further grow. To support this growth, higher degrees of network and memory resource sharing are implemented, causing a substantial increase in performance variation and degradation. These challenges call for innovations in HPC system middleware that reduce energy cost without trading off performance.

By taking the performance of an HPC system as a first-order constraint, this thesis establishes that HPC systems can participate in demand response programs while providing performance guarantees through a novel design of the middleware. Well-designed middleware also enables enhanced performance by mitigating resource contention induced by energy or cost restrictions. This thesis aims to realize these goals through two complementary approaches.

First, this thesis proposes novel policies for HPC systems to enable their participation in emerging power markets, where participants reduce their energy costs by following market requirements. Our policies guarantee that the Quality-of-Service (QoS) of jobs does not drop below given constraints and systematically optimize cost reduction based on large deviation analysis in queueing theory. Through experiments on a real-world cluster whose power consumption is regulated to follow a dynamically changing power target, this thesis claims that HPC systems can participate in emerging power programs without violating the QoS constraints of jobs.

Second, this thesis proposes novel resource management strategies to improve the performance of HPC systems. Better resource management can mitigate contention that causes performance degradation and poor system utilization. To resolve network contention, we design an intelligent job allocation policy for HPC systems that incorporate the state-of-the-art dragonfly network topology. Our allocation policy mitigates network contention, reduces network communication latency, and consequently improves the performance of the systems. As some latest HPC systems support the collection of high-granularity network performance metrics at runtime, we also propose a method to quantify the impact of network congestion and demonstrate that a network-data-driven job allocation policy improves HPC performance by avoiding network traffic hot spots.

Contents

List of Tables

List of Figures

List of Abbreviations

AQA	Adaptive Quality-of-Service Assurance
CDF	Cumulative Distribution Function
GPS	Generalized Processor Sharing
HPC	High-Performance Computing
MGHPCC	Massachusetts Green High Performance Computing Center
MOC	Massachusetts Open Cloud
NeDD	Network-Data-Driven
RSRs	Regulation Service Reserves
SST	Structural Simulation Toolkit
QoS	Quality of Service
QoSCap	Quality-of-Service-aware-Capping
QoSG	Quality-of-Service Guarantee

Chapter 1

Introduction

1.1 Motivation and Key Contributions

HPC systems are playing an irreplaceable role by offering a substantial amount of computing services to the society. Many companies and research facilities are relying on the computing power provided by HPC systems around the world. However, HPC systems are large power consumers. In 2020, the top supercomputer in the world, the Fugaku supercomputer, consumed a peak power of 28 MW (TOP500, 2020), equivalent to more than \$100,000 of energy cost per day. It is also reported that, in 2014, all data centers in the US consumed 70 billion kWh, which is close to 2% of US electricity usage (Shehabi et al., 2016). As the trend of building large HPC systems is expected to continue, suppressing the increasing energy cost of HPC systems is a key challenge and vital to sustain their growth (Shehabi et al., 2016).

To support this growth of HPC system size and energy cost, higher degrees of computer, memory, and network resource sharing are implemented as a tradeoff, causing performance variation and degradation in these systems. Especially, in many latest HPC systems, network resource is heavily shared among different jobs, which results in significant network contention, leading to performance degradation. It has been widely reported that network congestion on HPC systems is a major cause of performance degradation (Bhatele et al., 2013a; Smith et al., 2016; Bhatele et al., 2020), leading to extention of job execution times (e.g., 6X longer than the optimal as reported in recent work (Chunduri et al., 2017)). These challenges call for innovations in HPC system middleware that mitigate the impact of network contention and reduce energy cost without trading off performance.

Participation in *demand response programs* is a profitable and environmentally beneficial answer to the challenge of HPC energy cost. In this thesis, we focus on *regulation service*, which is a specific type of a demand response program (New York Independent System Operator (NYISO), 2020). In regulation service programs, a power consumer is asked to regulate its power by following a target signal broadcast by a grid operator. The power consumer determines the average power value and the range of the power according to their own regulation capability, but the exact value of the power target is not known in advance and changes every few seconds. Participants of regulation service programs benefit from significant electricity cost reduction as long as they track the target within a small error margin (New York Independent System Operator (NYISO), 2020).

HPC systems are good candidates to provide regulation service reserves because many HPC systems are capable of quickly regulating their power usage within a large range through job scheduling and server power management. Previous works have demonstrated in simulation that participation in demand response could reduce the energy cost of HPC systems by 50% (Chen et al., 2014; Chen et al., 2019). However, those works do not provide quality-of-service (QoS)[1] guarantees on jobs running in HPC systems, which could discourage many potential participants since QoS is one of their top concerns. Also, prior works have not demonstrated HPC participation in demand response on real clusters and with real-world workload traces.

In this work, we design novel HPC middleware to meet the challenges discussed above and evaluate it on real clusters and using real-world workload traces. We present the thesis statement as following:

[1]In this work, QoS refers to timely execution of the computational jobs submitted to HPC systems. In other words, QoS requirement places a constraint on the delay of executing each job.

This thesis claims that HPC systems can successfully participate in demand response programs while providing performance guarantees through a novel design of the middleware. Well-designed middleware also enables enhanced performance by mitigating resource contention induced by energy or cost restrictions.

Therefore, we design the following middleware for HPC systems to improve performance and energy cost efficiency:

- **Strategies enabling HPC systems to participate in demand response programs with QoS assurance**:

  We propose a QoS-Guarantee (QoSG) policy as well as an Adaptive QoS-Assurance (AQA) policy that enable HPC systems to participate in demand response with QoS assurance of jobs. These policies schedule jobs based on the generalized processor sharing (GPS) algorithm (Parekh and Gallager, 1993), where theoretical guarantees on delay are realized and proven by queueing theory (Paschalidis, 1999; Bertsimas et al., 1999). In addition, we also propose a QoS-aware-Capping (QoSCap) policy that is adapted to workload traces with hours-/days-long jobs. In all three policies, in order to enable HPC systems to track a given power target, our policies schedule different types of jobs according to their properties (e.g., job size, execution time, QoS constraints, etc.) and adjust the power caps of servers. Our policies also select the optimal bidding parameters to participate in regulation service markets. Using both simulation and real-system experiments, we demonstrate that our policies outperform baseline policies (Chen et al., 2014; Chen et al., 2019) in terms of cost reduction and QoS assurance. We show that our policy is robust to different workload profiles, and we demonstrate that our policy reduces the electricity cost by 10-50% while abiding by QoS constraints (see Chapter 2 for further details).

- **Strategies to quantify and mitigate the impact of network congestion on HPC systems**:

  We design a novel job allocation policy, *Level-Spread*, for HPC systems with dragonfly networks based on the hierarchical structure of the dragonfly topology. This policy balances the communication on network links and mitigates network contention by spreading jobs within the smallest network level that a given job can fit in at the time of job allocation. We implement and release the Level-Spread policy in the SST simulator (Sandia National Laboratories, 2020). Through packet-level network simulations, we show that our proposed policy reduces the communication time by 16% on average compared to the state-of-the-art policies by harnessing node locality and balancing link congestion (see Chapters 3.2-3.4 for further details).

  To quantify the impact of network congestion, we conduct experiments on a 12k-node HPC system, Cori, where we run a diverse set of parallel applications while running network congestors simultaneously on nearby nodes. We collect application performance as well as Aries network counter metrics. We demonstrate that the sensitivity of parallel applications on HPC network congestion varies. Especially, we show that applications which spend a significant amount of time on MPI collective communications suffer large performance degradation (i.e., application execution time extended by up to 7x) under network congestion situation. We also demonstrate that certain Aries network metrics are good indicators of network congestion (see Chapters 3.5-3.6 for further details).

  Based on the detected correlation between network metrics and job performance, we propose a Network-Data-Driven (NeDD) job allocation policy for HPC systems. This policy monitors the network traffic on each node at runtime and prioritizes allocating jobs onto routers with less traffic. Through experiments on a large HPC system, Cori, we demonstrate that our proposed policy outperforms the baseline job allocation policy by 7-30% (see Chapters 3.7-3.8 for further details).

1.2 Related Work

In this section, we discuss the related work in HPC system demand response participation and HPC resource management.

1.2.1 HPC System Demand Response Participation

The problem of improving energy cost efficiency for HPC systems has received considerable attention in recent years. According to a survey (Maiterth et al., 2018), several major HPC centers around the world are employing or actively exploring energy-aware resource management strategies. It has been reported that these HPC centers are integrating job schedulers with power grid information, developing power-adaptive scheduling in SLURM, detecting power-hungry applications at runtime, or building power-capping infrastructures (Maiterth et al., 2018).

In recent years, there have been significant advances on integrating HPC systems with power markets. To enable HPC systems to participate in power markets, various power programs including peak shaving (Govindan et al., 2011), dynamic energy pricing (Le et al., 2016a), and emergency load reduction (Zhang et al., 2015; Tran et al., 2016) have been explored. Different kinds of policies have been proposed for HPC systems to participate in frequency control, regulation service, or operating reserves (Wang et al., 2019a; Cioara et al., 2018; Aksanli and Rosing, 2014; Ahmed et al., 2017; Chen et al., 2014; Chen et al., 2019). The impact of power limitations on the performance of HPC systems is also important. Using a model inspired by a real HPC system, Borghesi et al. (Borghesi et al., 2019) showed that it is possible to apply frequency scaling to save energy without penalizing users. Another recent trend is exploring the coordination of multiple HPC systems together in power markets. Some strategies are proposed to enable multiple HPC systems to collaborate in power markets and mitigate workload uncertainty (Yu et al., 2017; Niu and Guo, 2016).

In conjunction with these efforts, there has been a growing interest for HPC systems to participate in demand response. Prior works have explored strategies for HPC systems to participate in demand response by joint management of IT workloads with cooling facilities (Cupelli et al., 2018; Chen et al., 2014; Cioara et al., 2018), renewable energy sources (Le et al., 2016b; Pahlevan et al., 2020), energy storage devices (Shi et al., 2016; Pahlevan et al., 2020), or electric vehicles (Li et al., 2014). It has also been shown that without renewable energy sources or energy storage devices, computing servers are already capable of adjusting power consumption to meet the power target in demand response. Chen et al. (Chen et al., 2014) developed a heuristic power regulation policy for HPC systems to participate in the regulation service program through job scheduling, processor power capping, and server state transition. Chen et al. (Chen et al., 2019) also designed a QoS-aware policy for HPC systems to meet regulation service requirement while considering jobs' QoS. However, none of the existing works have a mechanism to provide QoS assurance to jobs.

In addition to the power efficiency work from the perspective of HPC systems, there are also approaches from the perspective of power markets. Clausen et al. presented a qualitative study of service contracts between electricity service providers and HPC systems in the United States and Europe (Clausen et al., 2019). Novel incentive mechanisms have been proposed to motivate power grids and HPC systems to participate in demand response (Paschalidis et al., 2012; Zhou et al., 2020; Wang et al., 2019b; Liu et al., 2014). For geo-distributed HPC systems, Sun et al. proposed an online auction mechanism for emergency demand response to motivate HPC systems to shuffle workload across multiple sites (Sun et al., 2016).

Our work focuses on regulating the power of computing servers in HPC systems to participate in regulation service programs. We propose policies that provide QoS assurance to jobs during the participation, and we evaluate our polices on real-world clusters.

1.2.2 HPC Job Allocation and Network Contention

Many prior works have observed the negative impact of job interference on the performance of HPC systems (Bhatele et al., 2013b; Leung et al., 2002; Jokanovic et al., 2015; Kambadur et al., 2012). Especially for dragonfly networks, several studies have explored the influence of job allocation on job performance. Jain et al. (Jain et al., 2014) compared the performance of six dragonfly-specific job allocation policies. They observed that random allocation is generally beneficial in spreading network traffic and reducing communication hot spots. We include these allocation policies as part of the baseline policies in our work (Sec. 3.3.4). Budiardja et al. (Budiardja et al., 2013) showed that spreading jobs to all groups during allocation distributes the network traffic, and thus, reduces congestion.

Yang et al. (Yang et al., 2016) observed performance degradation due to random allocation when multiple jobs run simultaneously on a dragonfly machine. They found that the network congestion caused by communication-intensive jobs greatly impacts the performance of other jobs.

Job allocation on traditional network topologies such as fat-tree also inspired our work. For example, Jokanovic et al. (Jokanovic et al., 2015) proposed a size-aware policy that alleviates communication interference by allocating large jobs on one side of the system and small jobs on the other side.

On the topic of link configuration and bandwidth for dragonfly machines, Groves et al. (Groves et al., 2016) analyzed the influence of link bandwidth on job execution times. Several studies compared different global link arrangements for dragonfly networks in terms of theoretical bisection bandwidth (Hastings et al., 2015; Belka et al., 2017). Routing algorithms for dragonfly networks have also been explored extensively (Garcia et al., 2013; Faizian et al., 2016; Jiang et al., 2009; Kim et al., 2008; Jain et al., 2014). Task mapping, which refers to mapping the tasks of a parallel application onto the processors of the allocated computing nodes, has also been studied on dragonfly networks (Prisacari et al., 2014b; Tuncer et al., 2017).

Network contention/congestion is an important topic in HPC research. It has been reported that, on HPC systems, inter-job network contention causes performance variability as high as 2X (Bhatele et al., 2013a), 2.2X (Smith et al., 2016), 3X (Bhatele et al., 2020), or 7X (Chunduri et al., 2017). Analysis has shown that network contention, rather than other factors such as OS noise, is the dominant reason for performance variability (Bhatele et al., 2013a; Smith et al., 2016; Bhatele et al., 2020).

Some prior works have analyzed the statistics of flit or stall counts on HPC systems. Jha et al. analyzed packet and stall count on a 3D-torus network and provided a visualization method to identify network hot spots (Jha et al., 2018). Brandt et al. measured network stall/flit ratio and showed its variation across links and over time (Brandt et al., 2016). These works have not analyzed the relationship between network counters and job performance.

A few works have explored the relationship between network metrics and job performance using machine learning. Jain et al. trained tree-based classifiers to predict job execution time on a 5D-torus system (Jain et al., 2013; Bhatele et al., 2015). They found that buffers and injection FIFOs are important metrics and that the hop-count metric is not helpful in predicting performance. On Cori, Groves et al. demonstrated strong correlations between communication latency and Aries network counters (Groves et al., 2017). They built machine learning models to forecast sub-optimal performance and identified network counters related to performance variability (Bhatele et al., 2020). Machine learning methods in this domain often focus on predicting performance or other outcomes; in contrast, our work's focus is on providing an analysis on selected network counters' role in understanding job performance.

Chapter 2

Data Center Participation in Demand Response with QoS Assurance

2.1 Introduction

Demand response offers opportunities for electricity consumers to significantly reduce their electricity cost if they can regulate their power consumption to follow market requirements (Hansen et al., 2014). The aim of demand response programs is to help stabilize the power grid and to balance the demand and the supply side of the power grid. Balancing supply and demand becomes increasingly important because the integration of renewable sources of energy into the grid poses significant challenges. As renewable supplies such as solar and wind energy depend on weather conditions, they are highly volatile and intermittent (Novoa and Jin, 2011). Demand response offers a solution to this challenge by absorbing the volatility of energy generation with demand-side regulation (Hansen et al., 2014), and power providers motivate consumers' participation by offering electricity bill reduction (New York Independent System Operator (NYISO), 2020).

As a specific form of demand response power programs, Regulation Service Reserves (RSRs) require participants to regulate their power consumption following a dynamically-changing power target that is updated every few seconds (New York Independent System Operator (NYISO), 2020). When participating in RSRs, users first determine their average consumption and the reserves they can provide. Users get more reduction in their electricity cost if they provide larger reserves, i.e., if they are capable of tracking a power target varying in larger range. After the average value and the reserve amount are selected, the real-time value of the target power consumption is calculated based on a signal provided by independent system operators (ISOs). This signal cannot be known in advance, but its statistical features are known. The electricity cost of a user is determined by its average power, the reserves it provides, and the tracking error which quantifies the difference between the target power and the user's actual power usage at every moment. Larger tracking error results in higher cost.

Data centers[1] play an essential role in fulfilling computational tasks in various domains, including analyzing commercial data, supporting online services, conducting scientific research, etc. However, data centers are significant power consumers. In 2014, the power consumption of data centers in US has reached 70 billion kWh, close to 2% of US electrical usage (Shehabi et al., 2016). The top-1 supercomputer in 2020, the Fugako supercomputer, consumes a peak power of 28 MW (TOP500, 2020), which generates a cost of more than $100,000 per day. Therefore, the electricity cost is a burden for data centers, and reducing the electricity cost of data centers will not only benefit existing data centers but also pave the way for future exascale computing systems.

Data centers are good candidates to participate in RSRs because they are capable of regulating their power consumption in a large range through various strategies, including job scheduling (Cioara et al., 2018), server state transition (Gandhi et al., 2012), server power capping (Reda et al., 2012), virtual machine (VM) provisioning (Chen et al., 2013), etc. For example, to reduce the power consumption of a data center, we can postpone the execution of some jobs by scheduling, or turn some

[1]In this thesis, we define data centers broadly, including both enterprise and high performance computing data centers.

servers into sleep states (or other power-saving states). We can also apply power caps on the processors by limiting their voltage and frequency. For data centers supporting virtual environments, one can also reduce the CPU/memory resource limits of VMs (Dhiman et al., 2009; Chen et al., 2013). Previous work has proposed power regulation policies for data centers to participate in RSRs, but without guarantees on the Quality-of-Service (QoS) of jobs (Chen et al., 2014; Chen et al., 2019). The QoS of jobs, in this work quantified by the jobs' sojourn time in a data center, is a vital index that data center users care about. It is unacceptable for data centers to participate in RSRs at the cost of violating the QoS constraints of jobs.

In this thesis, we first propse a policy called QoS-Guarantee (QoSG) that enables data centers to participate in RSRs while providing guarantees on the QoS of jobs. Later in Section 2.6, we also propose a policy called Adaptive-QoS-Assurance (AQA) that is an improved version of the QoSG policy. Our QoSG and AQA policies minimize the electricity cost by selecting the optimal values of the average power consumption and the reserve amount. The policies handle the scheduling of a heterogeneous workload by coordinating separate groups of servers to run different types of jobs following the spirit of a Generalized Processor Sharing (GPS) algorithm. As each type of jobs differs in their processing time, power profile, and QoS constraint, our policies assign different weights to each job type. These weights determine the ratio of the number of servers running each type of jobs, and the weights are also optimized to minimize electricity cost under the QoS constraints. The key in providing QoS guarantee is to determine an acceptable range for the policy parameters using a queueing theoretic result from Paschalidis et al. (Paschalidis, 1999; Bertsimas et al., 1999), which quantifies the probability of large delay when services are following the GPS algorithm.

To summarize, the contributions of this section of the thesis are as follows:

- We propose policies that enable data centers and HPC systems to participate in demand response programs with theoretically-proven guarantees on the QoS of jobs.

- We evaluate our policies in both large-scale simulations and real-system experiments on a cluster. We demonstrate that applying our policies can potentially reduce the electricity cost of HPC systems by 10-50% while abiding by job QoS constraints.

2.2 Background on Demand Response and Regulation Service Reserves

In electricity grid, the generation and consumption of power should be balanced in (near) real-time, otherwise catastrophic events such as blackouts happen. However, due to the rapidly growing trend of incorporating renewable energy sources in the power grid (Bohringer et al., 2009; EIA, 2014), ensuring the stability of the power grid becomes increasingly challenging and important because renewable supplies such as solar and wind are highly volatile and intermittent (Novoa and Jin, 2011). To meet this challenge, various demand response programs have been developed to help stabilize the power grid by motivating the demand side of the grid to adjust power consumption in response to power supply. Peak shaving (Govindan et al., 2011), dynamic energy pricing (Le et al., 2016a), and emergency load reduction (Zhang et al., 2015; Tran et al., 2016) are programs of this kind.

Starting from 2006, one of the largest US ISOs, PJM, has allowed electricity consumers to participate in reserve transactions (New York Independent System Operator (NYISO), 2020). Since then, capacity reserves have started to make significant contributions in stabilizing power systems. The capacity reserve programs allow the demand side to provide capacity reserves, where power consumers are required to regulate their power consumption to track a dynamic power target based on the amount of reserve that they intend to offer. By providing a larger amount of reserves, consumers receive a larger cost reduction as a reward.

There are mainly three types of capacity reserves, ordered from more to less valuable as follows: (1) *frequency control* requires power consumers to counter frequency deviations by modulating their consumption at near-real-time; (2) *regulation service reserves (RSRs)* asks consumers to react to a power target broadcast by independent system operators (ISOs) every few seconds; (3) *operating reserves* are offered in a slower pace where a power target maintains its value for up to a few hours. We target regulation service in this work as it is a particularly profitable choice for data centers because data centers are capable of efficiently regulating power in a few seconds and matching the signal update interval of regulation service programs.

To participate in RSRs, at the beginning of every hour, electricity consumers should determine their average power consumption $\bar{P}$ and the reserve R for this hour. Within this hour, the dynamically-changing power target $P_{target}(t)$ is computed by $\bar{P}$, R, and a signal $y(t)$ provided by ISOs, according to the formula:

$$P_{target}(t) = \bar{P} + y(t)R. \tag{2.1}$$

Assuming the actual power usage of a consumer at this moment is $P(t)$, we define the tracking error by

$$\epsilon(t) = \frac{|P(t) - P_{target}(t)|}{R}, \tag{2.2}$$

which quantifies the difference between the power target and the actual power usage. At the end of this hour, the tracking performance is evaluated by computing the average tracking error $\bar{\epsilon}$ over this hour. Then, the electricity cost for this hour can be estimated as

$$Cost = \left(\Pi^P \bar{P} - \Pi^R R + \Pi^\epsilon R \bar{\epsilon}\right) \times 1\text{h}, \tag{2.3}$$

where Π^P, Π^R, Π^ϵ are fixed monetary cost coefficients determined by the power market. As shown in Eq. (2.3), consumers will get higher cost reduction if they provide larger reserves by choosing a larger value of R, but they will also be penalized for large tracking error. In addition to the average error penalty, a consumer may lose his contract with ISOs if their instantaneous tracking performance is too poor to meet the tracking error constraints from ISOs. In this thesis, we use the following tracking error constraint in a probabilistic form:

$$\text{Prob}[\epsilon > 0.3] < 10\%, \tag{2.4}$$

which states that the tracking error is only allowed to exceed a threshold of 0.3 for less than 10% of the time.

2.3 Data Center and Workload Model

The power consumption of a data center consists of power from servers, cooling systems, and affiliated components such as network and storage systems. Because servers have a large power contribution and are typically more flexible than other systems, in this thesis, we focus on the regulation of server power through job scheduling and server power capping.

We assume servers are homogeneous: they consume the same amount of power when running the same job and they finish the execution of that job in the same amount of time (when under the same power cap setting). This assumption is an approximation to real data centers and is a prerequisite step towards future work considering hardware variations or data centers composed of multiple types of servers.

We group the computing jobs in a data center workload into different types according to their power consumption, processing time, and QoS constraints. We assign separate queues for different job types, and inside each queue, jobs are processed in a first-come-first-serve manner. In the following, we denote the number of job-type in a workload by J. A j^{th}-type job ($j = 1, 2, ..., J$) has a shortest processing time T^j and average power consumption p_j. We assume those values are known in advance from prior measurements. In case the job types are not known in advance, a data center could record and analyze the properties of the jobs running on it in a "data collection period" prior to participating in demand response. The jobs running in this "data collection period" can be classified into several types according to their processing time, power usage, and QoS constraint. Then, the policies proposed in this thesis can be applied.

Our AQA policy can be applied to data centers with parallel jobs that simultaneously occupy multiple nodes to run (whereas the QoSG policy only applies to single-node jobs). We assume no job consolidation, i.e., different jobs cannot share the same node, which is typical in many HPC systems due to efficiency and security considerations. Unless specifically mentioned, we assume jobs cannot be interrupted or stopped in the middle of their execution, which is common in high-performance computing systems, and the relaxation of this assumption is discussed in Sections 2.4.6 and 2.7.7.

Since we have jobs grouped into different types, we also group the servers according to the jobs they are running. At runtime, the servers are dynamically and conceptually partitioned into an idle-server group and J separate active-server groups, as shown in Fig. 2·1. When a j^{th}-type job is started on a server, the server switches from the idle-server group into the j^{th} active-server group, and switches back after the job finishes.

We define the quality-of-service (QoS) degradation of a job as the extra time used for processing the job compared to its minimum processing time T^j_{min}, calculated by the formula:

$$Q^j = \frac{T_{so} - T^j_{min}}{T^j_{min}}.$$

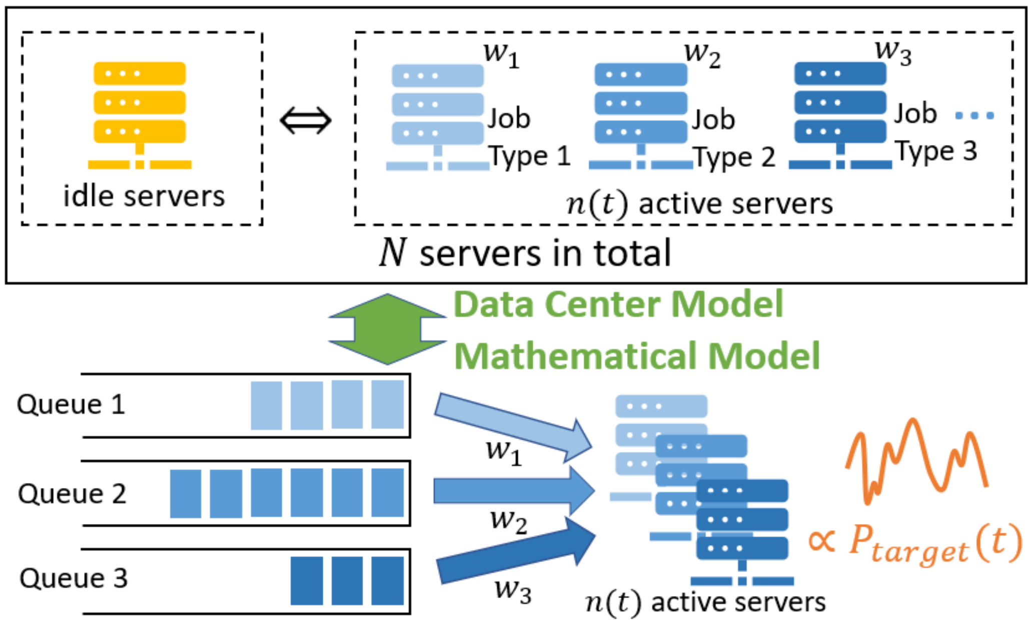


Figure 2·1: Our data center model. Jobs are grouped into different types and each type is processed in a separate queue. Servers are conceptually (not physically) grouped into an idle-server group and several active-server groups. Servers in the j^{th} active group run the j^{th}-type jobs. The total number of active servers $n(t)$ is adjusted to track the power target.

Here, minimum processing time T^j_{min} is defined as the time for processing a j^{th}-type job without power caps and without being delayed in the queue. The sojourn time T_{so} is defined as the time from a job's submission to completion, including the waiting time in the queue T_{wait} and the actual processing time T_{proc}:

$$T_{so} = T_{wait} + T_{proc}.$$

In this thesis, we focus on QoS constraints in a probabilistic form:

$$\text{Prob}[Q^j \geq Q^j_{thres}] \leq \delta^j \quad (j = 1, 2, ...), \tag{2.5}$$

where Q^j_{thres} is a given QoS threshold for the j^{th}-type jobs, and we set $\delta^j = 10\%$ in all of our evaluations. This formula means that only a small fraction (δ^j) of j^{th}-type jobs have a QoS degradation exceeding their QoS threshold Q^j_{thres}.

2.4 The QoS-Guarantee (QoSG) policy

In this section, we first give an overview of our QoSG policy. After that, we briefly explain the Generalized Processor Sharing (GPS) algorithm, which is used in developing our QoSG and AQA policies. We also introduce and generalize a theorem proven by Paschalidis et al. (Paschalidis, 1999; Bertsimas et al., 1999), which quantifies the delay in the GPS algorithm. Finally, we explain how our QoSG policy works at runtime in detail, as well as how our policy determines the optimal parameters that minimize the cost under QoS constraints.

2.4.1 An Overview of QoSG Policy

To enable data centers to participate in RSRs with theoretically-proven QoS guarantees, we design our QoSG policy following the spirit of a Generalized Processor Sharing (GPS) policy (Parekh and Gallager, 1993). When applied to the current context, according to the GPS algorithm, all the active servers are partitioned into several groups. Each group of servers executes one type of jobs. The number of servers in every group is determined by a set of weights, w_j (with $\sum_{j=1}^{J} w_j = 1$).

To follow the power target in Eq. (2.1) at runtime, our policy determines whether to start some jobs waiting in the queues as well as what power cap to be applied on each server. To be specific, with a given power target $P_{target}(t)$, our policy first computes the total number of servers that should be actively running jobs, $n(t)$. As we apply the GPS algorithm, approximately $w_j n$ servers should be running type-j jobs. Consequently, $n(t)$ can be solved by $P_{target} = \sum_{j=1}^{J} n w_j p_j + (N - n) p_{idle}$. Here, N is the number of servers in the data center. p_{idle} is the power consumption of an idle server, and p_j is the power consumption for running a type-j job. For each type of job, if the current number of servers running type-j jobs is less then the requirement determined by the GPS algorithm, some type-j jobs waiting in the queue will start running to match the requirement.

Since job scheduling alone cannot regulate HPC system power at very high granularity, after the waiting jobs are scheduled, we apply a power cap on each server to track the power target accurately. As there may not always exist a sufficient number of jobs in the queues, which prevents the actual power consumption to reach the target, we solve this mismatch by allowing the data center to have a queue of *standby jobs* with loose QoS constraints at the time-scale of hours/days. When there is a lack of a sufficient number of jobs in the other queues, some *standby jobs* will be started to guarantee good signal tracking.

At the beginning of every hour, our policy selects the optimal $\bar{P}$ (the average power), R (the reserves), as well as the weights w_j that determine how many servers to be allocated for each type of jobs. We select these parameters by solving an optimization problem that minimizes the monetary cost in Eq. (2.3) under the QoS constraints of jobs. The optimization formulation includes a constraint that guarantees QoS using a theorem proven by Paschalidis et al. (Paschalidis, 1999; Bertsimas et al., 1999). The theorem proves that, in the GPS algorithm, the number of jobs with large delay decreases exponentially as the delay increases. With this theorem, the constraints on the QoS of jobs are converted into constraints on $\bar{P}$, R, and w_j.

2.4.2 Generalized Processor Sharing (GPS) Algorithm

The Generalized Processor Sharing (GPS) algorithm is originally proposed to provide balanced performance in network scheduling (Parekh and Gallager, 1993). It assumes there are several streams of messages submitted to separate queues and processed by a single communication channel. Because these queues share the same channel, the policy partitions the channel's bandwidth and each part serves one of the queues.

With a given set of weights w_j $(j = 1, 2, ...)$ for the queues satisfying $\sum_j w_j = 1$, the GPS algorithm states that: (1) if all the queues are non-empty, the channel bandwidth will be partitioned according to the weights w_j; (2) if some queues are empty at a certain time, the portion of bandwidth for the empty queues will be shared by the non-empty queues; i.e., the channel bandwidth will be partitioned according to the weights of the non-empty queues. An advantage of the GPS algorithm is that there exists a minimal service rate for each queue. That is, assuming the channel has a fixed bandwidth B, the effective bandwidth of the j-th queue is at least w_jB, and this effective bandwidth will be larger if there is bandwidth sharing due to the existence of empty queues.

2.4.3 Generalization of a Theorem Quantifying Delay in GPS algorithm

Paschalidis et al. have proven a theorem that quantifies the delay of messages when processed by a stochastic channel following the GPS algorithm (Paschalidis, 1999; Bertsimas et al., 1999). Their theorem concerns the situation where two streams of

Table 2.1: Variables Used in the Problem Formulation

Label	Description
R	The reserve amount to participate in regulation service
$\bar{P}$	The average power to participate in regulation service
N	The total number of servers in the data center
p_{idle}	The power consumption of a server that is idle
J	The number of job types
p_j	The power consumption of a j^{th}-type job
λ_j	The average number of j^{th}-type jobs submitted to the data center per second
m_j	The number of servers (nodes) required to run each j^{th}-type job
T^j	The processing time for each j^{th}-type job
D^j	The delay of a j^{th}-type job, i.e., the time from submission to starting
D^j_{max}	The maximal delay that the majority of j^{th}-type jobs should satisfy
Q^j_{thres}	A threshold in the QoS constraint for j^{th}-type jobs
δ^j	A probability in the QoS constraint for j^{th}-type jobs
δ^j_D	A parameter calculated by δ^j
$A_j(t)$	The random variable representing the amount of work submitted to the j^{th} queue per second
$B(t)$	The random variable representing the total amount of service provided by the data center per second
$y(t)$	The ISO signal at time t, which is always within $[-1, 1]$
$n(t)$	The number of servers that should be active at time t
α_j	An empirically-determined coefficient in quantifying the probability of large delay
θ^*_j	A coefficient in the exponent quantifying the probability of large delay
w_j	The weights used in the GPS algorithm

messages are submitted to two queues respectively and processed by a single communication channel. Two queues are independent and the number of bits received by each queue in unit time follows a stochastic process. The bandwidth of the channel also varies from time to time following another stochastic process. Assuming the channel applies the GPS algorithm with fixed weights w_j (here $j = 1, 2$), the theorem quantifies the distribution of delay D_j, i.e., the waiting time for a certain bit of message to be processed in the queue j. They have proven that the portion of bits with large delay decreases exponentially according to

$$\text{Prob}[D_j \geq m] = \alpha_j e^{-m\theta_j^*} \quad (j = 1, 2) \tag{2.6}$$

as $m \to \infty$. Here, α_j is a coefficient to be determined empirically. θ_j^* is a coefficient that can be calculated based on the distribution of the bit arrival and the distribution of the channel bandwidth. This theorem is the key to providing QoS guarantees in our thesis, but first, we need to generalize it to multiple-queue scenarios. As mentioned in their thesis, a rigorous generalization of their theory to multiple-queue scenarios is particularly hard. The reason is that, to apply the GPS algorithm in multiple-queue scenarios, we need to analyze every case where some of the queues are empty while others are not. With the increase of the number of queues J, the number of cases increases exponentially with $O(2^J)$, making the analysis unapproachable.

To make the generalization of the theory to multiple-queue scenarios feasible, we make a "decoupling" assumption in our theoretical analysis: we assume different queues are decoupled from each other, so that the bandwidth will always be partitioned according to the weights w_j no matter whether there are empty queues or not. With this assumption, we do not need to analyze the exponentially large number of cases where some queues are empty. It deserves mentioning that this decoupling assumption does not reduce the validity of our theory. That is because if we provide

QoS guarantees by applying the GPS algorithm with this assumption, then applying the original GPS algorithm should also guarantee QoS because its effective bandwidth is never smaller.

Following the derivations in Ref. (Paschalidis, 1999; Bertsimas et al., 1999), we generalize their theorem to multiple-queue scenarios under the decoupling assumption. We prove that the portion of bits with large delay decreases exponentially as $m \to \infty$:

$$\text{Prob}[D^j \geq m] = \alpha_j e^{-m\theta_j^*}, \quad (j = 1, 2, ..., J). \tag{2.7}$$

The coefficients θ_j^* for the j-th queue can be calculated by

$$\theta_j^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\Lambda_B(-\theta w_j), \tag{2.8}$$

where the function $\Lambda_{GPS,j}(\theta)$ is defined as

$$\Lambda_{GPS,j}(\theta) = \Lambda_{A_j}(\theta) + \Lambda_B(-\theta w_j). \tag{2.9}$$

In this equation, Λ_{A_j} and Λ_B are the log moment-generating functions for the random variables $A_j(t)$, $B(t)$. Here, $A_j(t)$ represents the number of bits arriving in queue j per unit time, and $B(t)$ represents the channel bandwidth at time t. "sup" represents the supremum of the expression under the denoted condition.

The following derivations show how we arrive at Eq. (2.7) and Eq. (2.8): We derive Eq. (2.7) using Theorem 7 in Ref. (Paschalidis, 1999) and Theorem 7.2 in Ref. (Bertsimas et al., 1999).

Theorem 7 in Ref. (Paschalidis, 1999) proves that, in a two-class system under the GPS algorithm, as $m \to \infty$, the delay tail probability can be approximated by

$$\text{Prob}[D^j \geq m] \approx \alpha_j e^{-m\theta_j^*}, \quad (j = 1, 2). \tag{2.10}$$

Eq. (28) in Ref. (Paschalidis, 1999) gives

$$\theta_1^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,1}(\theta) < 0} \left[\Lambda_{A^1}(\theta) - \Lambda_{GPS,1}(\theta) \right], \tag{2.11}$$

where $\Lambda_{GPS,1}$ is defined by

$$\Lambda_{GPS,1} = \max \left[\Lambda^{\mathrm{I}}_{GPS,1}(\theta), \Lambda^{\mathrm{II}}_{GPS,1}(\theta) \right], \tag{2.12}$$

and $\Lambda^{\mathrm{I}}_{GPS,1}(\theta)$, $\Lambda^{\mathrm{II}}_{GPS,1}(\theta)$ are defined in Eqs. (22)(23) in Ref. (Paschalidis, 1999). Because $\Lambda^{\mathrm{I}}_{GPS,1}(\theta)$ corresponds to the case where the second queue is empty and the effective bandwidth of the first queue is larger than $w_1 B(t)$, we neglect this case following our decoupling assumption. This implies $\Lambda_{GPS,1} = \Lambda^{\mathrm{II}}_{GPS,1}(\theta)$, and

$$\theta_1^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,1}(\theta) < 0} \left[\Lambda_{A^1}(\theta) - \Lambda^{\mathrm{II}}_{GPS,1}(\theta) \right]. \tag{2.13}$$

In the proof of Theorem 7.2 in Ref. (Bertsimas et al., 1999), $\Lambda^{\mathrm{II}}_{GPS,1}(\theta)$ is given by

$$\Lambda^{\mathrm{II}}_{GPS,1}(\theta) = \sup_{a} \sup_{\substack{x_1 - w_1 x_3 = a \\ x_2 \geq w_2 x_3}} \left[\theta a - \Lambda^*_{A^1}(x_1) - \Lambda^*_{A^2}(x_2) - \Lambda^*_{B}(x_3) \right], \tag{2.14}$$

where $\Lambda^*(\cdot)$ is the Legendre transform of $\Lambda(\cdot)$, defined by

$$\Lambda^*(a) = \sup_{\theta} \left(\theta a - \Lambda(\theta) \right). \tag{2.15}$$

$\Lambda(\cdot)$ denote the log-moment generating functions, and $x_1(t)$, $x_2(t)$, $x_3(t)$ are the empirical rates of random process A^1, A^2, B, respectively. Because of the decoupling assumption, the influence of process A^2 can be removed from Eq. (2.14). Consequently, we get

$$\begin{aligned}
\Lambda^{\mathrm{II}}_{GPS,1}(\theta) &= \sup_{a} \sup_{x_1 - w_1 x_3 = a} [\theta a - \Lambda^*_{A^1}(x_1) - \Lambda^*_B(x_3)] \\
&= \sup_{x_1} \sup_{x_3} [\theta x_1 - \theta w_1 x_3 - \Lambda^*_{A^1}(x_1) - \Lambda^*_B(x_3)] \\
&= \sup_{x_1} [\theta x_1 - \Lambda^*_{A^1}(x_1) + \Lambda_B(-\theta w_1)] \\
&= \Lambda_{A^1}(\theta) + \Lambda_B(-\theta w_1). \qquad (2.16)
\end{aligned}$$

Combining Eq. (2.13) and Eq. (2.16), we conclude at

$$\theta_1^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,1}(\theta) < 0} -\Lambda_B(-\theta w_1). \qquad (2.17)$$

For a multi-queue system, because of our decoupling assumption, Eq. (2.17) holds for the first queue. Because all queues are equivalent to each other, we can directly generalize Eq. (2.17) by changing the subscripts and we arrive at Eq. (2.8).

2.4.4 Our QoSG Policy at Runtime

We first elaborate on how our policy schedules jobs and adjusts server power caps at runtime assuming the values of $\bar{P}$, R and the weights in the GPS algorithm w_j are already selected. The selection of these parameters will be discussed in Section 2.4.5.

To apply the GPS algorithm to the data center context, we regard the “channel bandwidth” as equivalent to the amount of service provided by the data center per second. Then, partitioning the bandwidth according to weights w_j is equivalent to partitioning the active servers in the data center into separate groups.

To let the data center's power consumption $P(t)$ follow the power target $P_{target}(t)$ in Eq. (2.1), at every moment, we control the total number of active servers $n(t)$ at this moment. Assume N is the total number of servers in the data center, p_{idle} is the power consumption of an idle server, and p_j is the power consumption for running a type-j job. According to the GPS algorithm with our decoupling assumption, there should be $n(t)w_j$ servers running type-j jobs at this moment. Then, the total power consumption of the data center is composed of the power from active servers, $\sum_{j=1}^{J} nw_j p_j$, and the power from idle servers, $(N-n)p_{idle}$. Thus, by matching the power target with the data center's power, we get

$$P_{target} = \bar{P} + y(t)R = \sum_{j=1}^{J} nw_j p_j + (N-n)p_{idle}. \tag{2.18}$$

We compute the number of servers that should be active at this moment as

$$n(t) = \frac{\bar{P} + y(t)R - p_{idle}N}{\sum_{j=1}^{J} w_j p_j - p_{idle}}. \tag{2.19}$$

Then, we determine the number of servers that should be in each group according to the weights w_j, and our policy schedules jobs to match those numbers. For example, if there are fewer servers running type-j jobs than there should be, the policy immediately starts some type-j jobs without power caps. On the other hand, if there are more servers running j-type jobs than there should be, the jobs waiting in the corresponding queue will not be processed.

However, because we assume interrupting a job in the middle is not allowed, the actual number of active servers may be larger than $n(t)$. To reduce the impact of this mismatch, our policy applies power caps on all active servers to fine-tune the power consumption. Assume the power of a type-j job can be adjusted from $p_{j,max}$ down to $p_{j,min}$ by applying power caps on servers. Let us denote the current number of active servers processing type-j jobs as n_j. Then, to track the power target better, for each job type j, our policy reduces the power cap on the corresponding servers to $p_{j,cap}$ ($p_{j,min} \leq p_{j,cap} \leq p_{j,max}$). That $p_{j,cap}$ is determined by letting the equation

$$\omega = \frac{p_{j,cap} - p_{j,min}}{p_{j,max} - p_{j,min}} \tag{2.20}$$

hold for every job type to ensure fairness. Here, $\omega \in [0,1]$ is a single number calculated by

$$P_{target} = \sum_{j=1}^{J} n_j p_{j,cap} + (N-n)p_{idle}. \tag{2.21}$$

2.4.5 Determining the Optimal $\bar{P}$, R and w_j

Our policy finds the optimal values of the average power $\bar{P}$, the reserves R, and the GPS algorithm weights w_j by minimizing the data center's cost in Eq. (2.3). Obviously, we should guarantee that the maximal power target $\bar{P}+R$ and the minimal power target $\bar{P}-R$ are achievable. Therefore, we have the following constraints:

$$\bar{P} + R \le N \cdot \max_j p_j, \tag{2.22}$$

$$\bar{P} - R \ge N \cdot p_{idle}. \tag{2.23}$$

Next, we derive the condition for guaranteeing the QoS of jobs. Based on the theorem introduced in Section. 2.4.3, we know that, to guarantee the QoS constraint for the type-j jobs, i.e.:

$$\text{Prob}[Q^j \ge Q^j_{thres}] \le \delta^j \tag{2.24}$$

is equivalent to the condition:

$$\text{Prob}\left[\frac{T^j_{wait} + T^j_{proc} - T^j_{min}}{T^j_{min}} \ge Q^j_{thres}\right] \le \delta^j \tag{2.25}$$

$$\Leftrightarrow \quad \text{Prob}[T^j_{wait} \ge Q^j_{thres} T^j_{min}] \le \delta^j \tag{2.26}$$

$$\Leftrightarrow \quad \text{Prob}[D^j \ge D^j_{max}] = \alpha_j e^{-D^j_{max}\theta^*_j} \le \delta^j \tag{2.27}$$

$$\Leftrightarrow \quad \theta^*_j \ge \delta^j_D = -\frac{1}{D^j_{max}} \ln\left(\frac{\delta^j}{\alpha_j}\right). \tag{2.28}$$

Here, because the actual processing time T^j_{proc} is usually close to the minimum T^j_{min}, we assume they are equal and we transform Eq. (2.25) into Eq. (2.26). As the delay D^j in Eq. (2.7) refers to the waiting time T^j_{wait}, we transform Eq. (2.26) into Eq. (2.27) after defining $D^j_{max} = Q^j_{thres} T^j_{min}$. The right hand side of Eq. (2.27) comes from Eq. (2.7).

As we mentioned in Section 2.4.3, to get an explicit expression for θ^*_j, we need the log moment-generating functions that characterize the stochastic processes of job arrival and job execution. To begin with, we denote the amount of service provided by the data center per second as $B(t)$, which corresponds to the "channel bandwidth" in Section. 2.4.3. Since the servers are homogeneous in the data center, each server provides the same amount of service per second. Let us set the amount of service provided by each server per second as 1, then the total amount of service provided by the data center per second is $n(t)$, where $n(t)$ is the current number of active servers. With this definition of "amount of service", a type-j job that takes T_j seconds to process will require T_j amount of service.

For the J types of jobs submitted to J separate queues, we assume the job arrival time in each queue follows a Poisson process[2], and we denote the parameter for the Poisson process as λ_j, which represents the average number of type-j jobs arriving every second. Then, the amount of work submitted to the j-th queue per second, $A_j(t)$, follows a Poisson distribution whose log moment-generating function is

$$\Lambda_{A_j}(\theta) = \lambda_j (e^{\theta T_j} - 1). \tag{2.29}$$

Practically, the processing time of a job increases when its power is capped, which makes T_j no longer a fixed number. In the following theoretical part, since server power-capping only plays a fine-tuning role in our policy, we assume the power usage and processing time of jobs are fixed. That is, we take $T_j = T^j_{min}$ and $p_j = p_{j,max}$. To handle situations where the processing time of jobs deviates significantly, we could

[2]Our approach is not limited to job arrivals following a Poisson process or ISO signals being Gaussian. Instead, the distributions of job arrivals and ISO signals can be general and our approach can still be applied. For other forms of distribution, the log moment-generating functions in Eqs. (2.29)(2.30) should be calculated accordingly.

change the log moment-generating functions accordingly if we know their probability distribution.

As discussed in Section 2.4.4, to participate in RSRs, our policy at runtime controls the total number of active servers $n(t)$ so as to adjust the data center power to match the power target, and $n(t)$ is derived in Eq. (2.19) according to the ISO signal $y(t)$.

Although we cannot know the ISO signal $y(t)$ in advance, its statistical features are usually stable. For the ISO signal samples that we use in our evaluation, $y(t)$ generally follows a normal distribution[2] with an average value $\bar{y} = 0$, and a standard deviation $y_\sigma = 0.40$. Then, from Eq. (2.19), we see that the number of active servers $n(t)$ should also follow a normal distribution with an average value

$$n_\mu = \frac{\bar{P} - p_{idle}N}{\sum_{j=1}^{J} w_j p_j - p_{idle}},$$

and a standard deviation

$$n_\sigma = \frac{y_\sigma R}{\sum_{j=1}^{J} w_j p_j - p_{idle}}.$$

Then, for the random variable $B(t) = n(t)$ that represents the amount of service processed by the data center per second, its log moment-generating function is

$$\Lambda_B(\theta) = n_\mu \theta + \frac{1}{2} n_\sigma^2 \theta^2. \tag{2.30}$$

Using Eq. (2.29) and Eq. (2.30), we calculate θ_j^* that quantifies the distribution of large delay by Eq. (2.8). Then, Eq. (2.28) tells us whether a set of values for $\bar{P}$, R, w_j meets the QoS constraints or not.

To summarize, our policy selects the optimal parameters $\bar{P}$, R, w_j that minimize monetary costs under QoS constraints by solving the following optimization problem:

$$\min_{\bar{P},R,w_j} \left(\Pi^P \bar{P} - \Pi^R R + \Pi^\epsilon R\bar{\epsilon}\right) \times 1\text{h} \tag{2.31}$$

$$\text{subject to} \quad \theta_j^* \geq \delta_D^j \quad (j = 1, 2, ..., J) \tag{2.32}$$

$$\delta_D^j = -\frac{1}{D_{max}^j} \ln\left(\frac{\delta^j}{\alpha_j}\right) \tag{2.33}$$

$$\theta_j^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\Lambda_B(-\theta w_j) \tag{2.34}$$

$$\Lambda_{GPS,j}(\theta) = \Lambda_{A_j}(\theta) + \Lambda_B(-\theta w_j) \tag{2.35}$$

$$\Lambda_{A_j}(\theta) = \lambda_j(e^{\theta T_j} - 1) \tag{2.36}$$

$$\Lambda_B(\theta) = n_\mu \theta + \frac{1}{2} n_\sigma^2 \theta^2 \tag{2.37}$$

$$n_\mu = \frac{\bar{P} - p_{idle} N}{\sum_{j=1}^J w_j p_j - p_{idle}} \tag{2.38}$$

$$n_\sigma = \frac{y_\sigma R}{\sum_{j=1}^J w_j p_j - p_{idle}} \tag{2.39}$$

$$\bar{P} + R \leq N \cdot \max_j p_j, \quad \bar{P} - R \geq N \cdot p_{idle} \tag{2.40}$$

$$\sum_{j=1}^J w_j = 1, \quad \bar{P}, R, w_j > 0 \tag{2.41}$$

After some calculations, we can simplify Eqs. (2.32) (2.34)-(2.37) into

$$n_\sigma \leq \frac{n_\mu}{\sqrt{2\delta_D^j}} \tag{2.42}$$

$$w_j \geq \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \equiv w_j^* \tag{2.43}$$

$$(j = 1, 2, ..., J).$$

To derive that, we first plug Eq. (2.37) into Eq. (2.34):

$$\sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta)<0} -\Lambda_B(-\theta w_j) \tag{2.44}$$

$$= \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta)<0} -n_\mu(-\theta w_j) - \frac{1}{2}n_\sigma^2(-\theta w_j)^2 \tag{2.45}$$

$$= \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta)<0} -\frac{1}{2}n_\sigma^2 w_j^2\left(\theta - \frac{n_\mu}{n_\sigma^2 w_j}\right)^2 + \frac{n_\mu^2}{2n_\sigma^2} \tag{2.46}$$

Whether the maximum point $\frac{n_\mu^2}{2n_\sigma^2}$ of the above quadratic function can be reached depends on whether $\theta = \frac{n_\mu}{n_\sigma^2 w_j}$ meets the conditions $\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0$.

To evaluate these conditions, we plug Eqs. (2.36) (2.37) into Eq. (2.35) and get

$$\Lambda_{GPS,j}(\theta) = \lambda_j(e^{\theta T_j} - 1) - n_\mu \theta w_j + \frac{1}{2}n_\sigma^2\theta^2 w_j^2. \tag{2.47}$$

As the second-order derivative $\Lambda''_{GPS,j}(\theta)$ is always positive, the function $\Lambda_{GPS,j}(\theta)$ is convex. As we already know 0 is a root of $\Lambda_{GPS,j}(\theta)$, there will be one positive root if

$$\Lambda'_{GPS,j}(\theta)\big|_{\theta=0} < 0 \tag{2.48}$$

$$\Leftrightarrow \quad w_j > \frac{\lambda_j T_j}{n_\mu} \tag{2.49}$$

$$\Leftrightarrow \quad n_\mu w_j > \lambda_j T_j, \tag{2.50}$$

and there will be no positive root otherwise. Actually, Eq. (2.50) is the requirement that the average computing service provided should be larger than the average work submitted for the j-th queue. We assume Eq. (2.50) is satisfied, otherwise the queue length will diverge. Then, there is a positive root for $\Lambda_{GPS,j}(\theta)$, and whether $\theta = \frac{n_\mu}{n_\sigma^2 w_j}$

meets the conditions $\Lambda_{GPS,j}(\theta) < 0$ can be converted to

$$\Lambda_{GPS,j}\left(\frac{n_\mu}{n_\sigma^2 w_j}\right) < 0 \tag{2.51}$$

$$\Leftrightarrow \quad \lambda_j(e^{\frac{n_\mu T_j}{n_\sigma^2 w_j}} - 1) < \frac{n_\mu^2}{2n_\sigma^2} \tag{2.52}$$

$$\Leftrightarrow \quad w_j > \frac{n_\mu T_j}{n_\sigma^2 \ln\left(1 + \frac{n_\mu^2}{2n_\sigma^2 \lambda_j}\right)} \tag{2.53}$$

Now, there are two cases.

Case I: If both Eqs. (2.49) (2.53) hold, from Eq. (2.46) we get

$$\theta_j^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta)<0} -\Lambda_B(-\theta w_j) = \frac{n_\mu^2}{2n_\sigma^2}. \tag{2.54}$$

Then, the condition in Eq. (2.32) is equivalent to

$$\theta_j^* \geq \delta_D^j \quad (j = 1, 2, ..., J) \tag{2.55}$$

$$\Leftrightarrow \quad n_\sigma \leq \frac{n_\mu}{\sqrt{2\delta_D^j}} \tag{2.56}$$

Case II: If Eq. (2.49) holds but Eq. (2.53) does not hold, i.e.,

$$w_j \leq \frac{n_\mu T_j}{n_\sigma^2 \ln\left(1 + \frac{n_\mu^2}{2n_\sigma^2 \lambda_j}\right)} \tag{2.57}$$

assuming θ_0 is the positive root of $\Lambda_{GPS,j}(\theta)$, i.e.

$$\Lambda_{GPS,j}(\theta_0) = \Lambda_{A_j}(\theta_0) + \Lambda_B(-\theta_0 w_j) = 0 \tag{2.58}$$

then the supremum in Eq. (2.44) is achieved at θ_0, i.e.,

$$\theta_j^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta)<0} -\Lambda_B(-\theta w_j) = -\Lambda_B(-\theta_0 w_j). \tag{2.59}$$

In this situation, we can convert the condition in Eq. (2.32) by

$$
\begin{aligned}
& \theta_j^* \geq \delta_D^j \quad (j = 1, 2, ..., J) && (2.60)\\
\Leftrightarrow\ & -\Lambda_B(-\theta_0 w_j) \geq \delta_D^j && (2.61)\\
\Leftrightarrow\ & \Lambda_{A_j}(\theta_0) \geq \delta_D^j && (2.62)\\
\Leftrightarrow\ & \lambda_j(e^{\theta_0 T_j} - 1) \geq \delta_D^j && (2.63)\\
\Leftrightarrow\ & \theta_0 \geq \frac{1}{T_j} \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right) \equiv \kappa && (2.64)\\
\Leftrightarrow\ & \Lambda_{GPS,j}(\kappa) \leq 0 && (2.65)\\
\Leftrightarrow\ & \lambda_j(e^{\kappa T_j} - 1) - n_\mu \kappa w_j + \frac{1}{2} n_\sigma^2 \kappa^2 w_j^2 \leq 0 && (2.66)\\
\Leftrightarrow\ & \delta_D^j - n_\mu \kappa w_j + \frac{1}{2} n_\sigma^2 \kappa^2 w_j^2 \leq 0 && (2.67)\\
\Leftrightarrow\ & \delta_D^j + \frac{1}{2} n_\sigma^2 \kappa^2 \left(w_j - \frac{n_\mu}{n_\sigma^2 \kappa}\right)^2 \leq \frac{n_\mu^2}{2n_\sigma^2} && (2.68)\\
\Leftrightarrow\ & \begin{cases} n_\sigma \leq \frac{n_\mu}{\sqrt{2\delta_D^j}} \\ w_j \geq \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \equiv w_j^* \end{cases} && (2.69)
\end{aligned}
$$

In the derivation from Eq. (2.68) to Eq. (2.69), we omit another requirement

$$
w_j \leq \frac{2\delta_D^j T_j}{\left(n_\mu - \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \tag{2.70}
$$

because it can be proven that Eq. (2.70) is already implied by Eq. (2.57).

Combining Case I and Case II, we see that to guarantee the QoS we need

$$
\begin{cases}
n_\sigma \leq \frac{n_\mu}{\sqrt{2\delta_D^j}} \\
w_j > \frac{\lambda_j T_j}{n_\mu} \\
w_j \geq \min\left\{ \frac{n_\mu T_j}{n_\sigma^2 \ln\left(1 + \frac{n_\mu^2}{2n_\sigma^2 \lambda_j}\right)}, \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \right\}
\end{cases}
$$

To further simplify the equations above, in the following, I will prove that there is always

$$\frac{n_\mu T_j}{n_\sigma^2 \ln\left(1+\frac{n_\mu^2}{2n_\sigma^2\lambda_j}\right)} > \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right)\ln\left(1+\frac{\delta_D^j}{\lambda_j}\right)} \tag{2.71}$$

and

$$\frac{\lambda_j T_j}{n_\mu} < \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right)\ln\left(1+\frac{\delta_D^j}{\lambda_j}\right)}. \tag{2.72}$$

In fact, defining $\alpha = \frac{n_\sigma^2}{n_\mu^2}$, we get

$$\frac{n_\mu T_j}{n_\sigma^2 \ln\left(1+\frac{n_\mu^2}{2n_\sigma^2\lambda_j}\right)} > \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right)\ln\left(1+\frac{\delta_D^j}{\lambda_j}\right)}$$

$$\Leftrightarrow \quad \ln\left(1+\frac{\delta_D^j}{\lambda_j}\right) > \left(1-\sqrt{1-2\alpha\delta_D^j}\right)\ln\left(1+\frac{1}{2\alpha\lambda_j}\right)$$

Define

$$H(\delta_D^j) = \ln\left(1+\frac{\delta_D^j}{\lambda_j}\right) - \left(1-\sqrt{1-2\alpha\delta_D^j}\right)\ln\left(1+\frac{1}{2\alpha\lambda_j}\right), \tag{2.73}$$

then we need to prove

$$H(\delta_D^j) > 0, \quad \delta_D^j \in (0, \frac{1}{2\alpha}). \tag{2.74}$$

First, we notice that

$$H(0) = H(\frac{1}{2\alpha}) = 0. \tag{2.75}$$

As

$$\frac{dH}{d\delta_D^j} = \frac{1}{\delta_D^j + \lambda_j} - \frac{\alpha}{\sqrt{1-2\alpha\delta_D^j}}\ln\left(1+\frac{1}{2\alpha\lambda_j}\right), \tag{2.76}$$

we have

$$\frac{dH}{d\delta_D^j}\Big|_{\delta_D^j=0} = \frac{1}{\lambda_j} - \alpha\ln\left(1+\frac{1}{2\alpha\lambda_j}\right). \tag{2.77}$$

Then, define $\gamma = \frac{1}{\lambda_j}$, we get

$$\frac{dH}{d\delta_D^j}\Big|_{\delta_D^j=0} = \gamma - \alpha\ln\left(1+\frac{\gamma}{2\alpha}\right) \equiv I(\gamma). \tag{2.78}$$

Because $I(0) = 0$ and

$$\frac{dI(\gamma)}{d\gamma} = \frac{\alpha + \gamma}{2\alpha + \gamma} > 0, \tag{2.79}$$

we know for $\gamma > 0$ there are

$$\left.\frac{dH}{d\delta_D^j}\right|_{\delta_D^j = 0} = I(\gamma) > 0. \tag{2.80}$$

From Eq. (2.76), we also have

$$\lim_{\zeta \to \frac{1}{2\alpha}^-} \left.\frac{dH}{d\delta_D^j}\right|_{\delta_D^j = \zeta} = -\infty. \tag{2.81}$$

Furthermore, $H(\delta_D^j)$ is a concave function because

$$\frac{d^2 H}{d(\delta_D^j)^2} = -\frac{1}{(\delta_D^j + \lambda_j)^2} - \frac{\alpha^2}{(1 - 2\alpha\delta_D^j)^{3/2}} \ln\left(1 + \frac{1}{2\alpha\lambda_j}\right) < 0. \tag{2.82}$$

Combing Eqs. (2.75) (2.80) (2.81) (2.82), we get the conclusion $H(\delta_D^j) > 0$.

Similarly, there are

$$\frac{\lambda_j T_j}{n_\mu} < \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \tag{2.83}$$

$$\Leftrightarrow \quad \frac{1 + \sqrt{1 - 2\alpha\delta_D^j}}{2} \ln(1 + \delta_D^j \gamma) < \delta_D^j \gamma. \tag{2.84}$$

Define

$$L(\gamma) = \frac{1 + \sqrt{1 - 2\alpha\delta_D^j}}{2} \ln(1 + \delta_D^j \gamma) - \delta_D^j \gamma, \tag{2.85}$$

we can simply prove $L(0) = 0$ and $\frac{dL}{d\gamma} < 0$, which results in $L(\gamma) < 0$ for $\gamma > 0$.

Summarizing the discussion above, we get the final form of the conditions to guarantee QoS as

$$\begin{cases} n_\sigma \leq \frac{n_\mu}{\sqrt{2\delta_D^j}} \\ w_j \geq \frac{2\delta_D^j T_j}{\left(n_\mu + \sqrt{n_\mu^2 - 2\delta_D^j n_\sigma^2}\right) \ln\left(1 + \frac{\delta_D^j}{\lambda_j}\right)} \end{cases} \tag{2.86}$$

Because increasing R creates a larger variation in the power target, QoS degradation increases with the increase of R. Therefore, when $\bar{P}$ is fixed, there exists a maximal value of R, beyond which no choice of weights w_j can meet the QoS constraints. When $\bar{P}$ and R are fixed, increasing w_j is always beneficial to type-j jobs because it allows that type of jobs to be processed with more servers. These observations intuitively explain the existence of an upper limit of n_σ and a lower limit of w_j in Eqs. (2.42) (2.43). Based on this, we design an algorithm to solve Eqs. (2.31∼2.41):

1. Start with a fixed value of $\bar{P}$.

2. Use binary search to find the maximal R that guarantees QoS. We start with an initial value for R, and use Eq. (2.43) to compute the minimal weights w_j^*. If the sum of these minimal weights, $\sum_{j=1}^{J} w_j^*$, is smaller than 1, then it means this value of R is small enough so that there exists a set of weights that meet the QoS constraints of the jobs, consequently we can increase R. On the other hand, if the sum $\sum_{j=1}^{J} w_j^*$ is already larger than 1, then it means no weights exist to meet the QoS constraints, and we should reduce the value of R. After several iterations, we will arrive at the maximal value of R. This maximal R also minimizes the cost function for this fixed $\bar{P}$ because of the negative term $-\Pi^R R$ in Eq. (2.31). Note that the third term in Eq. (2.31) related to tracking error is much smaller and also changes much slower when changing R, compared to the first two terms. With this set of $\bar{P}$, R, and w_j, we run one simulation to get the actual tracking error and use it to compute the cost in Eq. (2.31).

3. Loop through different values of $\bar{P}$ within an acceptable range of $\bar{P}$ derived from Eq. (2.40), and repeat the second step. By comparing the cost from different $\bar{P}$, we arrive at the global minimum and get the optimal choice of $\bar{P}$, R, and w_j. In practice, we apply this algorithm to get the optimal $\bar{P}$, R at the granularity of 1%.

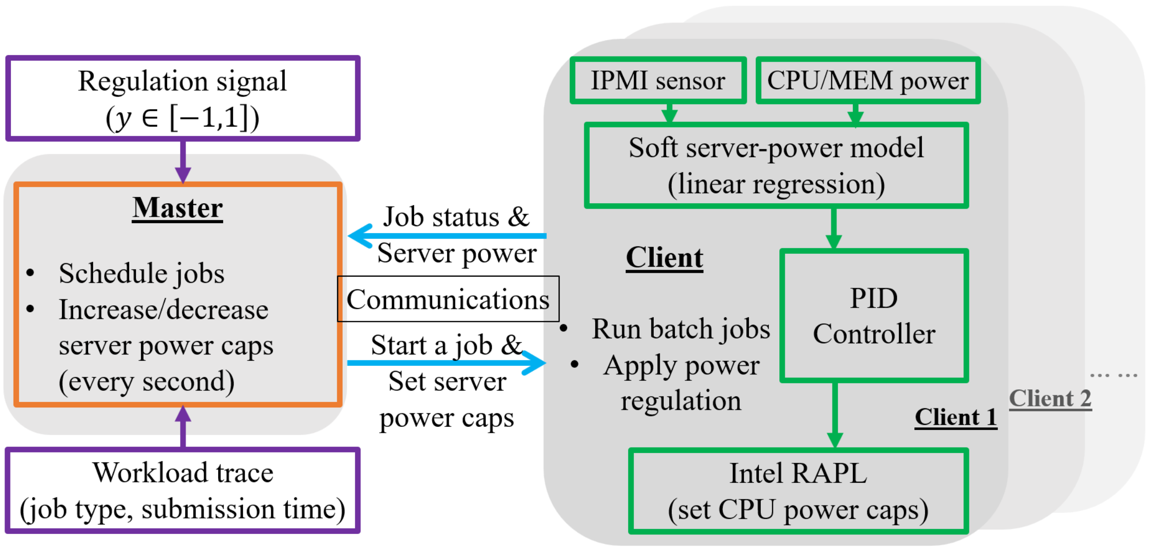


Figure 2·2: Real-system implementation architecture.

As we mentioned in Section 2.4.3, the coefficients α_j are determined empirically. Therefore, before we solve the optimization problem, we always run a simulation to determine the coefficients α_j by fitting the QoS distributions with Eq. (2.7).

2.4.6 Additional Methods to Regulate Power

In the sections above, we have explained how our policy regulates power consumption to match the target by job scheduling and server power capping. The flexibility of our policy allows additional methods to address potential non-ideal situations. Two typical non-ideal situations include:

Long execution time: When jobs with long execution times are running but the target power decreases sharply, the data center power may not be able to match the target because long-running jobs do not finish quickly and server power capping alone may not be able to reduce a large portion of power. To handle this situation, our policy can further apply *job preemption*, which interrupts the long-running jobs temporarily, and only resume their execution when target power matching is achievable. When applying job preemption, our policy prioritizes selecting jobs that are least possible to violate their QoS constraints, and the number of jobs preempted are determined by matching the power target. When the power target increases, resuming preempted jobs is prioritized before starting new jobs, and jobs closer to QoS violation are prioritized to resume.

Lack of jobs: When there are not enough jobs waiting in the queues but the target power is high, the data center power may not be able to match the target as servers are mostly in idle state due to this lack of jobs. To handle this situation, our policy can work with a queue of *standby jobs* and start jobs in this standby-job-queue. The standby-job-queue can be composed of jobs that have no QoS constraint or have a relatively loose QoS constraint at the time-scale of days. When other queues are empty, our policy starts jobs from this standby-job-queue, and the number of jobs to start is determined by matching the estimated total power consumption with the power target.

One real-life example for the standby-job-queue is the *overrun queue* in the Cori system at the National Energy Research Scientific Computing Center (NERSC) (National Energy Research Scientific Computing Center (NERSC), 2020). The overrun queue allows Cori users running out of their CPU-hour quota to submit jobs to this queue without monetary costs. Even though the overrun queue is convenient for the users, there is no guarantee on the waiting time in the overrun queue, and a job there may also be terminated by the system after running for 4 hours. Another similar example is the Amazon EC2 Spot Instances which offer spare computing capacity in the Amazon cloud at a steep discount (Amazon, 2020). Users can enjoy the low price of this service at the cost of unexpected interruptions: the cloud service provider may interrupt the service with a warning two minutes in advance whenever there are no sufficient spare servers.

Table 2.2: Characteristics of benchmark applications.

Application Name	Max Processing Time (T^j_{max})	Min Processing Time (T^j_{min})	Max Power ($p_{j,max}$)	Min Power ($p_{j,min}$)
`fs`	100.8 s	97.6 s	142 W	137 W
`sc`	53.8 s	52.6 s	221 W	218 W
`bt`	143.0 s	108.5 s	279 W	241 W
`lu`	85.3 s	62.8 s	262 W	231 W
`mg`	161.2 s	132.6 s	309 W	261 W
`sp`	108.4 s	99.9 s	290 W	260 W
`ft`	35.2 s	24.9 s	281 W	229 W

2.5 Simulation and MOC Experiments of QoSG policy

To evaluate our QoSG policy, we run simulations at various scales and conduct experiments on a real cluster composed of 12 servers from Massachusetts Open Cloud (MOC).

2.5.1 Simulation and Experiments Setup

In both simulations and real-system experiments, we use computing workloads composed of several types of jobs. These jobs are applications from NAS Parallel Benchmark (NPB) (Bailey et al., 1991) and PARSEC Benchmark (Christian, 2011) suites, which are good representatives of real applications running on high-performance computing data centers. Table 2.2 summarizes the characteristics of the benchmark applications collected by running them on our servers. Among them, `bt`, `lu`, `mg`, `sp`, `ft` are from the NPB. `fs` and `sc` are abbreviations for applications `facesim` and `streamcluster` from the PARSEC suite.

We randomly create workload traces composed of these benchmark applications for our experiments. For each type-j job, its arrival times are generated as a Poisson process with an arrival rate λ_j. This arrival rate is determined by the data center utilization level $\eta \in [0, 1]$, which represents the average ratio of active servers in the data center. By default, we assume different types of jobs will occupy the data center equally on average, so λ_j is determined by:

$$\lambda_j T_{min}^j = \frac{\eta N}{J} \quad (j = 1, 2, ..., J). \tag{2.87}$$

To compare with our new policy, we implement two policies proposed in previous work, the *Tracking-only* policy (Chen et al., 2014) and the *EnergyQARE* policy (Chen et al., 2019), as baselines.

The *Tracking-only* policy uses job scheduling and processor power-capping to control a data center's power to follow the power target. It only focuses on target tracking, ignorant of the QoS of jobs. This policy determines $\bar{P}$ and R heuristically. It selects a $\bar{P}$ equal to the estimated average power of the data center according to its utilization level, i.e., $\bar{P} = \eta N \sum_{j=1}^{J} p_j / J + (1 - \eta) N p_{idle}$. And R is selected so that the minimal (and maximal) possible power targets, $\bar{P} - R$ (and $\bar{P} + R$), are achievable. Thus, $R = \min\{\bar{P} - N \cdot p_{idle}, N \cdot \max_j p_j - \bar{P}\}$.

EnergyQARE monitors the tracking error and jobs' QoS degradation at every moment. Based on whether the tracking error or the QoS degradation at this moment is larger, it either focuses entirely on tracking more accurately without concerning jobs' QoS, or tries to reduce QoS degradation by running more jobs without concerning the tracking error. The *EnergyQARE* policy runs simulations with each pair of $(\bar{P}, R)$ and selects the best $\bar{P}$ and R that minimize the cost under the tracking error constraint.

These two baselines assume the idle servers can transition into sleeping states to further reduce power. Since the servers in our real system do not support sleeping states, we implement the baselines without server state transition.

In the following, we describe our simulation method and our real-system implementation.

Simulation: We implement a simulator in Python. The simulator creates a cluster of servers and keeps track of the servers' power usage. For each server, we assume an idle power of $p_{idle} = 90$ W, and we assume the power usage of jobs follows the values in Table 2.2. When our policy applies a power cap on a server that is running a type-j job, we assume the processing time of the job increases linearly. That is, when the power cap decreases from $p_{j,max}$ to $p_{j,min}$, the processing time will linearly increase from T^j_{min} to T^j_{max}.

Real-System Implementation: The cluster used in our experiments is composed of 12 Dell PowerEdge M620 blade servers. Each server has two Intel Xeon E5-2650 v2 processors. Each server consumes an idle power of $p_{idle} = 90$ W, and its power can rise to more than 300 W when running certain applications.

For these servers, we monitor their processor and memory power using the *perf* utility (Melo and Carvalho, 2010), and we monitor the total power of a server using IPMI tool (Laurie et al., 2018). However, the IPMI tool on these servers gives a running average value at a granularity of 4 Watt, which is too coarse to serve our purpose. Therefore, we build a power model to get the server power with higher granularity. This power model takes the processor power P_{proc} and the memory power P_{mem} as input, and gives the total server power P_{server} as output, following a linear relation: $P_{server} = \phi_1 P_{proc} + \phi_2 P_{mem} + \phi_3$.

This power model inherently considers the power contributed by various components in a server, including fans, hard drives, motherboards, etc. We run each benchmark application several times and collect their processor/memory/server power readings, and fit them with the power model. For our servers, the fitting provides the following model parameters: $\phi_1 = 1.35$, $\phi_2 = 1.32$, $\phi_3 = 53.4$ W, and this model gives a server power reading at high granularity with an error of less than 2%.

To apply a power cap on a server, i.e., to add a power limit that the server cannot exceed, we use the Intel Running Average Power Limit (RAPL) tool (David et al.,

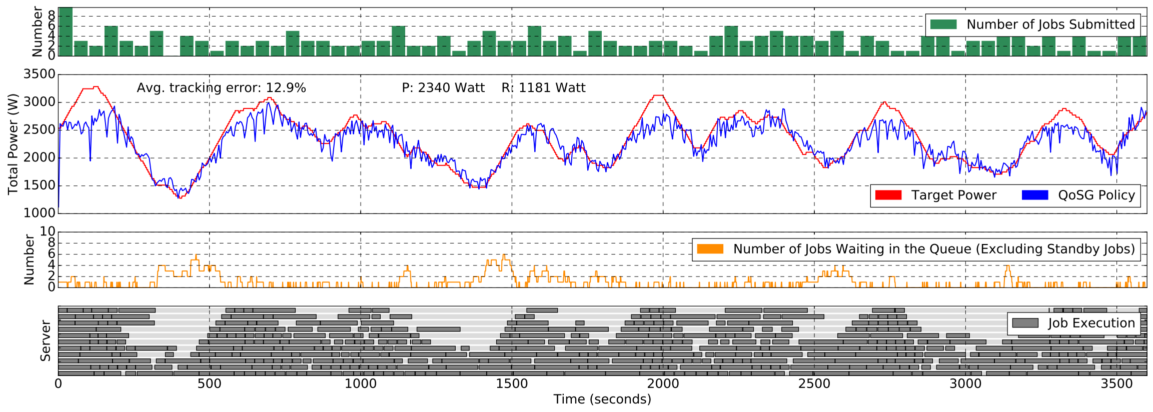


Figure 2·3: Experiments on a real 12-server cluster using our QoSG policy.

2010). As RAPL can apply power caps on processors and cannot directly control the total server power, we build a PID controller to apply power caps on servers. Given a server power cap, the PID controller recurrently adjusts the processor power caps by RAPL to let the server power match the cap. On our servers, we explore different parameters for the PID controller and the choice $P = 0.525$, $I = 0$, $D = 0$ works well.

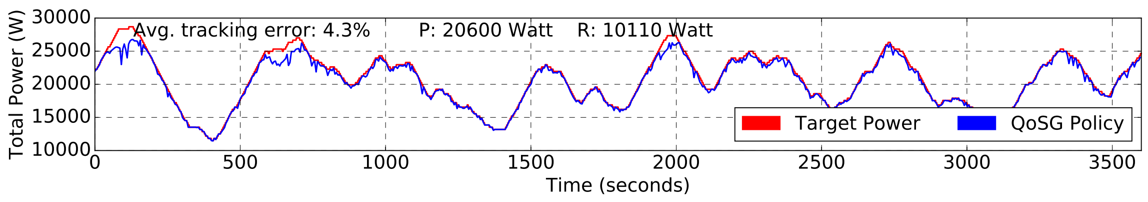
To experiment on this cluster, we let one server be a "master" node to schedule jobs and determine power caps following our QoSG policy. The other servers work as "client" nodes to follow the order of the master and run jobs. Every second, the master node sends a message about the scheduling decision to each client and receives a message about their status from each client using the *rabbitmq* (Chapman et al., 2018) tool. The architecture of this system is shown in Fig. 2·2.

2.5.2 Evaluation with the Default Setting

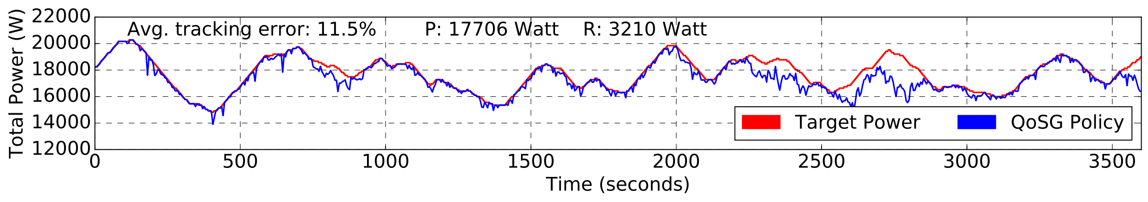
We evaluate our QoSG policy through both simulations and real-system experiments. In our default setting, we use a data center with $N = 100$ servers (when in simulations), or 12 servers (when in real-system experiments). We assume the workload arrivals follow Poisson processes that maintain an average data center utilization level of $\eta = 50\%$. When using our QoSG policy, by default we let one type of job, `ft`, to be

Table 2.3: Thresholds Q^j_{thres} in the QoS constraints of jobs.

QoS Constraint Level	`fs`	`sc`	`bt`	`lu`	`mg`	`sp`
Tight	0.65	0.60	0.55	0.45	0.40	0.35
Medium	1.3	1.2	1.1	0.9	0.8	0.7
Loose	2.6	2.4	2.2	1.8	1.6	1.4



(a) With *standby jobs*.



(b) Without *standby jobs*.

Figure 2·4: Simulation results for a 100-server data center participating in demand response using our QoSG policy. The actual power consumption (blue) of the data center follows the target power (red) closely, with a 4.3% average tracking error.

the *standby jobs*, and we assign some type-specific QoS constraints in Table 2.3 for the other 6 types of jobs. By default, we use the Medium-level QoS constraints. When estimating the electricity cost using Eq. (2.3), we assume $\Pi^P = \Pi^R = \Pi^\epsilon = 0.1\$/\text{kWh}$. In our experiments, we use a historical trace of an ISO signal from PJM. The signal is updated every 4 seconds, and we run each experiment with a length of one hour.

Fig. 2·4(a) shows the result from a one-hour simulation when using the default setting. This simulation uses the optimal values of the average power $\bar{P}$, reserves R, and weights w_j:

$$\bar{P} = 20600\text{ W},\ R = 10111\text{ W},\ w_{fs} = 12.8\%,\ w_{sc} = 21.2\%,$$

$$w_{bt} = 12.7\%,\ w_{lu} = 27.1\%,\ w_{mg} = 12.4\%,\ w_{sp} = 14.0\%.$$

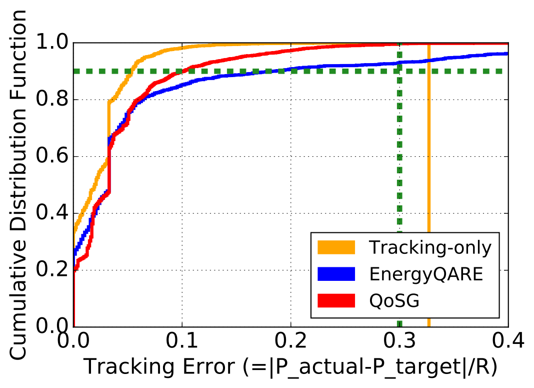


Figure 2·5: CDFs of tracking error in simulation with the default setting.

These optimal values are derived following the method in Section 2.4.5. Among all weights, w_{sc} and w_{lu} are larger than the others, because these two types of jobs have tighter absolute tolerance on the sojourn time, computed by $(1 + Q^j)T^j_{min}$.

In Fig. 2·4(a), the red curve is the target power; the blue curve is the power consumption when applying our policy. The actual power follows the target very well, with an average tracking error of 4.3%. The electricity cost for this hour is \$3931. If not participating in RSRs, the electricity cost will be \$7292. Therefore, our policy reduces the cost by 46.1%. The cumulative distribution function (CDF) of tracking error in Fig. 2·5 show that our policy meets the tracking error constraint in Eq. (2.4). Fig. 2·6 also show the CDF of job QoS degradation, which meets the QoS constraint in Eq. (2.5).

Note that the use of *standby jobs* is not the main reason for the 46.1% cost reduction, because the energy consumed by *standby jobs* only occupies 14.2% of all energy consumed in this hour. When there are no *standby jobs* at all, data centers can still apply our policy, and Fig. 2·4(b) shows the simulation result. In this case, the cost reduction drops to 14.4%. That is because a smaller $\bar{P}$ is needed to guarantee good signal tracking, and a smaller R is also needed to reduce the power target's variation so as to guarantee job QoS.

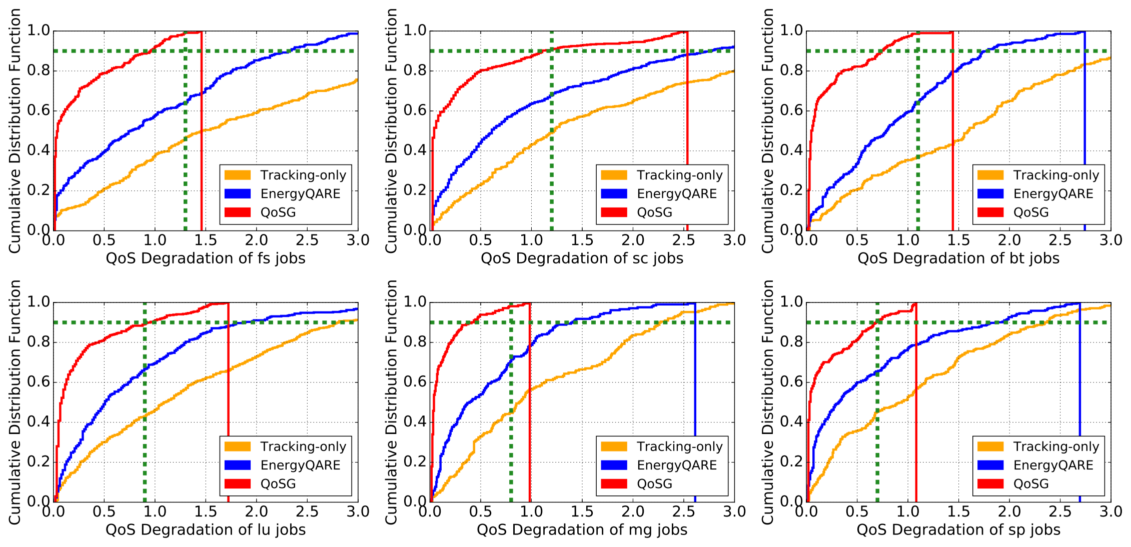
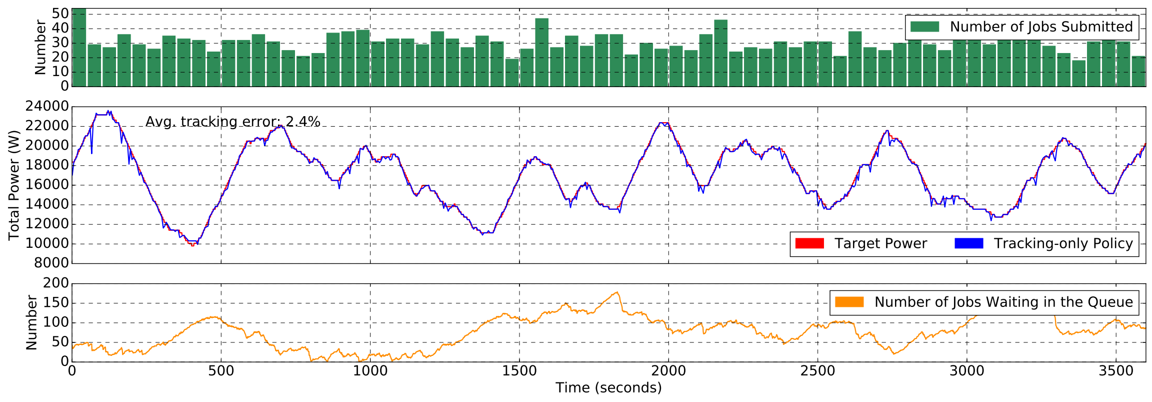
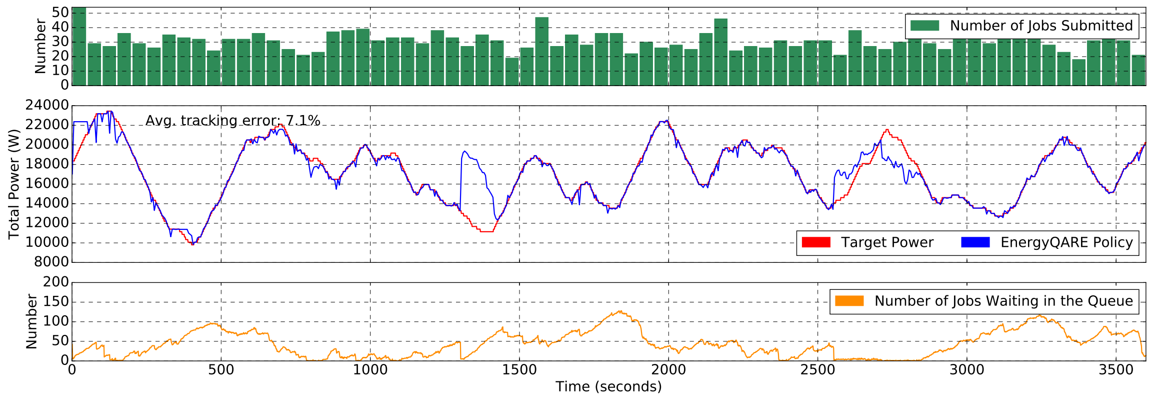


Figure 2·6: Cumulative distribution functions of the QoS degradation of each type of jobs in simulation. This simulation uses the default setting with 100 servers. The green dashed lines are the thresholds in the QoS constraints in Eq. (2.5) and Table 2.3. Our QoSG policy guarantees that all job types meet their constraints. The *Tracking-only* policy and the *EnergyQARE* policy violate the QoS constraints in this case.

Fig. 2·5 also compares our QoSG policy (with *standby jobs*) with the two baselines: the *Tracking-only* policy and the *EnergyQARE* policy. We compare their CDFs for the tracking error and `sp` jobs' QoS degradation. Because the *Tracking-only* policy only focuses on tracking and does not consider job QoS, it leads to the smallest tracking error but the largest QoS degradation. The *EnergyQARE* policy balances tracking and QoS, so it reduces the QoS degradation compared to the *Tracking-only* policy. However, as their policy is designed heuristically without any theoretically-proven guarantees, QoS degradation still violates the constraints in this case. Even if we enhance the *EnergyQARE* policy by allowing it to have a queue of *standby jobs*, the QoS constraints of some job types are still violated because that policy does not assign optimally-determined weights to different queues and, thus, cannot balance different types of jobs well. The power-time curves of these baseline policies are shown in Fig. 2·7.



(a) The *Tracking-only* policy



(b) The *EnergyQARE* policy

Figure 2·7: Power-time curves of the simulations using the two baseline policies with the default setting. The *Tracking-only* policy provides good tracking performance, with an average tracking error of 2.4%. The *EnergyQARE* policy consumes more power than the target at some places (e.g., $t = 1300$) in order to reduce the QoS degradation of the jobs. Therefore, *EnergyQARE* leads to smaller number of jobs waiting in the queues, at the cost of larger tracking error than *Tracking-only*.

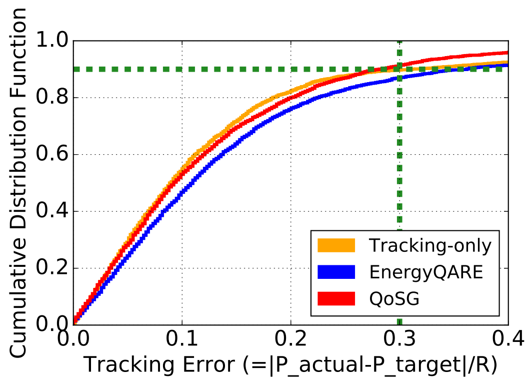


Figure 2·8: CDFs of tracking error in real experiments with the default setting.

We evaluate our proposed policy and the two baselines on a real cluster using the default setting. The cluster has 12 servers including 1 “master” node and 11 “client” nodes. The results using our QoSG policy are shown in Fig. 2·3. The number of jobs submitted to the data center in each time interval is displayed as the green bars. The grey rectangles represent the execution (from start to end) of a job on a certain server. We see that, even with this small cluster, our QoSG policy enables this cluster to participate in RSRs and follow the power target well. The yellow curve shows the number of jobs (excluding *standby jobs*) waiting in the queues at every moment. When the target power drops (e.g., at $t = 400$ s), the number of active servers is reduced in order to track the target, so the number of waiting jobs increases. At a few places (e.g., at $t = 100$), the real power does not reach the target. That is because at these moments, all servers are running, so the total power cannot be further increased by starting more jobs. Since this mismatch seldom happens, our policy still meets the tracking error constraint.

For these real-system experiments, Fig. 2·8 shows the CDFs of the tracking error and Fig. 2·9 shows the QoS degradation of jobs. They both meet the constraints for QoSG, but QoS constraints are not met by using the other policies. The cost reduction using the QoSG policy is 42%, which is 38% when using *Tracking-only*, and

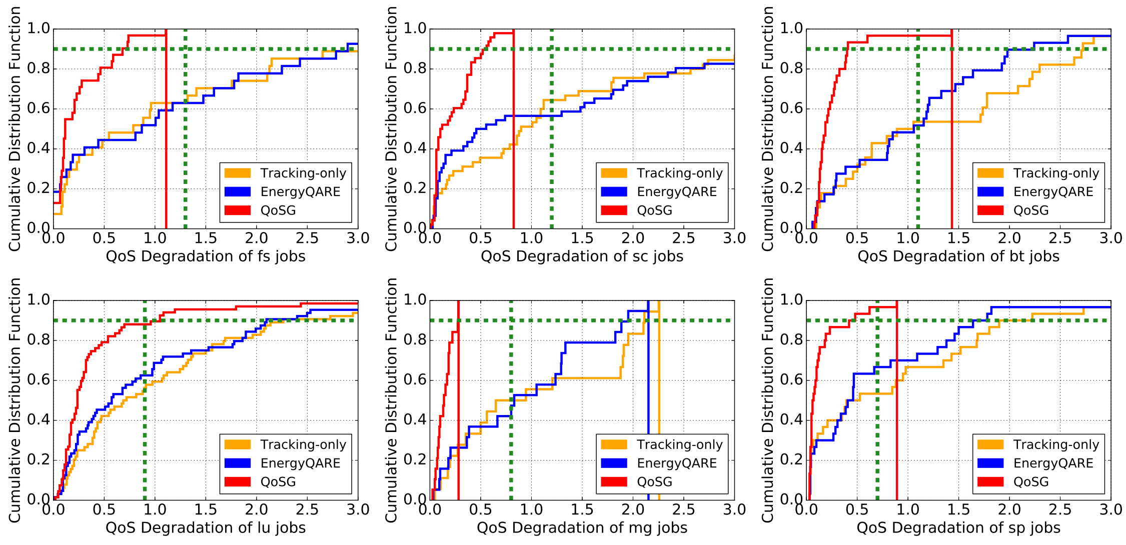


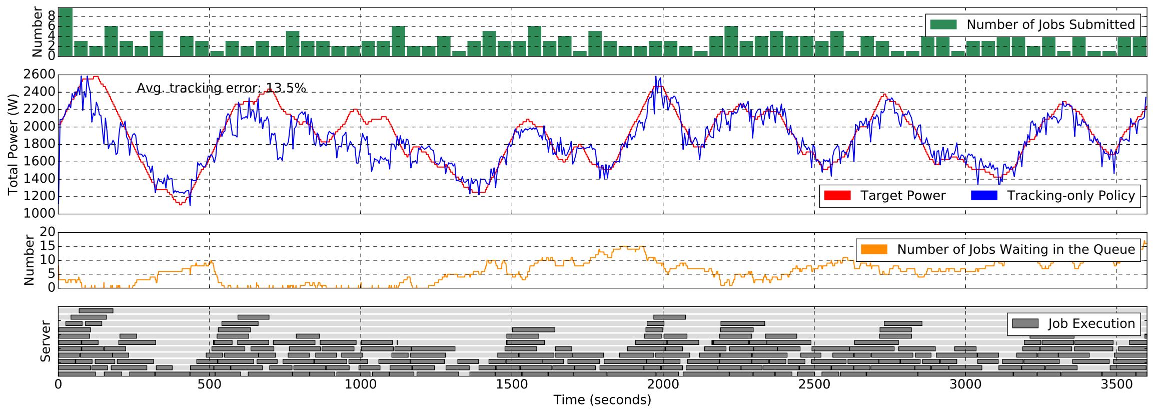
Figure 2·9: Cumulative distribution functions of the QoS degradation of each type of jobs in a real-system experiment with 12 servers. Our QoSG policy guarantees that all job types meet their constraints; meanwhile, the baseline policies cannot.

37% when using *EnergyQARE*. Compared to simulation, real-system experiments show higher tracking error because, in reality there are variations in the power usage of a job, especially at a job's beginning and completion stage. Fig. 2·10 shows the power-time curves for the baselines. We also simulate the experiment with 12 servers, and the difference between the power consumption in our simulator and that in the real-system is only 5%, which indicates a reasonable accuracy for our simulator.

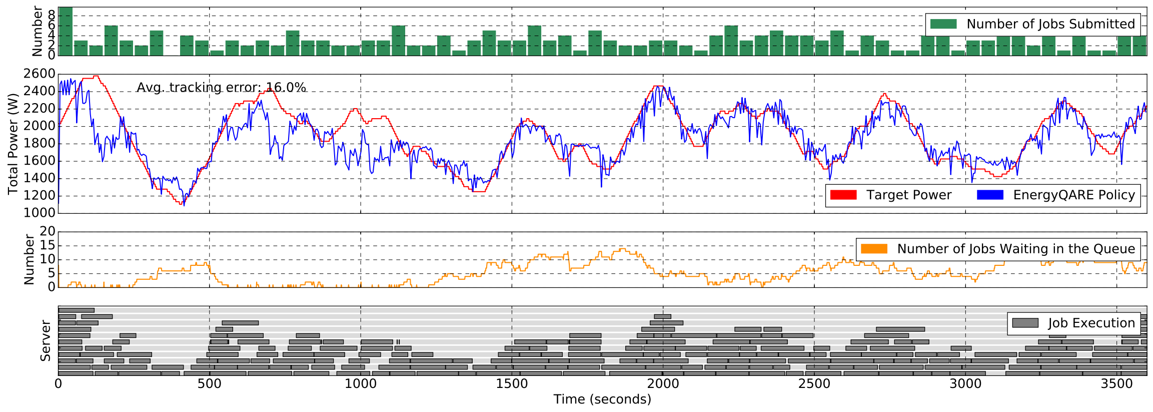
2.5.3 Comparison of Different Settings

Using simulations, we evaluate the QoSG policy with different data center sizes. Fig. 2·11 shows the optimal $\bar{P}$, R values, the cost reduction, and the ratio of energy consumed by *standby jobs*. As we increase the size from 20 to 2500, the cost reduction remains above 45%, which shows our policy scales well with data center size.

Fig. 2·12 compares the results when using different QoS constraint levels in Table 2.3. Because tight QoS constraints are easier to be violated, more conservative selection of $\bar{P}$ and R is required. Therefore, in Fig. 2·12, tighter QoS constraints lead



(a) The *Tracking-only* policy



(b) The *EnergyQARE* policy

Figure 2·10: Power-time curves of the real-system experiments using the two baseline policies with the default setting.

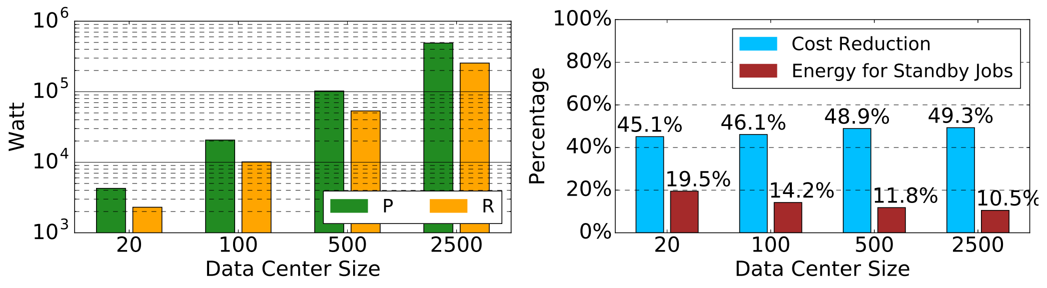


Figure 2·11: Results for different data center sizes.

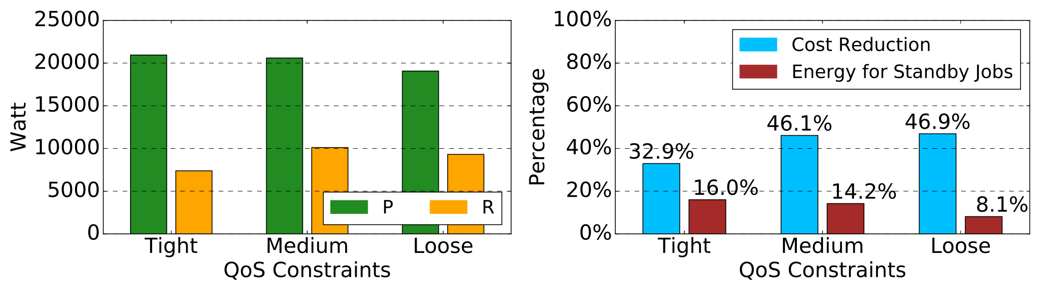


Figure 2·12: Results for different QoS constraint levels.

to larger $\bar{P}$, smaller cost reduction, and more *standby jobs*.

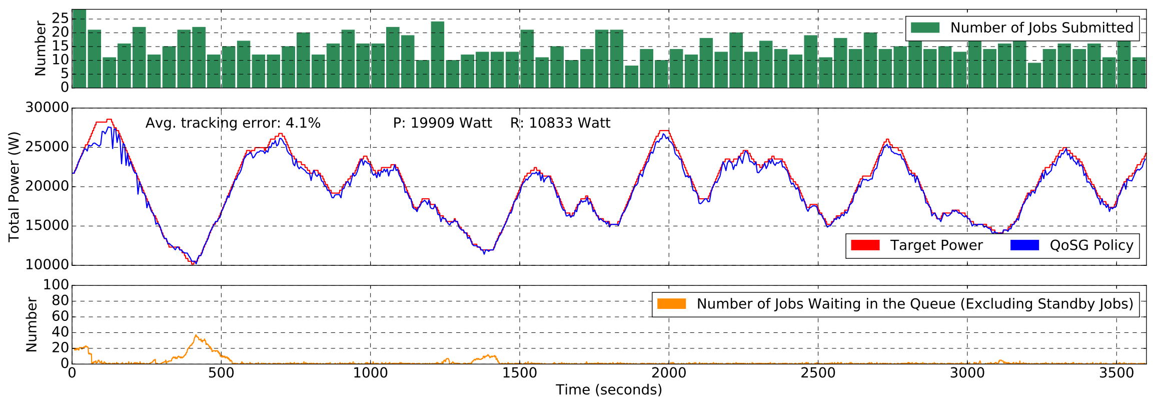
We evaluate the influence of data center utilization level by simulations with the utilization level $\eta = 25\%$, 50% (the default), or 75%. Fig. 2·14 shows the optimal $\bar{P}$, R, the cost reduction, and the percentage of energy consumed by *standby jobs*. Their power-time curves are in Fig. 2·13. As the utilization level increases, the flexibility of data centers in power regulation decreases because more power becomes necessary in order to guarantee QoS. Therefore, we see the cost reduction drops to 25.1% when $\eta = 75\%$.

In previous experiments, we assume different types of jobs are balanced in the workload, i.e., they occupy the data center equally on average (see Eq. (2.87)). We evaluate the impact of this factor by comparing two unbalanced workload traces: in trace W1, the jobs with longer processing time (`bt`, `mg`, `sp`) have higher occupancy; in trace W3, the jobs with shorter processing time (`fs`, `sc`, `lu`) have higher occupancy. W2 is the default balanced trace. To be specific, the job arrival rates in these traces satisfy:

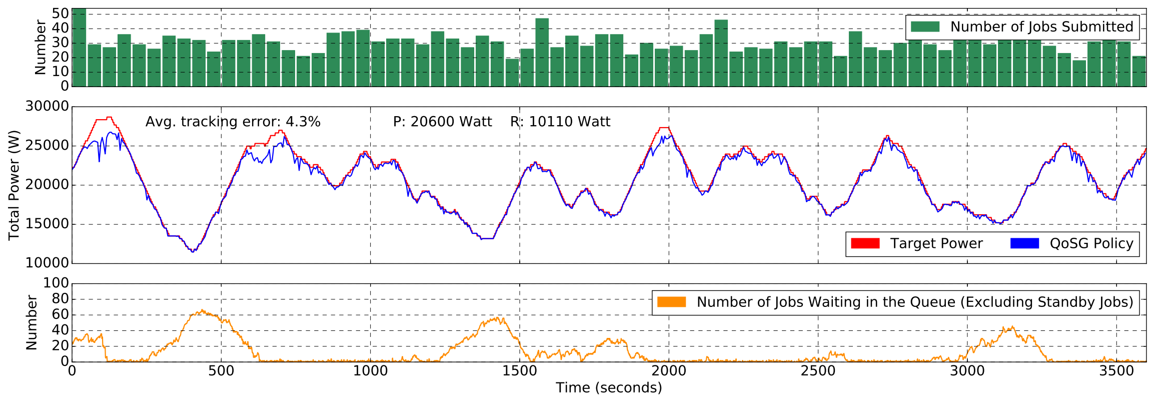
$$[\mathrm{W1}] \quad \lambda_1 T_1 : \lambda_2 T_2 : \lambda_3 T_3 : \lambda_4 T_4 : \lambda_5 T_5 : \lambda_6 T_6 = 1:1:3:1:3:3$$

$$[\mathrm{W2}] \quad \lambda_1 T_1 : \lambda_2 T_2 : \lambda_3 T_3 : \lambda_4 T_4 : \lambda_5 T_5 : \lambda_6 T_6 = 1:1:1:1:1:1$$

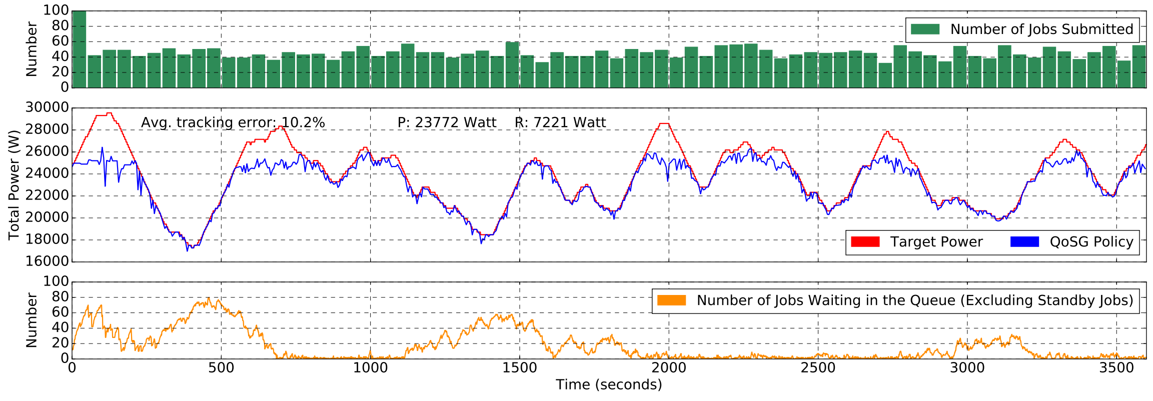
$$[\mathrm{W3}] \quad \lambda_1 T_1 : \lambda_2 T_2 : \lambda_3 T_3 : \lambda_4 T_4 : \lambda_5 T_5 : \lambda_6 T_6 = 3:3:1:3:1:1$$



(a) At 25% utilization level



(b) At 50% utilization level



(c) At 75% utilization level

Figure 2·13: Power-time curves for simulations using our *QoSG* policy at different data center utilization levels. Our policy selects different optimal values for $\bar{P}$ and R according the utilization level.

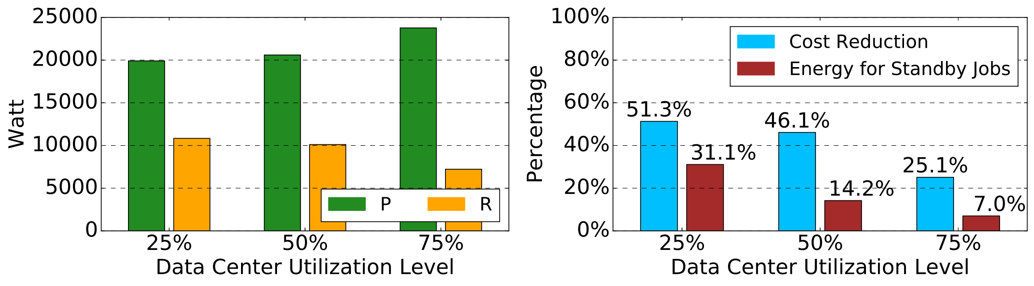


Figure 2·14: Results for different data center utilization levels.

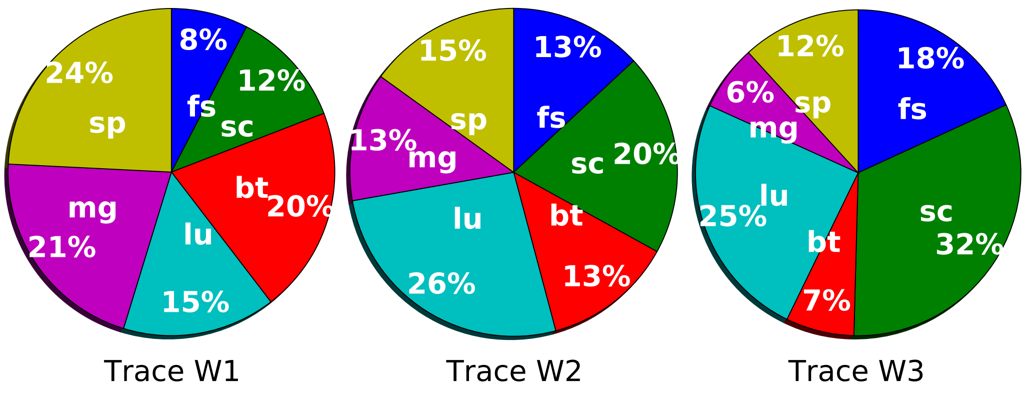


Figure 2·15: Optimal weights w_j in traces W1, W2, W3.

Here, indices 1 to 6 refer to `fs`, `sc`, `bt`, `lu`, `mg`, `sp`. Fig. 2·15 shows the optimal weights w_j when using these workload traces. We see that job types with higher occupancy tend to have higher weights. For example, from W1 to W3, the occupancy of `fs` jobs increases, so their weight increases from 8% to 18%.

We further consider the situation when there are only two types of jobs (including one type of *standby jobs*). Fig. 2·16 compares the simulation results for workloads of different compositions. Because the power usage of `fs` jobs is the smallest among all jobs in Table 2.2, there is not much room for power regulation, so the cost reduction in this case is only 21.3%. On the other hand, for `bt` jobs whose power usage is much larger, the cost reduction becomes 39.8%.

In all the results above, we use the same one-hour ISO signals from PJM. To check the robustness against different signals, in Fig. 2·17, we show simulation results (with

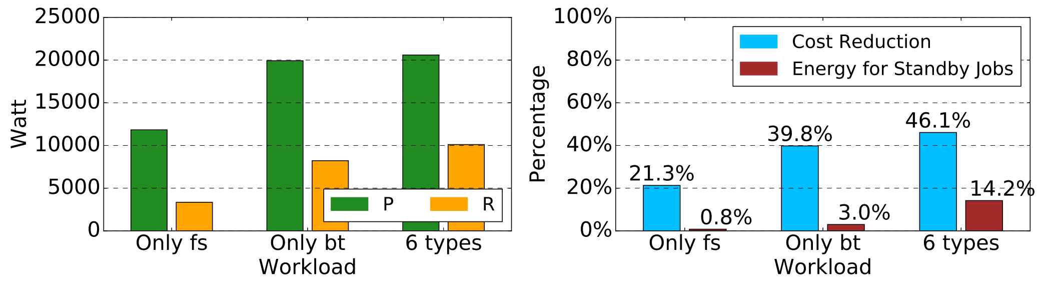


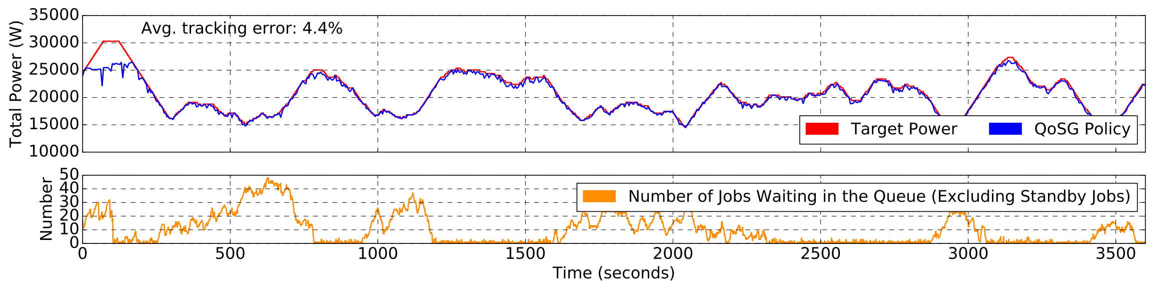
Figure 2·16: Comparing workloads of different compositions.

the same parameters as Fig. 2·4) using 4 different one-hour samples of the PJM ISO signals. The actual power matches the target well in general. We have verified that different signals lead to similar tracking error, QoS degradation, and cost reduction.

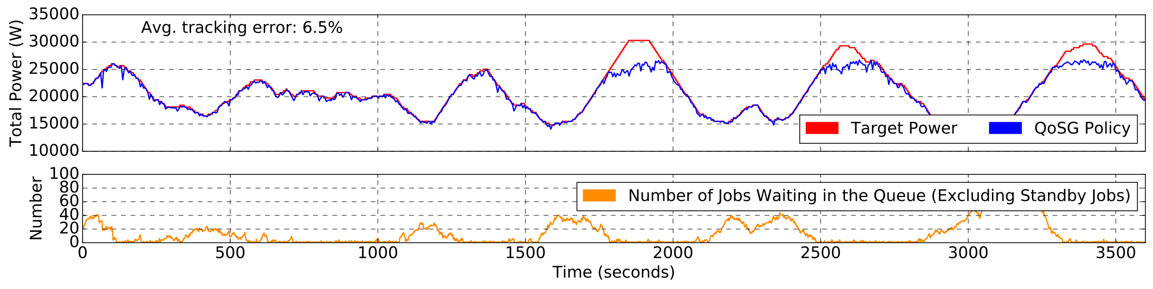
2.6 The Adaptive QoS-Assurance (AQA) policy

The work with QoSG policy suffers from the following problems: (1) The policy only applies to single-node jobs. (2) Some empirical paramters used in the policy to guarantee job QoS are fit from a single run of simulation, which may suffer from empirical errors. (3) The experiment is done on a small cluster with a limited number of workloads. To solve these problems, we further propose the Adaptive-QoS-Assurance (AQA) policy. The AQA policy has the following improvements: (1) Instead of considering only single-node jobs, the AQA policy now allows parallel applications that run on multiple nodes. (2) The AQA policy adaptively optimizes the bidding parameters and the weight parameters used by the policy, in contrast to estimating the optimal parameters based on a single simulation. (3) Compared to our earlier experiments on a 12-server cluster with a limited number of workloads, the following work includes real-system experiments on a 36-server cluster with a broad set of workload settings.

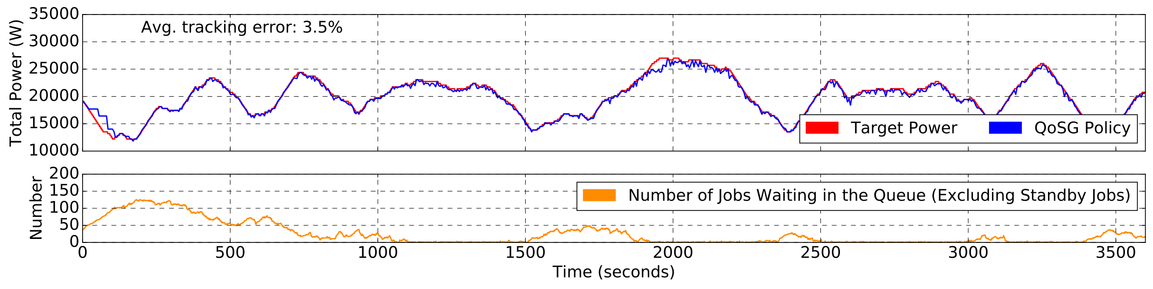
In this section, we first introduce our data center model and give an overview of our AQA policy. Next, we explain how our AQA policy regulates the total power



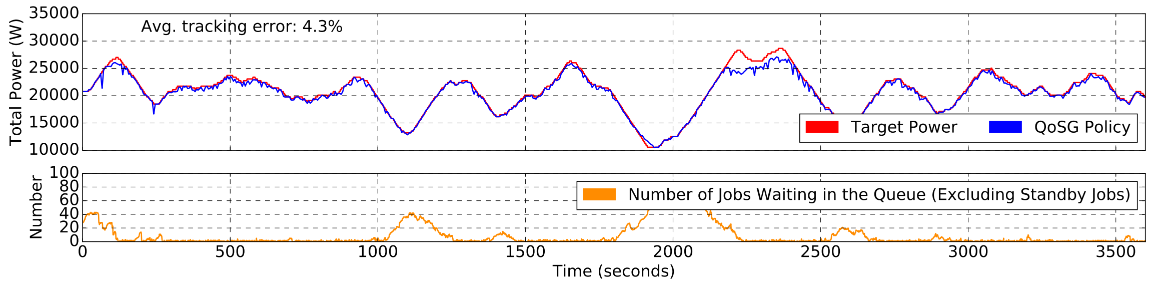
(a) ISO signal sample 1



(b) ISO signal sample 2



(c) ISO signal sample 3



(d) ISO signal sample 4

Figure 2·17: Results using different one-hour samples of the ISO signal. These results demonstrate that our result is robust to the behavior of ISO signal. In all these simulations, the average tracking error is less than 7%. Our *QoSG* policy meets the tracking error constraint and the job QoS constraints.

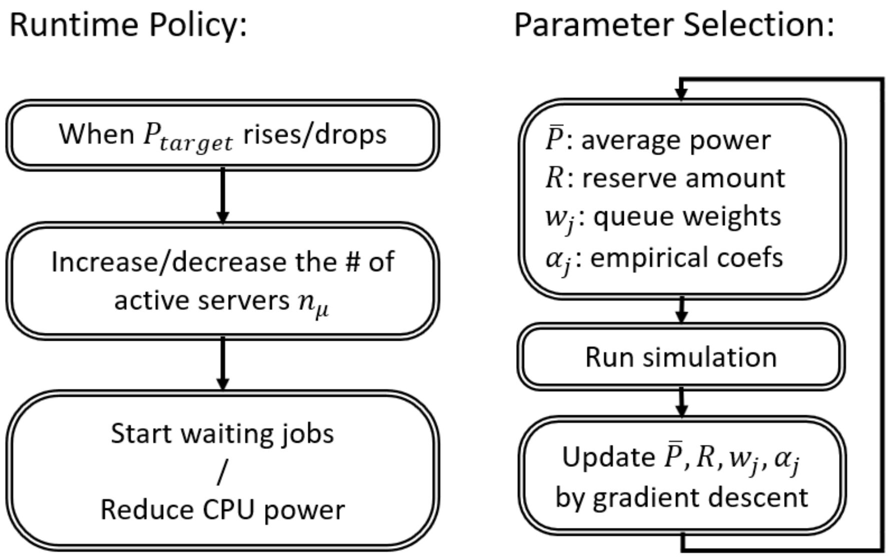

Runtime Policy:
When P_{target} rises/drops
Increase/decrease the # of active servers n_μ
Start waiting jobs / Reduce CPU power
Parameter Selection:
$\bar{P}$: average power
R: reserve amount
w_j: queue weights
α_j: empirical coefs
Run simulation
Update $\bar{P}, R, w_j, \alpha_j$ by gradient descent

Figure 2·18: The runtime part and the parameter selection part of our AQA policy.

to track the target by job scheduling and server power capping. Then, we explain how our policy adaptively finds the optimal bidding parameters ($\bar{P}$, R) and weight parameters (w_j), when participating in the regulation service markets.

2.6.1 An Overview of the AQA Policy

To provide QoS guarantees to each type of job, we partition the active servers to job types following the Generalized Processor Sharing (GPS) algorithm (Parekh and Gallager, 1993), discussed in Section 2.4.2. According to the algorithm, the number of active servers allocated for each job type is proportional to a set of non-negative weights, w_j (with $\sum_{j=1}^{J} w_j = 1$). As we follow the GPS algorithm, a queueing-theoretic result guarantees that the delay in each queue meets QoS constraints (Paschalidis, 1999; Bertsimas et al., 1999).

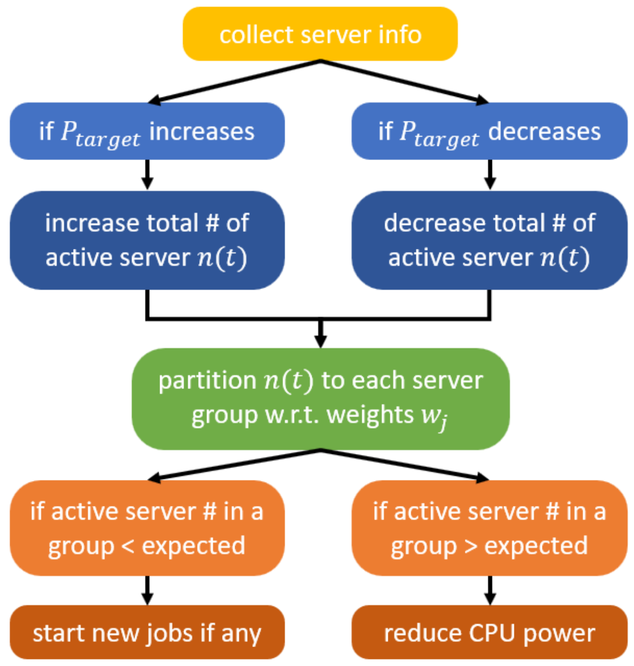
When applying our policy at runtime, at the beginning of every cycle (one cycle is one second in our experiments), the policy adjusts the total number of servers expected to be active in order to match the total power consumption with the target power. Next, the number of servers expected to be active for each job type is determined following the GPS algorithm. Then, for each job type, if the number of active servers in this group needs to increase to meet the expectation, our policy will activate idle servers to run some queued jobs of this type if there are any. On the other hand, if the number of active servers in this group needs to decrease to meet the expectation, our policy will reduce these servers' power cap instead of deactivating them because we assume no job interruption. These procedures form the runtime policy in Fig. 2·18.

The key to guarantee QoS and simultaneously reduce monetary cost is to select optimal bidding parameters ($\bar{P}$, R) and weight parameters (w_j). Although bidding for a larger average power $\bar{P}$ is more beneficial to guarantee QoS because a higher power target (as a result of a larger $\bar{P}$) allows more servers to run, larger $\bar{P}$ also increases the monetary cost. The weight parameters (w_j) need to be well-tuned so that a job type that is harder to meet its QoS constraint will take a larger weight and consequently, be able to access more servers. Our policy determines the optimal parameters by running simulations and applying the gradient descent on a cost function. That cost function includes both the monetary cost and an additional term penalizing QoS violation. These procedures are depicted in Fig. 2·18.

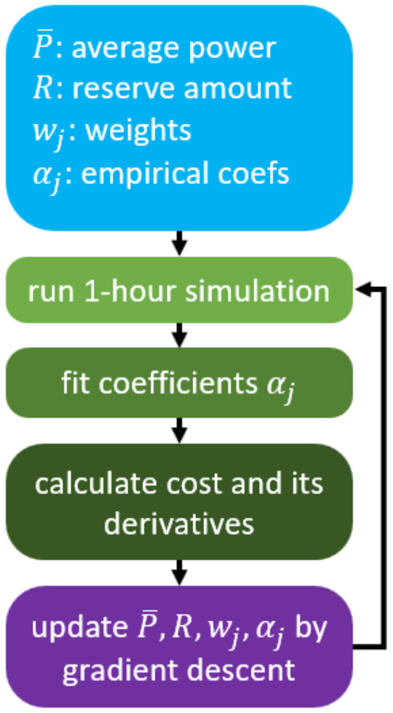
2.6.2 Job Scheduling and Power Capping in AQA

Our AQA policy adjusts a data center's power consumption at runtime to match the power target P_{target} by job scheduling and server power capping strategies, as shown in Fig. 2·19(a). The parameters used in the following are listed in Table 2.1.

In order to provide guarantees on job QoS, we schedule J different queues of jobs following the GPS algorithm. To match power target $P_{target}(t)$ at time t, our policy first determines the total number of active servers $n(t)$. Then, the $n(t)$ active servers are partitioned for the J queues according to their weights w_j following the GPS


collect server info
if P_{target} increases
if P_{target} decreases
increase total # of active server $n(t)$
decrease total # of active server $n(t)$
partition $n(t)$ to each server group w.r.t. weights w_j
if active server # in a group < expected
if active server # in a group > expected
start new jobs if any
reduce CPU power

(a) Job scheduling and server power capping at runtime.


$\bar{P}$: average power
R: reserve amount
w_j: weights
α_j: empirical coefs
run 1-hour simulation
fit coefficients α_j
calculate cost and its derivatives
update $\bar{P}, R, w_j, \alpha_j$ by gradient descent

(b) Optimal selection of bidding and weight parameters.

Figure 2·19: The two components of our AQA policy.

algorithm. As a result, we get the number of active servers for the j^{th} queue (i.e., for j^{th}-type jobs), n_j. This n_j equals to $n(t)w_j$ if all queues are non-empty, and larger if not. Next, from n_j, we get the number of j^{th}-type jobs that should be running as n_j/m_j, where m_j is the number of servers required for running each j^{th}-type job[3].

If n_j/m_j is smaller than the current number of running jobs of the j^{th} type, we deduct n_j/m_j from the number of j^{th}-type jobs that are running, and we arrive at the number of j^{th}-type jobs that should be scheduled to start at this moment. On the other hand, if n_j/m_j is larger than the current number of running jobs of the j^{th} type, since we do not want to terminate jobs before they finish, we reduce data center power by reducing server power caps of all active servers, which is discussed in the following paragraphs. Whenever reducing the server power caps is not necessary, our policy always let servers run without power caps.

From the policy described above, we see that the total number of active servers $n(t)$ at time t should be determined by matching the target power with the data center's power consumption, i.e.:

$$P_{target} = \bar{P} + y(t)R = (N - n(t))p_{idle} + \sum_{j=1}^{J} n_j p_j,$$

which is equivalent to

$$n(t) = \frac{\bar{P} + y(t)R - p_{idle}N}{\left(\sum_{j=1}^{J} w_j p_j\right) - p_{idle}}. \tag{2.88}$$

Here, N is the total number of servers in the data center. p_{idle} is the idle server power, and p_j (also denoted as $p_{j,max}$ later) is the power for running a j^{th}-type job without power capping. We have also made an approximation $n_j = n(t)w_j$.

Our AQA policy applies server power capping only in situations when the already-running jobs are consuming more power than the target. When that happens, our policy reduces the power cap on all the active servers by the same ratio $\omega \in [0, 1]$ to

[3]We call a job as a single-server job if $m_j = 1$, and a multi-server job or parallel job if $m_j > 1$.

ensure fairness. To be more specific, for a j^{th}-type job whose power consumption is $p_{j,max}$ and $p_{j,min}$ when under the highest/lowest server power cap, we apply a server power cap that makes the job running at power $p_{j,cap}$ defined by the equation

$$\omega = \frac{p_{j,cap} - p_{j,min}}{p_{j,max} - p_{j,min}}.$$

The ratio ω is determined by matching the target power and the actual consumption:

$$P_{target} = (N - n(t))p_{idle} + \sum_{j=1}^{J} n_j p_{j,cap}.$$

2.6.3 Bidding and Weight Parameter Selection in AQA

Our policy determines the optimal selection of bidding parameters $\bar{P}$, R, and weights w_j by solving the following optimization problem:

$$\min_{\bar{P},R,w_j} \left(\Pi^P \bar{P} - \Pi^R R + \Pi^\epsilon R \bar{\epsilon}\right) \times 1\text{h} \tag{2.89}$$

$$\text{subject to} \quad \text{Prob}[Q^j \geq Q^j_{thres}] \leq \delta^j, \quad j = 1, 2, ... \tag{2.90}$$

$$\sum_{j=1}^{J} w_j = 1, \quad \bar{P}, R, w_j > 0. \tag{2.91}$$

We first simplify the QoS constraints in Eq. (2.90) using the queueing-theoretical result in Sec. 2.4.3:

$$\text{Prob}[Q^j \geq Q^j_{thres}] \leq \delta^j \tag{2.92}$$

$$\Leftrightarrow \quad \text{Prob}\left[\frac{T^j_{wait} + T^j_{proc} - T^j_{min}}{T^j_{min}} \geq Q^j_{thres}\right] \leq \delta^j \tag{2.93}$$

$$\Leftrightarrow \quad \text{Prob}[T^j_{wait} \geq Q^j_{thres} T^j_{min}] \leq \delta^j. \tag{2.94}$$

Here, Eq. (2.93) is transformed into Eq. (2.94) by approximation since the actual processing time T^j_{proc} is usually close to the minimum T^j_{min}. Combining Eq. (2.7) with

Eq. (2.94) leads to

$$\Leftrightarrow \quad \text{Prob}[D^j \geq D^j_{max}] = \alpha_j e^{-D^j_{max}\theta^*_j} \leq \delta^j \tag{2.95}$$

$$\Leftrightarrow \quad \theta^*_j \geq \delta^j_D = -\frac{1}{D^j_{max}} \ln\left(\frac{\delta^j}{\alpha_j}\right). \tag{2.96}$$

Converting Eq. (2.94) into Eq. (2.95) is merely a change of notation in order to match the notation in Eq. (2.7).

To further simplify Eq. (2.96) using Eq. (2.8), we quantify the statistical properties of job arrival and power target. Because different types of jobs have different power consumption and processing times, a job cannot be simply regarded as a "request" in the original GPS algorithm explained in Sec. 2.4.2. Instead, we convert jobs and servers into the unit of "amount of service". A j^{th}-type job, using m_j servers to run and with a minimum processing time of T^j_{min}, is considered as requiring $m_j T^j_{min}$ amount of service. As a result, if we assume the job arrival for this queue follows a Poisson process with parameter λ_j (i.e., the average number of j^{th}-type jobs arriving per unit time), then, the amount of service injected to the j^{th} queue per unit time, $A_j(t)$, follows a Poisson process. The log moment-generating function for $A_j(t)$ is

$$\Lambda_{A_j}(\theta) = \lambda_j(e^{\theta m_j T_j} - 1). \tag{2.97}$$

Similarly, $n(t)$ active servers at time t are considered as having a processing capability of $B(t) = n(t)$ amount of service per unit time. Since the matching of power target with data center power gives us the relation Eq. (2.88), the statistical property of $n(t)$ depends on the property of the signal $y(t)$. Regulation service programs require the average value over a long time $\bar{y}$ to be close to 0. From the ISO signal sample we have, we empirically determine that the signal $y(t)$ generally follows a normal distribution, whose standard deviation is estimated as $y_\sigma = 0.40$. As a consequence,

$n(t)$ follows a normal distribution with an average value

$$n_\mu = \frac{\bar{P} - p_{idle}N}{\left(\sum_{j=1}^{J} w_j p_j\right) - p_{idle}}, \tag{2.98}$$

and a standard deviation

$$n_\sigma = \frac{y_\sigma R}{\left(\sum_{j=1}^{J} w_j p_j\right) - p_{idle}}. \tag{2.99}$$

Thus, the log moment-generating function of $B(t)$ is

$$\Lambda_B(\theta) = n_\mu \theta + \frac{1}{2} n_\sigma^2 \theta^2. \tag{2.100}$$

Eqs. (2.8)(2.97)(2.100) provide us the θ_j^* defined in Eq. (2.96), and as we have seen, satisfying Eq. (2.96) provides the QoS assurance we need.

Although θ_j^* can be derived from the statistic properties of job arrival and ISO signal, the coefficient α_j can only be estimated empirically. In the QoSG policy, we obtained a fixed estimate of this coefficient by running one experiment and fitted it with the observed QoS-degradation probability using Eq. (2.95). However, because the fixed estimate of α_j using a specific set of parameters ($\bar{P}$, R, and w_j) could have errors at other $\bar{P}$, R, w_j values, in the following, we improve this procedure by adaptively adjusting the estimation of α_j while optimizing the cost function using gradient descent.

In order to apply the gradient descent optimization, we make the QoS-assurance constraint (Eq. (2.90)) as a part of the cost function (Eq. (2.89)), and we estimate the cost of tracking error as C_{error}. Then, the cost function becomes

$$\begin{aligned} C &= \left(\Pi^P \bar{P} - \Pi^R R\right) H + C_{error} \\ &\quad + \beta \sum_j \text{SoftPlus}\left(\rho \left(\text{Prob}[Q^j - Q^j_{thres}] - \delta^j\right)\right). \end{aligned} \tag{2.101}$$

Here, H represents 1 hour. Function SoftPlus(x) is defined as $\ln(1+e^x)$, which is a smooth approximation of the ramp function $\max(0,x)$. Therefore, the QoS-related term in Eq. (2.101) is close to zero when the QoS constraint is met, and positive when violated. Parameters β and ρ control whether the QoS constraints are less or more strict. Although larger β and ρ result in more strict constraints, they also make the surface of cost function steeper, and consequently, finding the optimum becomes harder.

To calculate the derivatives of the tracking error cost, we need an analytical estimation of the tracking error cost:

$$
\begin{aligned}
C_{error} &= \Pi^{\epsilon} R \bar{\epsilon} H && (2.102)\\
&= \Pi^{\epsilon} \int_0^H |P_{target}(t) - P_{actual}(t)| dt && (2.103)\\
&\simeq \Pi^{\epsilon} \int_0^H (\bar{P} + y(t)R - P_{actual}(t)) dt && (2.104)\\
&= \Pi^{\epsilon} \left[\bar{P}H - E_{actual}\right] && (2.105)\\
&\simeq \Pi^{\epsilon} H \left[\bar{P} - p_{idle}N - \sum_j \lambda_j m_j T_j \left(p_j - p_{idle}\right)\right]. && (2.106)
\end{aligned}
$$

Here, H again represents 1 hour. Eq. (2.102) is the definition of tracking error cost, and Eq. (2.103) is from the definition of the average tracking error. Eq. (2.104) removes the absolute value sign because this tracking error term is significant only when $\bar{P} + y(t)R \gg P_{actual}(t)$, and the other case where $\bar{P} + y(t)R \ll P_{actual}(t)$ is precluded by the QoS-related term in Eq. (2.101) because a small $\bar{P}$ already violates QoS significantly. In Eq. (2.105), E_{actual} represents the actual energy consumption in that 1 hour, which is estimated in Eq. (2.106) following our assumption of job arrivals according to Poisson processes.

Applying Eqs. (2.95)(2.97-2.101)(2.106), we can compute the derivatives of the cost function, $\frac{\partial C}{\partial \bar{P}}$, $\frac{\partial C}{\partial R}$, $\frac{\partial C}{\partial w_j}$, to be used in gradient descent optimization. The following derivations show how we compute these derivatives:

We show the derivation of $\frac{\partial C}{\partial \bar{P}}$. The other derivatives including $\frac{\partial C}{\partial R}$ and $\frac{\partial C}{\partial w_j}$ can be derived similarly.

Starting with Eq. (2.101), we have

$$\begin{aligned}\frac{\partial C}{\partial \bar{P}} &= \Pi^P H + \Pi^\epsilon H + \\ &\quad \beta \sum_j \left(\frac{e^{\rho(\text{Prob}[Q^j - Q^j_{thres}] - \delta^j)}}{1 + e^{\rho(\text{Prob}[Q^j - Q^j_{thres}] - \delta^j)}} \times \right. \\ &\quad \left. \rho \frac{\partial}{\partial \bar{P}} \text{Prob}[Q^j - Q^j_{thres}] \right). \end{aligned} \tag{2.107}$$

Using Eq. (2.95), we obtain

$$\frac{\partial}{\partial \bar{P}} \text{Prob}[Q^j - Q^j_{thres}] = \alpha_j e^{-D^j_{max}\theta^*_j} (-D^j_{max}) \frac{\partial \theta^*_j}{\partial \bar{P}}. \tag{2.108}$$

To calculate the derivatives of θ^*_j, we first calculate θ^*_j by plugging Eq. (2.100) into Eq. (2.8), which yields

$$\begin{aligned} & \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\Lambda_B(-\theta w_j) & (2.109) \\ = & \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -n_\mu(-\theta w_j) - \frac{1}{2} n_\sigma^2 (-\theta w_j)^2 & \\ = & \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\frac{1}{2} n_\sigma^2 w_j^2 \left(\theta - \frac{n_\mu}{n_\sigma^2 w_j} \right)^2 + \frac{n_\mu^2}{2n_\sigma^2}. & (2.110) \end{aligned}$$

Here, whether the maximum point $\frac{n_\mu^2}{2n_\sigma^2}$ of the above quadratic function can be reached depends on whether $\theta = \frac{n_\mu}{n_\sigma^2 w_j}$ meets the conditions $\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0$.

To evaluate these conditions, we plug Eqs. (2.97)(2.100) into Eq. (2.9) and obtain

$$\Lambda_{GPS,j}(\theta) = \lambda_j (e^{\theta m_j T_j} - 1) - n_\mu \theta w_j + \frac{1}{2} n_\sigma^2 \theta^2 w_j^2.$$

As the second-order derivative $\Lambda''_{GPS,j}(\theta)$ is always positive, the function $\Lambda_{GPS,j}(\theta)$ is convex. Since 0 is a root of $\Lambda_{GPS,j}(\theta)$, the other root will be positive if and only if

$$\begin{aligned} & \Lambda'_{GPS,j}(\theta)\big|_{\theta=0} < 0 && (2.111)\\ \Leftrightarrow\quad & w_j > \frac{\lambda_j m_j T_j}{n_\mu} && (2.112)\\ \Leftrightarrow\quad & n_\mu w_j > \lambda_j m_j T_j. && (2.113)\end{aligned}$$

Equation (2.113) is actually the requirement that the average computing service provided to the j-th queue should be larger than the average amount of work submitted to this queue. Thus, we can safely assume Eq. (2.113) is satisfied, otherwise the queue length will diverge and the QoS constraint will be violated. Therefore, there exists a positive root for $\Lambda_{GPS,j}(\theta)$, and whether $\theta = \frac{n_\mu}{n_\sigma^2 w_j}$ meets the conditions $\Lambda_{GPS,j}(\theta) < 0$ can be converted to

$$\begin{aligned} & \Lambda_{GPS,j}\left(\frac{n_\mu}{n_\sigma^2 w_j}\right) < 0 \\ \Leftrightarrow\quad & \lambda_j\left(e^{\frac{n_\mu m_j T_j}{n_\sigma^2 w_j}} - 1\right) < \frac{n_\mu^2}{2n_\sigma^2} \\ \Leftrightarrow\quad & w_j > \frac{n_\mu m_j T_j}{n_\sigma^2 \ln\left(1 + \frac{n_\mu^2}{2n_\sigma^2 \lambda_j}\right)}. && (2.114)\end{aligned}$$

In the following, we separately discuss the two cases depending on whether Eq. (2.114) is satisfied or not.

Case I: If Eq. (2.114) holds, from Eq. (2.110) we get

$$\theta_j^* = \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\Lambda_B(-\theta w_j) = \frac{n_\mu^2}{2n_\sigma^2}. \tag{2.115}$$

From Eq. (2.98), we get

$$\frac{\partial n_\mu}{\partial \bar{P}} = \frac{1}{\left(\sum_{j=1}^{J} w_j p_j\right) - p_{idle}}. \tag{2.116}$$

To summarize, in Case I, combining Eqs. (2.107)(2.108)(2.115)(2.116) provides us the derivative $\frac{\partial C}{\partial \bar{P}}$ we need.

Case II: If Eq. (2.114) does not hold, i.e.,

$$w_j \leq \frac{n_\mu m_j T_j}{n_\sigma^2 \ln\left(1 + \frac{n_\mu^2}{2n_\sigma^2 \lambda_j}\right)} \tag{2.117}$$

assuming Θ_j is the positive root of $\Lambda_{GPS,j}(\theta)$, i.e.

$$\Lambda_{GPS,j}(\Theta_j) = \Lambda_{A_j}(\Theta_j) + \Lambda_B(-\Theta_j w_j) = 0 \tag{2.118}$$

then the supremum in Eq. (2.109) is achieved at Θ_j, i.e.,

$$\begin{aligned} \theta_j^* &= \sup_{\theta \geq 0,\ \Lambda_{GPS,j}(\theta) < 0} -\Lambda_B(-\theta w_j) \\ &= -\Lambda_B(-\Theta_j w_j) \\ &= \Lambda_{A_j}(\Theta_j) \\ &= \lambda_j(e^{\Theta_j m_j T_j} - 1). \end{aligned}$$

Then, we have

$$\frac{\partial \theta_j^*}{\partial \bar{P}} = \lambda_j e^{\Theta_j m_j T_j} m_j T_j \frac{\partial \Theta_j}{\partial \bar{P}}. \tag{2.119}$$

To calculate $\frac{\partial \Theta_j}{\partial \bar{P}}$, we take derivative $\frac{\partial}{\partial \bar{P}}$ on both sides of Eq. (2.118), and after re-arrangement, we get

$$\frac{\partial \Theta_j}{\partial \bar{P}} = \frac{\frac{\partial n_\mu}{\partial \bar{P}} \Theta_j w_j - n_\sigma \frac{\partial n_\sigma}{\partial \bar{P}} \Theta_j^2 w_j^2}{\lambda_j e^{\Theta_j m_j T_j} m_j T_j - n_\mu w_j + n_\sigma^2 \Theta_j w_j^2}. \tag{2.120}$$

From Eq. (2.99), we have

$$\frac{\partial n_\sigma}{\partial \bar{P}} = 0. \tag{2.121}$$

To summarize, in Case II, combining Eqs. (2.107)(2.108)(2.116)(2.118-2.121) provides us the derivative $\frac{\partial C}{\partial \bar{P}}$ we need.

Figure 2·19(b) shows the algorithm we apply to perform the optimization. The algorithm starts with a set of initial values for parameters $\bar{P}$, R, w_j, and α_j, and runs a one-hour[4] simulation. Next, parameter α_j is estimated for each jobtype j by fitting the QoS degradation curve with Eq. (2.95). Then, we calculate the cost function with its derivatives, and update $\bar{P}$, R, w_j through gradient descent[5]. These new parameters are fed into simulation again and we iterate over the process above. In this optimization approach, the time complexity of performing the theoretical calculations is constant irrespective of the data center size, and in our experiments, the time spent on theoretical calculations is negligible (less than 1 second). The time for conducting simulation depends on the data center size. In our experiments, this optimization approach always finds a solution meeting all constraints in less than 200 iterations, which takes no more than a few minutes even for a large data center with 10k nodes.

2.7 Experiments of AQA policy on MGHPCC Servers

We implement our AQA policy and evaluate it using both simulation and real-system experiments. The architecture of our implementation follows the same design as in Fig. 2·2. Among all the servers, we choose one server to be the "master" that receives the ISO signal, applies our AQA policy, and controls job scheduling and power capping. The other servers are called "clients" and they communicate with the master frequently (once per second in our experiments) to receive a control message and send their job/power status. We implement the communication between the master and clients using *rabbitmq* (Chapman et al., 2018).

A controller process in each client server executes the power capping of the server.

[4]Simulating a one-hour running time of the data center takes 5 seconds using our simulator.

[5]Because there is a constraint $\sum_j w_j = 1$ in our optimization problem, we apply gradient descent with projection.

It reads the power consumption from sensors and accordingly sets a cap on the CPU power to match the control message sent from the master. In our system, that controller process is a PID controller with $P = 0.4$, $I = 0$, $D = 0$. We determine these parameters by running benchmark applications under a changing powercap and selecting the parameters that shows the quickest response and the best stability.

2.7.1 System Setup

We use 36 servers from the Boston University Shared Computing Cluster (BU-SCC) that are physically located at the Massachusetts Green High Performance Computing Center (MGHPCC). Each server has two Intel Xeon Gold 6132 processors. Each processor has 14 cores, and its thermal design power is 140 W. When conducting experiments on this cluster, we use one server as the master, and the other 35 as clients.

We use the Linux *perf* tool (Melo and Carvalho, 2010) to read the power of CPU and memory in a server. We also use the IPMI tool (Laurie et al., 2018) to read a entire server's power, which includes the power from CPU, memory, disk, fan, network interface, motherboard, etc. Because the IPMI reading is a running-average value of the server power and it has a large granularity of 4 W, we fit a power model based on the *perf* reading to obtain the real-time server power and increase the granularity. We fit the power model by running benchmark applications and collecting the CPU/memory power (P_{proc} and P_{mem}) from *perf* and server power (P_{server}) from IPMI. We find that a linear model, $P_{server} = \phi_1 P_{proc} + \phi_2 P_{mem} + \phi_3$, is accurate enough for our purpose, and we empirically determine $\phi_1 = 1.29$, $\phi_2 = 1.63$, $\phi_3 = 44.0$ W in our experiments.

Table 2.4: Applications and workloads used in evaluation. The meaning of application name is shown by this example: `bt.C.16` means running benchmark `bt` with input C and with 16 threads. Here, m_j is the size (number of servers used to run). T_{min} (T_{max}) is the minimum (maximum) processing time in seconds and p_{max} (p_{min}) is the corresponding power consumption of a server in Watts.

App	m_j	T_{min}	p_{max}	T_{max}	p_{min}	Q_{thres}	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
bt.C.16	1	73	339	86	249	2.5		✓					✓	✓		✓				✓
bt.C.25	1	53	402	70	276	4.7		✓					✓							
mg.D.16	1	84	380	105	266	2.8				✓			✓		✓	✓				
sp.C.16	1	54	375	62	267	7.1		✓		✓			✓		✓		✓			
ep.D.88	4	40	360	53	237	6.3			✓											
is.D.32	3	42	249	42	241	5.6	✓			✓			✓	✓			✓	✓	✓	
bt.C.36	2	38	343	46	249	3.1						✓	✓	✓		✓			✓	
bt.D.49	2	551	391	729	250	5.6	✓				✓		✓		✓		✓	✓	✓	
ep.D.64	3	54	353	70	237	3.9				✓			✓		✓					
sp.D.100	4	343	399	381	264	3.3			✓		✓		✓		✓					
lu.D.224	8	89	400	119	250	7.6			✓								✓			
ep.D.28	1	124	383	175	238	5.9					✓				✓					
cg.C.4	1	28	238	28	239	4.0				✓				✓						
bt.D.25	1	1022	402	1370	254	3.2		✓			✓		✓		✓	✓				
lu.D.28	1	763	429	954	270	5.0						✓								
mg.D.8	1	141	297	151	258	2.9	✓	✓			✓		✓	✓		✓		✓	✓	
sp.D.16	1	1165	355	1302	270	5.5		✓					✓				✓			
is.D.4	1	122	204	123	194	7.3	✓				✓							✓	✓	
cg.D.16	1	743	326	823	253	7.3		✓					✓	✓			✓			
ep.D.56	2	64	383	90	238	2.0														✓
ft.D.64	3	284	321	313	247	3.1							✓			✓				
ft.D.128	6	165	321	179	242	7.7			✓				✓							
sp.D.196	8	329	370	352	253	3.7			✓											
lu.C.28	1	29	413	43	255	6.9		✓		✓										
cg.D.128	6	231	336	242	246	4.0			✓		✓			✓						
cg.D.32	3	364	281	390	246	5.5					✓			✓			✓			
ep.D.100	4	36	366	49	238	4.5	✓		✓							✓		✓	✓	
is.D.64	4	27	287	28	228	3.1	✓		✓	✓								✓		
lu.D.112	4	164	405	222	251	4.1	✓									✓		✓	✓	
mg.D.32	2	49	378	58	266	5.0	✓			✓			✓		✓		✓	✓	✓	
sp.C.64	3	31	371	32	258	2.2														✓

2.7.2 Workload Profile

We generate 14 different workload traces using parallel applications from the NAS Parallel Benchmark (NPB) suite (Bailey et al., 1991). These benchmarks allow a few different inputs and can be processed using different number of threads/servers. Because modifying the input and the number of threads/servers significantly changes the processing time and power consumption, in our evaluation, we use the word "application" or "a type of job" to refer to a benchmark with a specific input and processed with a specific number of threads on a specific number of servers. For example, application `bt.C.16` means running benchmark `bt` with input C and with 16 threads, and we run it on 1 server, as shown in Table 2.4.

Table 2.4 shows the properties of the applications and the composition of the 14 workload traces: W1∼W14. The applications in each trace are marked in the table. Column m_j is jobsize that represents the number of servers we use to run a certain application. When running an application, T_{min} (T_{max}) is the minimum (maximum) processing time in seconds and p_{max} (p_{min}) is the corresponding power consumption of a server in Watts. We determine the values of these parameters by running experiments on our 36-server cluster. Column Q_{thres} is the QoS threshold defined in Eq. (2.5), whose values are randomly generated within 2 to 7.9.

For each workload trace, the applications in the trace are selected with some randomness while following a trace-specific rule summarized in Table 2.5. In addition, 8 applications are selected in W1∼W5 and W8∼W13. Cases with less/more types of applications are explored by W6 and W7. We also assume a 50% data center utilization level for W1∼W11 and W14. Cases with lower/higher utilization level are explored by W12 and W13.

We generate a workload trace by generating the job arrival time of each application to follow a Poisson process with arrival rate λ_j. These arrival rates are related to the

Table 2.5: Workload trace properties.

Trace	Property	Note
W1	Mix	Applications cover a wide range of size, processing time, power, and QoS threshold
W2	Single	Only contains single-server applications
W3	LargeSize	Applications of size 4∼8 servers
W4	ShortTime	Applications with $T_{min} \leq 120$ s
W5	LongTime	Applications with 122 s $\leq T_{min} \leq$ 1022 s
W6	LessType	Contains 2 types of applications
W7	MoreType	Contains 16 types of applications
W8	LowPower	Applications with $p_{max} \leq 350$ W
W9	HighPower	Applications with $p_{max} > 350$ W
W10	TightQoS	Applications with $Q_{thres} \leq 5$
W11	SlackQoS	Applications with $Q_{thres} > 5$
W12	Util-25%	Average system utilization is 25%
W13	Util-90%	Average system utilization is 90%
W14	ExtraTight	Top 3 applications with the smallest Q_{thres}

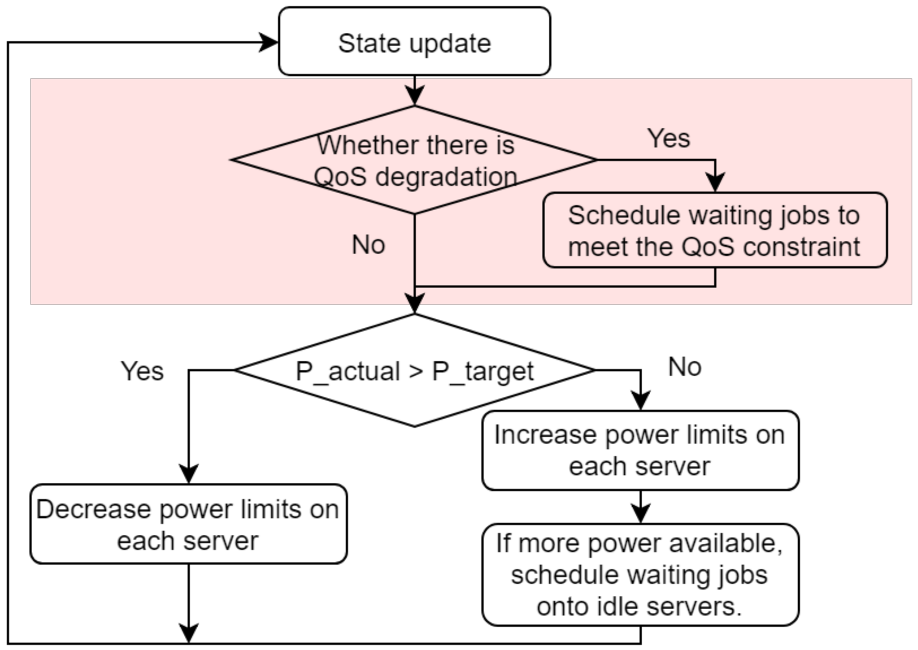

State update
Whether there is QoS degradation
Yes
Schedule waiting jobs to meet the QoS constraint
No
P_actual > P_target
Yes
No
Increase power limits on each server
Decrease power limits on each server
If more power available, schedule waiting jobs onto idle servers.

Figure 2·20: Simplified flow chart illustrating the two baseline policies. EnergyQARE (Chen et al., 2019) or Tracking-only policy (Chen et al., 2014) corresponds to the chart with or without the pink region, respectively.

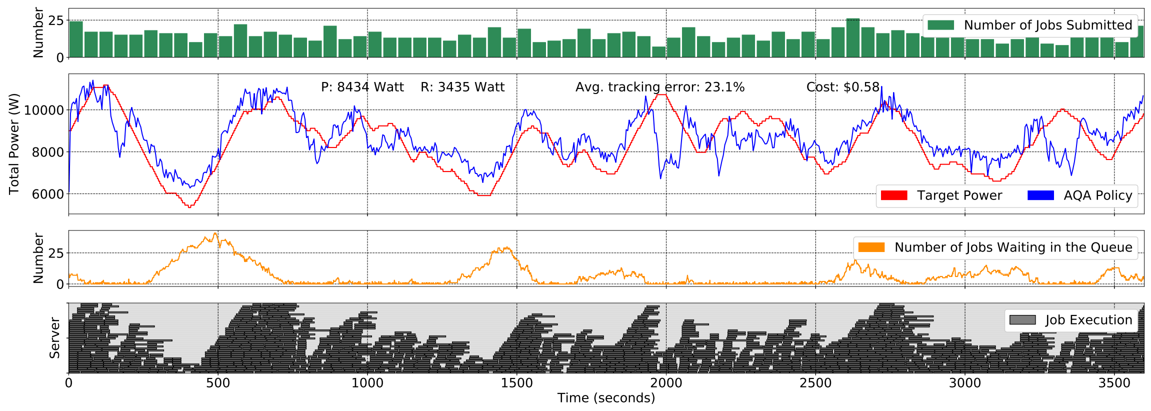


Figure 2·21: Experiments on a real 36-server cluster running workload W4 using our AQA policy.

data center utilization level η by an approximate equation:

$$\sum_{j=1}^{J} \lambda_j T_{j,min} = \eta N.$$

Assuming each application shares the data center utilization equally, we can derive the arrival rate for type-j jobs as

$$\lambda_j = \frac{\eta N}{T_{j,min} J}.$$

It deserves mentioning that our assumption of Poisson-distribution job arrival and normal-distribution ISO signal is only a special application of our policy for the experimental evaluation in this work, and our AQA policy can also be applied for other distributions of job arrivals and ISO signal.

For simulation, we build a simulator following the same architecture as the one shown in Fig. 2·2. Our simulator models simulated servers whose behaviors are close to the BU-SCC servers in our real-system experiments. When a simulated server is idle, we assume it consumes 169 W power, which is the average idle server power in our real system. When the simulated servers are running an application, we assume the power they consume and the time they take to finish the application follow the power and time values in Table 2.4. To be specific, when an application is run with p_{max} (or p_{min}) power, it takes T_{min} (or T_{max}) time to finish. If the power cap of running an application is set to be a value p_{cap} within the range of (p_{min}, p_{max}), then we assume the application finishes in T time in simulation. Here, T satisfies

$$\frac{T_{max} - T}{T_{max} - T_{min}} = \frac{p_{cap} - p_{min}}{p_{max} - p_{min}}.$$

This formula follows a first-order approximation to the actual power-performance relation.

2.7.3 Baseline Policies

We compare our AQA policy with two baselines proposed in previous works: the Tracking-only policy (Chen et al., 2014) and the EnergyQARE policy (Chen et al., 2019). The flow chart in Fig. 2·20 with or without the pink region corresponds to the EnergyQARE or the Tracking-only policy. Obviously, the Tracking-only policy focuses entirely on tracking the target and it ignores job QoS. EnergyQARE has an additional QoS-aware block (shown in pink) which is activated whenever the relative QoS degradation, defined as Q^j/Q^j_{thres}, is larger than the relative tracking error, defined as $\epsilon/0.3$. This QoS-aware block tries to schedule waiting jobs as much as possible and the power target constraint is ignored temporarily. To select parameters $\bar{P}$ and R in EnergyQARE, we do a grid search in the valid range of $\bar{P}, R$ by running simulations, and we choose the best $\bar{P}, R$ that minimize QoS degradation and tracking error. We use the same $\bar{P}, R$ from EnergyQARE to run the Tracking-only policy.

2.7.4 36-server Real-System Experiments

To conduct real-system experiments, we run each workload trace and each policy (AQA, Tracking-only, and EnergyQARE) for one hour on our cluster. We are not applying job preemption and standby-job-queue in Sections 2.7.4-2.7.5. Later in

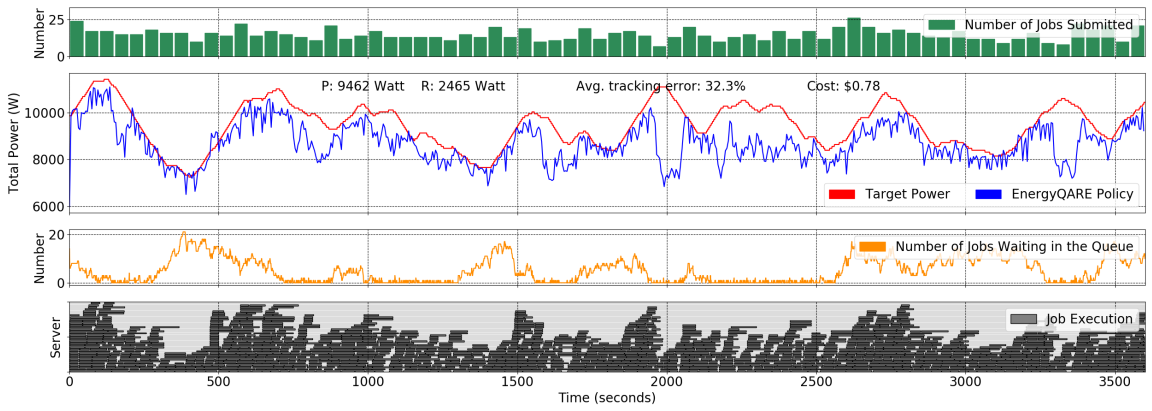


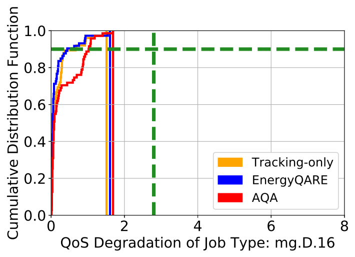
Figure 2·22: Experiments on a real 36-server cluster running workload W4 using the EnergyQARE policy.

Section 2.7.6, we present the results of AQA policy with standby jobs. In Section 2.7.7, we present the results of AQA policy with job preemption. All results presented in Sections 2.7.4-2.7.6 are from experiments on our real system. Results in Section 2.7.7 are from simulation.

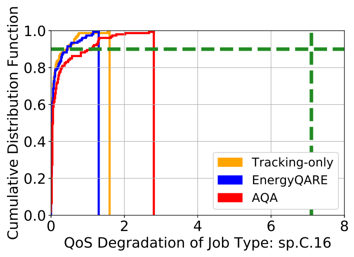
Figure 2·21 shows a typical result for running the AQA policy on our 36 servers. The workload trace for this experiment is W4. The green bars show the number of jobs (summed over all types) submitted to the queues in each time interval. The red curve represents the target power calculated from the ISO signal. The blue curve represents the total power consumption of the 35 client servers. We see clearly that the real power consumption follows closely with the target power.

The yellow curve shows the total number of jobs waiting in the queues. Clearly, this waiting job number is negatively correlated with the target power. At $t = 400$ s and 1400 s, when the target power drops, the number of active servers needs to be diminished, and consequently, more jobs are held in the waiting state. The deep grey stripes at the bottom of Fig. 2·21 show the period when a server is running a job, and the vertical displacement of a stripe denotes the server's index number.

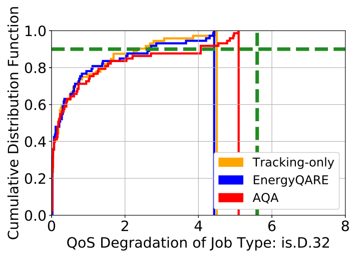
In this experiment, the estimated electricity cost for these servers is $0.58 when



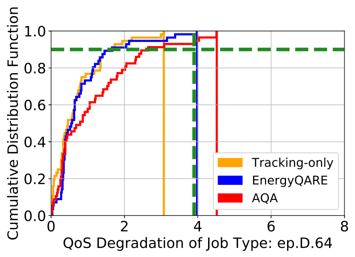
(a) `mg.D.16`



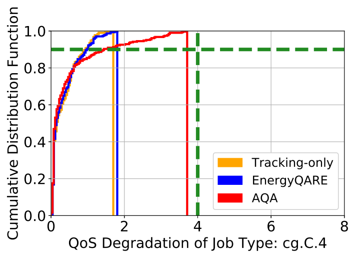
(b) `sp.C.16`



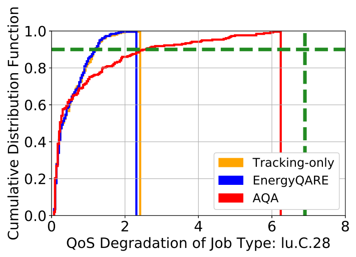
(c) `is.D.32`



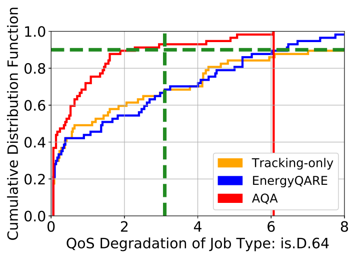
(d) `ep.D.64`



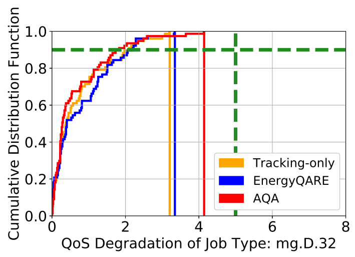
(e) `cg.C.4`



(f) `lu.C.28`



(g) `is.D.64`



(h) `mg.D.32`

Figure 2·23: The cumulative distribution functions (CDF) of all 8 applications' QoS degradation when running workload W4 in real-system experiments with three different policies. Both the EnergyQARE and the Tracking-only policies cannot meet the QoS constraints of application `is.D.64`, as shown in (g). On the other hand, our *QoSG* policy can meet that application's QoS constraint by giving it a large weight as shown in Fig. 2·25. In these CDF curves, the solid vertical line shows where the curves reach 100%.

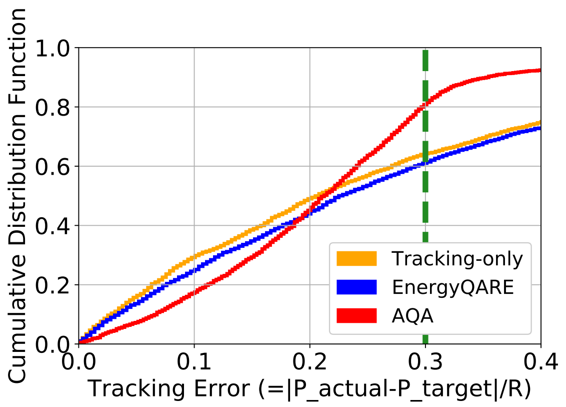


Figure 2·24: The cumulative distribution functions (CDF) of tracking error according to the three policies.

applying our AQA policy[6]. On the other hand, the electricity cost for applying the Tracking-only or EnergyQARE policy is \$0.77 or \$0.78, 33% higher than AQA (see Fig. 2·22 for the results of the EnergyQARE policy). Figure 2·24 and Fig. 2·23 compares the cumulative distribution functions (CDFs) of the tracking-error violation and the QoS violation among the three policies. This proves that our AQA policy provides more cost reduction than the baselines without violating tracking and QoS constraints. If not participating in demand response, the electricity cost can be estimated as $\Pi^P \bar{P} \times 1\text{h} = \$0.84$. Therefore, we conclude that AQA policy reduces the cost by $1 - 0.56/0.84 = 33\%$.

The optimal $\bar{P}$ and R for AQA policy with this workload are 8434 W and 3435 W, which are selected following the gradient-descent-based method discussed in Sec. 2.6.3. Figure 2·25 shows how the weights for the 8 types of application are adjusted through iterations. Although all weights are similar at initialization, the weight for application `is.D.64` increases significantly to 31.3% at the end, meanwhile the weights for other applications decrease slightly to save space for `is.D.64` as all weights should sum to 1. The application `is.D.64` needs a much larger weight than others because of its

[6]We can see the value is at the correct magnitude according to this estimation: 8000 W × 1h × 70% (cost reduction) × 0.1\$/kWh = \$0.56.

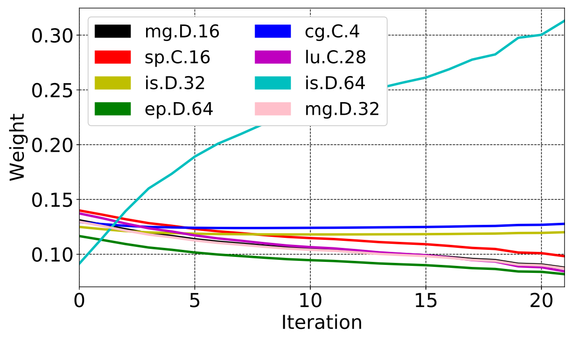


Figure 2·25: Weights adjusted by gradient-descent optimization.

relatively large size (taking 4 servers), short processing time (27 s), and relatively strict QoS constraint (Q_{thres}=3.1).

This capability of fine-tuning the weights for different applications by gradient-descent optimization is one of the key advantages of our AQA policy over the EnergyQARE policy. In EnergyQARE, all applications are treated similarly in using servers, so the jobs that are easier to violate their QoS constraint cannot gain higher priority. Our AQA policy, instead, balances the QoS of all types of applications by optimizing the cost function in Eq. (2.101). Through iterations of simulation and gradient-descent optimization, any application type that suffers from a high QoS violation probability will be given a higher weight by reducing the weights of other applications. If the weights of other applications are already at critical values and cannot be further reduced, $\bar{P}$ will increase and R will decrease to provide more space in tuning the weights.

2.7.5 Comparison of 14 Workload Traces

We experiment with the 14 workload traces listed in Table 2.5, and our findings are summarized as follows:

- With workload trace W2, all three policies meet both the QoS constraints and the tracking constraint.

Table 2.6: Experiments with 14 workload traces using the AQA policy.

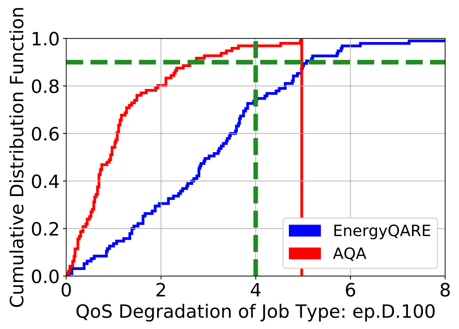
Trace	Cost	Cost Reduction
W1	\$0.58	30%
W2	\$0.57	37%
W3	\$0.68	29%
W4	\$0.58	31%
W5	\$0.34	56%
W6	\$0.64	34%
W7	\$0.58	33%
W8	\$0.56	29%
W9	\$0.58	37%
W10	\$0.58	34%
W11	\$0.46	46%
W12	\$0.46	38%
W13	\$0.71	31%
W14	\$0.64	33%

- With workload traces W6, W8, W10, and W11, AQA and EnergyQARE can meet all constraints, whereas the Tracking-only policy violates QoS constraints.

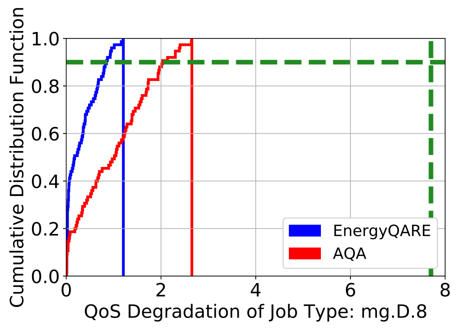
- With workload trace W1, W3∼W5, W7, W9, W12∼W14, only the AQA policy meets all constraints (standby jobs needed for W3, see Section 2.7.6), whereas the Tracking-only policy and the EnergyQARE policy violate either the QoS constraints or the tracking constraint.

In summary, our experiments with the 14 workloads prove that our proposed AQA policy outperforms EnergyQARE and Tracking-only as AQA meets QoS and tracking constraints in all tested workloads while EnergyQARE and Tracking-only cannot. The results also prove that Tracking-only is the worst among the three policies at meeting QoS objectives as Tracking-only can only meet the QoS constraints for 1 out of 14 workloads. Although Tracking-only policy performs well at tracking the regulation signal, it sacrifices QoS too much.

Table 2.6 summarizes the cost and cost reduction of AQA policy in these experiments. Here, cost reduction of AQA policy means the percentage of reduction with regard to the cost without demand response participation which is estimated as



(a) Application `ep.D.100`.



(b) Application `mg.D.8`.

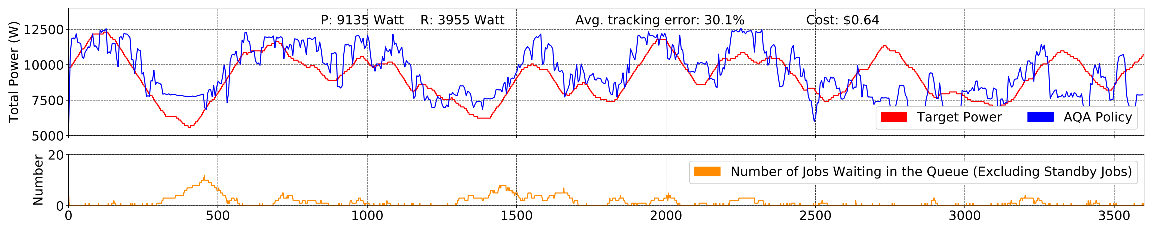
Figure 2·26: The cumulative distribution functions of QoS degradation when running W13 with AQA or EnergyQARE.

$\Pi^P \bar{P} \times 1\text{h}$. From this, we see our AQA policy reduces the electricity cost by 29-56%.

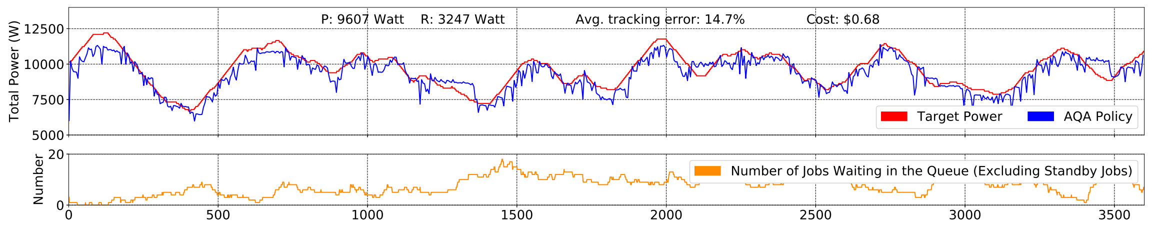
The single-server workload mix, W2, performs well in all evaluated policies. This matches the results of the evaluations in the prior works that proposed the EnergyQARE and the Tracking-only policies (Chen et al., 2014; Chen et al., 2019). Intuitively, single-server jobs are much better than multi-server jobs in regulation service participation because single-server jobs are easier for scheduling while larger jobs need to wait until the time when enough servers are available. In addition, even though all three policies perform well in trace W2, AQA is better than the other two as AQA achieves the lowest electricity cost: \$0.57, which is 7% (or 8%) lower than the \$0.61 (or \$0.62) from Tracking-only (or EnergyQARE).

It also deserves mentioning the result from workload trace W13 as it is a good example demonstrating the benefits of the adaptive optimization of weights in AQA policy. For this workload trace, AQA can meet both tracking and QoS constraints. EnergyQARE either fails to meet the tracking error constraint or fails to meet the QoS constraints of some applications even after we exhaustively experiment with all valid pairs of $\bar{P}, R$ in simulation. Figure 2·26(a) compares the cumulative distribution functions of the QoS degradation of application `ep.D.100` in AQA or EnergyQARE[7],

[7]In this figure, EnergyQARE runs with a best pair of $\bar{P}, R$ that has the lowest QoS degradation



(a) AQA policy without standby jobs.



(b) AQA policy with standby jobs.

Figure 2·27: Experiments on a real 36-server cluster running workload trace W3 using our AQA policy.

and Fig. 2·26(b) is for application `mg.D.8`. These figures show that the AQA policy enables both `ep.D.100` and `mg.D.8` to meet their QoS constraints. On the other hand, using EnergyQARE, `ep.D.100` fails to meet its QoS constraint while `mg.D.8`'s QoS is too good and even better than its QoS in AQA policy.

A main reason behind this is that `mg.D.8` runs on 1 server but `ep.D.100` takes 4 server, so `mg.D.8` is easier to be scheduled due to its small size, resulting in its low QoS degradation in EnergyQARE. However, the failure of `ep.D.100` to meet its QoS constraints in EnergyQARE proves that we need a mechanism to give different priorities to these two applications so that we can sacrifice the performance of `mg.D.8` to improve the performance of `ep.D.100`. Therefore, the reason why AQA enables both applications to meet their constraints is the adaptive optimization of weights. In fact, the gradient-descent-based optimization in AQA results in a 38.8% weight for `ep.D.100` and a 4.2% weight for `mg.D.8`.

To verify that our policy can be applied to QoS constraint values other than the

while meeting tracking error constraint.

specific Q_{thres} values shown in Table 2.4, we also conduct simulations for 10 times with randomized selection of Q_{thres} values within the range from 2 to 7.9. In all cases, we find that our optimization method is able to find weights and bidding parameters that guarantee all jobs to meet their QoS constraints while participating in demand response.

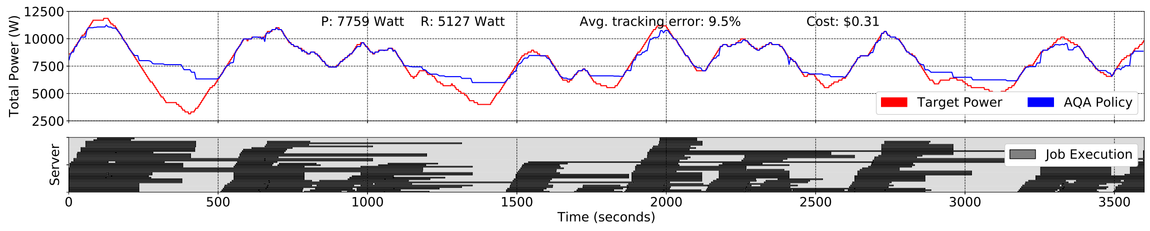
2.7.6 AQA with Standby Jobs

As discussed in Section 2.4.6, we suggest a standby job queue to solve the potential problem caused by lack of submitted jobs. In case that regular jobs are large in size, standby jobs also help improve the power tracking performance if their sizes are smaller.

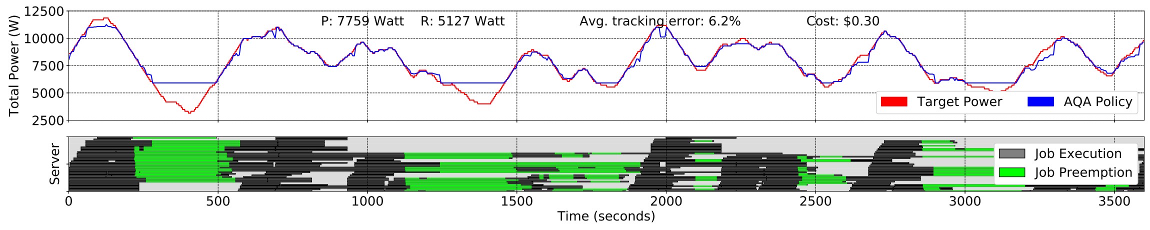
In our experiments with workload trace W3 where each regular job takes 4 to 8 servers (which is considered "large" as they already occupy 11% to 22% of our cluster), applying our AQA policy without standby jobs results in violation of tracking error (see Fig. 2·27(a)) because large jobs are much less flexible in scheduling. The average tracking error here is 30.1%. On the other hand, with standby jobs of one server in size, Fig. 2·27(b) shows better tracking performance where the average tracking error is 14.7%. The standby jobs will start whenever there are insufficient jobs. As a result, AQA with standby jobs smooths the power consumption curve and enables the power consumption to match the target at $t = 2700$, instead of leaving a 3000 W gap at $t = 2700$ in Fig. 2·27(a) due to the lack of waiting jobs.

2.7.7 AQA with Job Preemption

We evaluate job preemption in simulation using workload trace W5, where applications have relatively long execution time ranging from 122 s to 1022 s. Figure 2·28 compares the results of the AQA policy with or without job preemption. The same $\bar{P}$, R, and weights are used. The green regions in Fig. 2·28(b) represent the time where



(a) AQA policy without job preemption.



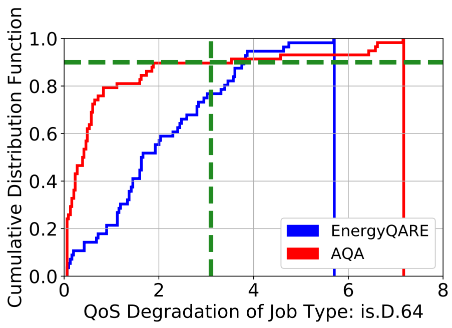
(b) AQA policy with job preemption.

Figure 2·28: Evaluating AQA policy with/without job preemption by simulation using workload trace W5.

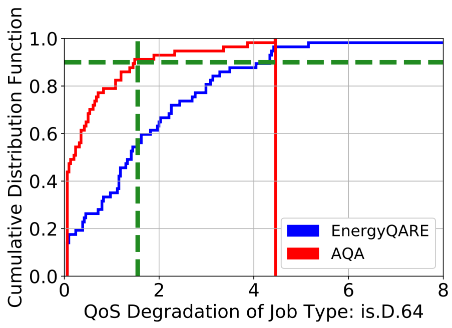
a job is preempted to reduce power and resumed later. As a result, the total power consumption at $t = 400$, $t = 1400$, $t = 1800$, etc. in job preemption case is lower than the case without job preemption, enabling AQA with job preemption to achieve a lower average tracking error of 6.2%, in comparison with the 9.5% tracking error from AQA without job preemption. Both cases in Fig. 2·28 meet the tracking error constraint (according to their CDFs curves, not shown here), so the real benefit of job preemption here is a slight decrease of electricity cost from \$0.31 to \$0.30. However, if using some workload traces that have application execution time even longer than the ones in W5, we find that AQA without job preemption may not be able to meet the tracking error constraint. In those cases, job preemption becomes necessary.

2.7.8 Comparison of different QoS constraint levels

To evaluate the impact of different QoS constraint levels on our policy, we conduct simulations with a workload when applications' QoS constraints are changed from tight to loose. Starting from the QoS constraints shown in Table 2.4 column Q_{thres} (we



(a) Medium QoS level.



(b) Tight QoS level.

Figure 2·29: Comparing results from workload W4 with either a medium or a tight QoS constraint level.

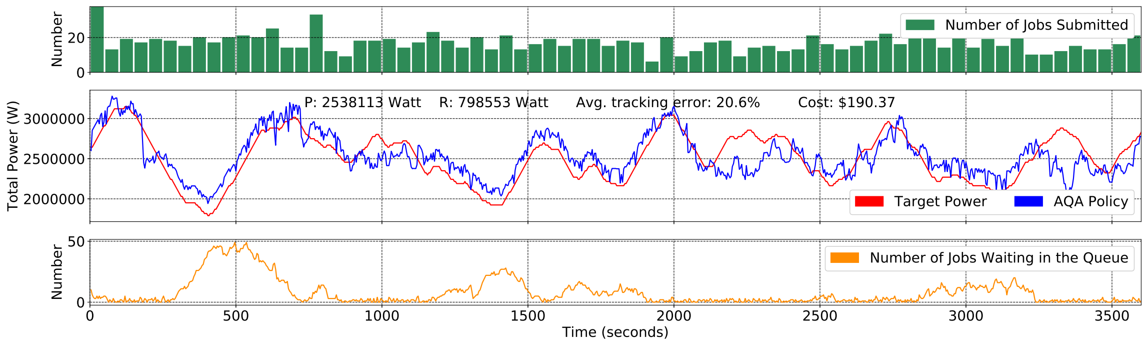


Figure 2·30: Evaluating *QoSG* policy by simulating a 10k-node data center with a job-size-scaled version of W1 workload.

call them as in a "medium" QoS constraint level), we tighten the QoS constraints of the applications by decreasing the Q_{thres} of each application by half, and we loosen the QoS constraints of the applications by doubling the Q_{thres} of each application. These simulations do not apply a standby job queue or job preemption. Figure 2·29 compares the simulation results of workload W4 with either a medium or a tight QoS constraint level. As we can see, when the QoS constraint levels change from medium in Fig. 2·29(a) to tight in Fig. 2·29(b), both our *QoSG* policy and the EnergyQARE policy improve the QoS of this application, and our policy enables the application to meet the QoS constraint while EnergyQARE policy does not. We only show the QoS degradation curve of application `is.D.64` due to the space limit, but the results for other applications have similar behaviors as this one. Therefore, we conclude that our *QoSG* policy has the opportunity to meet either a loose or a tight QoS constraint assuming there is a valid solution that can be achieved by selecting the appropriate values for the weights and bidding parameters.

2.7.9 Scalability to large data centers

To evaluate the scalability of our policy when applied to large data centers, we conduct simulation studies with a data center composed of ten thousand servers. Figure 2·30 shows the simulation results of running a job-size-scaled version of workload W1 for one hour. Here, the workload is composed of the same types of applications in Table 2.4 (W1) but the number of nodes used by each application is scaled up by 100x (e.g., application `is.D.32` takes 300 nodes instead of 3 nodes), and the job arrival rates are also adjusted to match an assumed 50% utilization level of this large data center. As shown in Fig. 2·30, our policy works well for a large data center with 10k-node and enables the actual power to follow the target power closely. We also check the QoS degradation of the applications, and they all meet their QoS constraints. The electricity cost with demand response participation in this experiment is \$190.4

according to Eq. (2.3). Without demand response participation, the electricity cost for running these jobs can be estimated as $\Pi^P \bar{P} \times 1\text{h} = \$253.8$. Therefore, the cost reduction for this experiment is 25%. If the data center has similar power consumption and cost reduction throughout the year, then it can save \$555,822 ($= 253.8 \times 25\% \times 24 \times 365$) per year by participation in demand response with our policy.

2.8 QoSCap Power Management Policy and Adaptive Bidding Policy

The QoSG and AQA policies proposed above enable data centers to participate in regulation service programs while abiding by job QoS contraints. However, these policies only consider a single bidding cycle (1 hour) and do not explicitly target parallel jobs with long execution times, from several hours to days, as these policies mostly rely on job scheduling to regulate a data center's power. Job scheduling, however, does not always provide sufficient control points when handling uninterruptible jobs with long duration. Also, long-duration jobs may go through multiple contract bidding cycles and previous bidding policies are not suitable for this scenario either. In addition, most prior approaches rely on synthetic workload traces to evaluate the policies, and they seldom experiment with real-world workload traces taken from system logs which contain large variations in job arrival times.

We propose the QoSCap power management policy and the Adaptive Bidding policy to address these needs. Our power management policy relies on the power capping capability of servers to regulate data center power for demand response participation. The proposed policy applies power limits while considering the Quality-of-Service (QoS) of jobs. We also propose an Adaptive Bidding policy that selects appropriate contract parameters for data centers to participate in regulation service markets when applying the QoSCap policy. We evaluate our policies by simulation

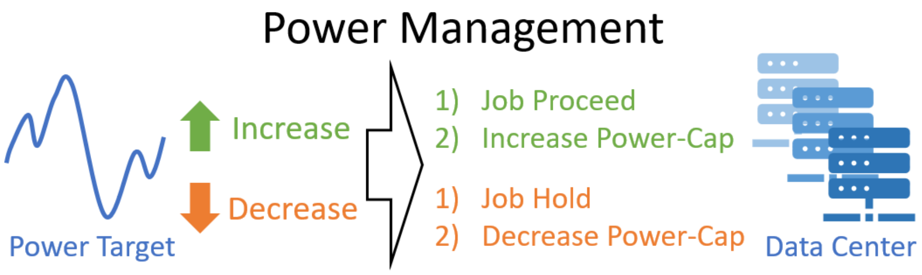


Figure 2·31: Power management policies regulate data center power through job scheduling and power-capping to match the actual power consumption with the target power.

using parameters and workload traces taken from a real data center (Patel et al., 2020). Our results show that our proposed policies enable data centers to participate in regulation service and save 10% on electricity costs while abiding by all QoS constraints of real workload traces.

In the following, we first discuss our new QoSCap power management policy, then we discuss three bidding policies including our new Adaptive Bidding policy. We make similar assumptions on the data center model as discussed in Section 2.3. In the following, we assume there are constraints on the average QoS degradation of each type of jobs: $\text{Avg}[Q_j] < Q_j^{thres}$. Here, Q_j^{thres} is the QoS threshold for job type j. A job's QoS degradation is $Q_j = (T_j^{so} - T_j^{min})/T_j^{min}$, where T_j^{so} is the sojourn time of the job in the system, including the waiting time T_j^{wait} and the actual processing time T_j^{proc}, i.e., $T_j^{so} = T_j^{wait} + T_j^{proc}$.

2.8.1 Power Management Policies

Power management policies match the actual data center power with the data center's target power through job scheduling and server power capping. A common strategy for these power management policies is to start running more jobs and to increase CPU power caps when the actual power is lower than the target power. These policies typically hold waiting jobs and decrease the CPU power caps when the actual power is higher than the target power, as shown in Fig. 2·31.

The QoSCap (QoSCap) policy not only applies the strategy in Fig. 2·31 but also intelligently adjusts power caps on servers considering the estimated QoS of jobs at run time. Every second, the policy calculates an *estimated QoS degradation* of each job by

$$Q_{est} = (T_{wait} + T_{elapse} + T_{remain})/T_{min}. \tag{2.122}$$

Here, T_{wait} is the waiting-in-queue time of the job. If the job is currently running, then T_{elapse} is non-zero and represents the time from job beginning to the current time. T_{remain} represents the remaining time to finish the job, estimated as the remaining percentage of the work to be done multiplied by the minimum execution time. In real world workloads, the percentage of work to be done for a job can be estimated based on the number of finished phases or loops of the job. Calculating the estimated QoS degradation metric enables the system to know which job's QoS degradation will exceed the threshold in advance.

Based on the estimated QoS degradation, the QoSCap policy starts jobs whose QoS is close to violation before it is too late to recover. The policy prioritizes job types with larger average QoS degradation when it is scheduling jobs to increase power consumption. The policy only decreases server power caps for job types that meet their QoS constraints. As a consequence, jobs that wait too long in the queue or run with too low power caps are reflected by their higher estimated QoS degradation values, and they will be prioritized to start and run with full power.

2.8.2 Bidding Policies

Bidding policies select the bidding parameters, $\bar{P}$ and R, at the beginning of every hour. $\bar{P}$ and R determine the average and the variation of the target power, as shown in Fig. 2·32.

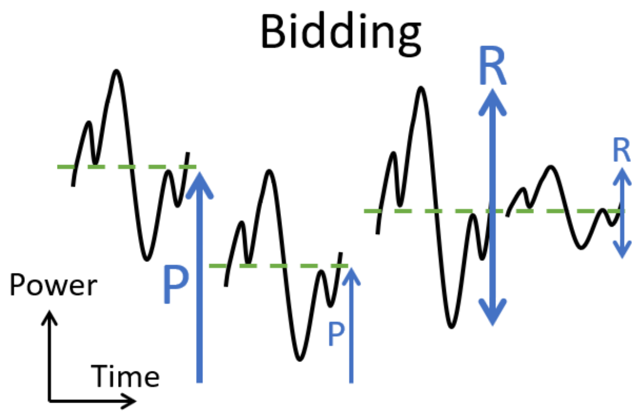


Figure 2·32: Bidding policies select the appropriate $\bar{P}$,R parameters that determine the average and the variation of the target power.

Based on the electricity cost in Eq. (2.3), a data center should pursue a smaller $\bar{P}$ and a larger R to reduce its cost. However, selecting $\bar{P}$ too small or R too large leads to QoS degradation. Therefore, an appropriate bidding policy is needed. In the following, we discuss the Fixed Heuristics Bidding policy and the Fixed Exhaustive Search Bidding policy used in previous works (Chen et al., 2014; Chen et al., 2019). Then, we introduce our new Adaptive Bidding policy.

The Fixed Heuristics (FH) Bidding policy selects $\bar{P}$ and R based on the long-term average power and power control range estimation. As long as the estimated job arrival rates do not change, that power estimation does not change over time, so the FH policy selects the same $\bar{P}$ and R at all times.

Assuming we know the arrival rate λ_j for each job type j, we can estimate the average number of active servers in the data center as $N_{active} = \sum_j \lambda_j m_j T_j^{min}$. Here, m_j is the size (i.e., required number of nodes) for each job of type j. Then, the total power of all idle servers can be estimated as $P_{idle} = (N_{total} - N_{active}) \times p_{idle}$ on average. The active servers can change their power consumption through power capping, so the total power of all active servers can vary from $P_{active}^{min} = N_{active} \times p^{min}$ to $P_{active}^{max} = N_{active} \times p^{max}$. Here, p^{min} or p^{max} is the min/max power of an active server, which can be estimated as the average min/max power of all job types, i.e., $p^{min} = (\sum_j p_j^{min})/J$, $p^{max} = (\sum_j p_j^{max})/J$. Here, J is the number of job types.

Based on the discussion above, we can derive the average minimum and maximum power of the data center as $P_{all}^{min} = P_{active}^{min} + P_{idle}$ and $P_{all}^{max} = P_{active}^{max} + P_{idle}$. Therefore, the FH policy selects $\bar{P} = (P_{all}^{max} + P_{all}^{min})/2$ and $R = (P_{all}^{max} - P_{all}^{min})/2$ as they are the estimated long-term average power and control range.

The Fixed Exhaustive Search (FES) Bidding policy selects the optimal parameters that minimize the electricity cost. It finds the optimal point through exhaustive search by running simulation over a wide range of $\bar{P}$, R with synthetic workloads (generated according to the arrival rates λ_j), and selects the parameters that minimize the electricity cost while meeting the tracking-error and the QoS constraints. The FES policy does the exhaustive search once and applies the fixed values of $\bar{P}$, R to all hours.

The Adaptive Bidding policy determines the bidding parameters based on the current active and waiting jobs in the system instead of selecting the parameters based on the long-term average utilization of a data center. The motivation is, when long-duration and large parallel jobs are common in a data center, the average power and the power control range in a certain hour could significantly deviate from their long-term average.

At the bidding time, the Adaptive Bidding policy calculates the possible max/min total power, P_{all}^{max} and P_{all}^{min}, by summing the max/min power of each active or waiting job. For an active or waiting job that suffers from a QoS violation ($Q_j > Q_j^{thres}$), its power is set as the maximum, p_j^{max} per node. For an active or waiting job not violating QoS constraints, its achievable min/max power is p_j^{min} and p_j^{max} per node. Idle nodes always consume power p_{idle}. After calculating P_{all}^{max} and P_{all}^{min}, the policy selects $\bar{P} = (P_{all}^{max} + P_{all}^{min})/2$ and $R = (P_{all}^{max} - P_{all}^{min})/2$.

2.9 Simulation Results of QoSCap and Adaptive Bidding Policies Using Real Workload Traces

To evaluate our policies in a real-world scenario, we conduct simulations using real system parameters and real workload traces taken from the *emmy* and *meggie* clusters at the Regional Computing Center in Erlangen (RRZE) (Patel et al., 2020). The following subsections describe how we obtained data about these clusters and their workloads, and how our simulator utilizes that data.

2.9.1 Workload Traces

Simulating a cluster requires the descriptions of the cluster's nodes, the workloads being executed, and the job submission times of the workloads in its job queue. We extract all of these properties from real traces provided by Patel et al. along with their analysis of the *emmy* and *meggie* clusters (Patel et al., 2020).

Node Properties

The *emmy* cluster has 560 nodes, and the *meggie* cluster has 728 nodes. The logs and traces we have available do not report the idle power of these nodes, but a workload efficiency model on this cluster by Klawonn et al. suggests that idle power is in the vicinity of 31 watts (Klawonn et al., 2020). So, we assume $p_{idle} = 31$ W.

Workload Extraction

The *emmy* and *meggie* clusters do not use the same job scheduler and resource manager, and their trace outputs follow a different format with different types of events. We use a common subset of the events and properties that are available from both clusters. The selected job description fields are:

- Average DRAM and CPU power from start to end

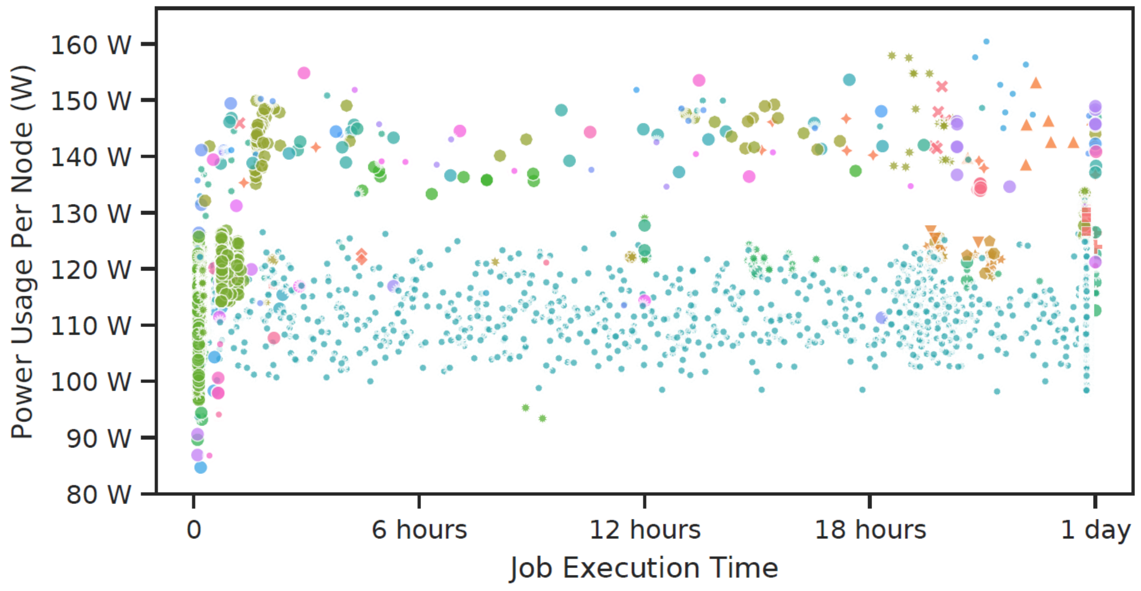
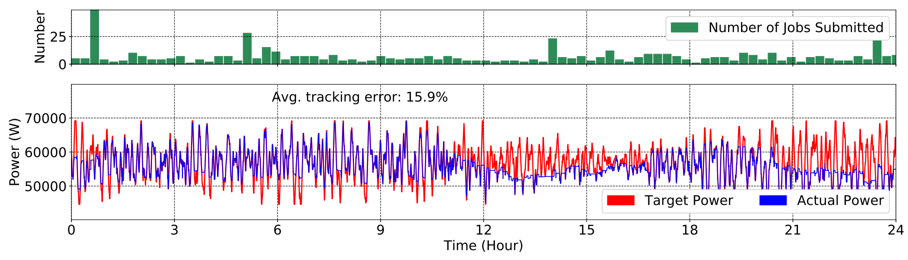


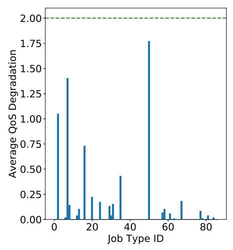
Figure 2·33: The logged power and execution time of jobs running on the *meggie* cluster in February 1-8, 2019.

- First enqueued time (in case a single job instance was enqueued and requeued multiple times, we only take the first time)
- Job start time and end time
- Anonymized job name
- Count of nodes assigned to the job

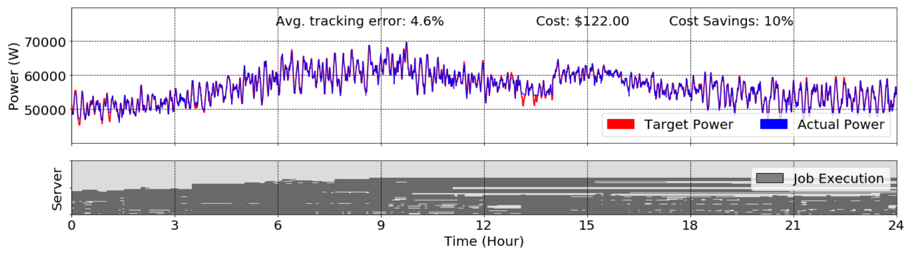
As the power reported in the logs only includes CPU and DRAM power, our simulation and analysis only considers the CPU and DRAM power, and we ignore the cooling power because we have no data for that. The data from the *emmy* cluster also reports job deletion events. We ignore any jobs that were deleted before they could start executing. An example of the power and execution time of jobs extracted from the log and used in our simulation are displayed in Fig. 2·33. Jobs with the same name and node count are shown as marks with the same color, size, and shape.



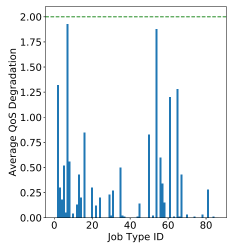
(a) EnergyQARE + FES Bidding policy results.



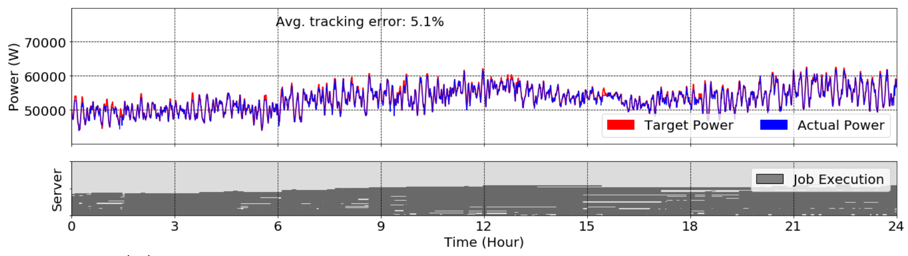
(b) QoS Degradation.



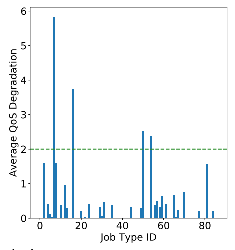
(c) QoSCap + Adaptive Bidding policy results.



(d) QoS Degradation.



(e) EnergyQARE + Adaptive Bidding policy results.



(f) QoS Degradation.

Figure 2·34: Simulating a data center participating in regulation reserve programs when applying our policies. The simulations use real server parameters and run workload traces taken from a real 728-node cluster called *meggie*. In (c) and (d), the QoSCap with Adaptive Bidding policy enables the data center to match its actual power (blue) with the target power (red) and meet the QoS constraints of jobs.

2.9.2 Simulator Setup

We design a simulator based on the parameters of the clusters and workloads. The simulator is fed idle power and the host count for each cluster, as well as a list of jobs and their properties. The workload properties used by the simulator are nodes per job, maximum allowed performance degradation, minimum and maximum observed power, and the elapsed run time of the job when running under those power levels. When we apply the FES bidding policy, to generate a synthetic queue for bidding parameter searches, our simulator also takes the mean job incoming rates as inputs. For our final evaluations of the selected bidding parameters, we replay the queue submission times from the cluster logs instead of using the synthetic queues.

To simulate the effect of a continuous range of power capping on the execution time of a job, the simulator assumes there is a linear relation between the power and execution time for each job type. To be specific, when we apply a power cap on every node of a job type j, if we reduce the power cap from p_j^{max} to p_j^{min}, we assume the execution time linearly increases from T_j^{min} to T_j^{max}. For a parallel job, we always apply the same power-cap to all the nodes running this job.

2.9.3 Simulation Results

We simulate the *meggie* or *emmy* cluster with job arrivals taken from multiple periods of the workload trace. Figure 2·34 presents a typical 24-hour result which simulates the *meggie* cluster with workload trace on Feb. 5^{th}, 2019. To get this result, we actually simulate the workload trace from Feb. $1^{st} - 5^{th}$ instead of starting directly from Feb. 5^{th}, so we can avoid the unrealistic underutilization of the cluster at the first few days. These simulations assume the threshold for the average QoS degradation of all job types (Q_j^{thres}) is 2.

Figure 2·34(a) shows the results of applying the EnergyQARE with FES Bidding policy, and the average QoS degradation of each job type when applying these policies is shown in Fig. 2·34(b). Figures 2·34(c)(d) shows the results of the QoSCap and Adaptive Bidding policy. Figures 2·34(e)(f) are for the EnergyQARE with Adaptive Bidding policy. In Figs. 2·34(a)(c)(e), the red curve shows the target power and the blue curve shows the simulated actual power of the cluster. The green bars in Fig. 2·34(a) show the number of jobs submitted to the cluster in different time intervals, and these job arrivals are the same for Figs. 2·34(c)(e) because they simulate the same workload trace. In Figs. 2·34(c)(e), we also draw the time span of executing each job as the gray lines, where the vertical placement of the gray lines represents the server index number. We also calculate the electricity cost according to Eq. (2.3), and it shows that the proposed QoSCap with Adaptive Bidding policy enables the data center to get an 10% reduction of its electricity cost compared to the cost without regulation service participation.

From the results in Fig. 2·34, we see that the first two policy combinations meet the QoS constraints of computing jobs, QoSCap with Adaptive Bidding also meets tracking constraints as we see the actual power curve follows the target power closely. On the other hand, EnergyQARE with FES Bidding cannot meet tracking constraints as the actual power sometimes cannot follow the lower part of the target (during hours 4 to 10) and sometimes cannot follow the higher part of the target (during hours 11 to 16). The $\bar{P}$ and R for this case are already optimally selected using the FES Bidding policy, so any other $\bar{P}$, R selection either violates the tracking constraints or violates the QoS constraints.

The reasons for EnergyQARE with FES Bidding not performing well include the abundance of long-duration parallel jobs and the large variation of job arrivals. The work that proposed the EnergyQARE and FES Bidding policies (Chen et al., 2019) targets minute-long and single-node jobs which provide higher granularity to regulate power through job scheduling. In the real workload trace we simulate here, jobs mostly have an execution time of multiple hours (up to 24 hours) and many parallel jobs require multiple nodes to run (up to 64 nodes).

Since jobs use many nodes at a time and span long durations, variations in job arrivals cannot be handled well by a fixed bidding parameter selection. On the other hand, Fig. 2·34(c) shows that the Adaptive Bidding policy can improve the tracking by selecting a higher $\bar{P}$ during hours 4 to 10 and a lower $\bar{P}$ later. However, EnergyQARE with Adaptive Bidding also does not perform well as shown in Fig. 2·34(e)(f) because QoS constraints are not met. This is because the EnergyQARE policy shifts its priority between tracking and QoS based on the average QoS of all jobs. So, QoS violations of a small count of job types is not handled by the policy, which leads to the large QoS violation of individual job types shown in Fig. 2·34(f).

Results for other combinations of power management and bidding policies are presented in Table 2.7. The table shows whether a policy combination meets the tracking or QoS constraints. The table also includes results for the 24-hour simulation of the *meggie* cluster with workload trace on Jan. 16^{th}, 2019, as well as the 24-hour simulation of the *emmy* cluster with workload trace on Nov. 15^{th}, 2018. Among all different policy combinations, only the QoSCap with Adaptive Bidding policy meets all constraints in all the three workload traces. Results for the Tracking-only policy are not listed since that policy is ignorant of QoS and never meets all constraints. We have also simulated several other periods of the entire workload traces (not shown) and we also observe the same result that QoSCap with Adaptive Bidding meets both constraints.

Workload	Power Management	Bidding Policy	Tracking Error	QoS Degrad.	Energy Cost
Meggie Feb. 5 2019	EnergyQARE	FH	7.7%	4.0	-
		FES	18.1%	1.8	-
		Adaptive	2.0%	5.8	-
	QoSCap	FH	9.3%	1.5	$124
		FES	20.3%	1.4	-
		Adaptive	1.0%	1.9	$122
Meggie Jan. 16 2019	EnergyQARE	FH	2.2%	3.0	-
		FES	0.2%	4.8	-
		Adaptive	1.9%	2.6	-
	QoSCap	FH	18.0%	2.0	-
		FES	4.1%	1.9	$151
		Adaptive	0.2%	2.0	$166
Emmy Nov. 15 2018	EnergyQARE	FH	73.9%	2.2	-
		FES	0.8%	1.0	$115
		Adaptive	0.1%	2.1	-
	QoSCap	FH	70.4%	1.9	-
		FES	2.8%	2.1	-
		Adaptive	0.7%	2.0	$119

Table 2.7: Simulation results of applying different policies for data center participation in regulation service. The proposed policy combination QoSCap + Adaptive is in bold. The "Tracking Error" column shows the percentage of large tracking error, and a value larger than 10% violates the tracking constraint. "QoS Degrad." column shows the largest average QoS degradation among all job types, and a value larger than 2.0 violates the QoS constraints. Values violating constraints are shown in red, otherwise in green. The electricity cost for policies that meet all constraints are displayed.

Chapter 3

Mitigating Network Congestion and Improving Performance through Novel Job Allocation Strategies

3.1 Introduction

Efficient system management in ever-growing high performance computing (HPC) systems is a common concern of designers, administrators, and users. As the number of cores required by parallel programs continues to increase, network communication time among the compute nodes becomes a performance bottleneck (Bhatele et al., 2013b; Leung et al., 2002). Recently, it is commonly reported that network congestion is a major cause of performance degradation in HPC systems (Bhatele et al., 2013a; Smith et al., 2016; Bhatele et al., 2020), and network congestion can lead to an extention on job execution time by 6X(Chunduri et al., 2017). Thus, mitigating network congestion and reducing the communication cost on HPC systems are essential to improve the performance of HPC systems and promote better utilization of valuable computation resources.

In Section 3.2, I propose a novel job allocation policy called Level-Spread which mitigates network congestion and improves performance by harnessing the hierarchical structure of dragonfly networks. This policy is evaluated in simulation and discussed in Section 3.3. In Section 3.5 and Section 3.6, I propose an approach to quantify the impact of network congestion on applications and demonstrate this approach using experiments on a large HPC system. My experiments demonstrate that applications which spend more time on MPI collective operations are more sensitive to network congestion, and I also show that certain network metrics are indicative of network congestion and performance degradation. Based on these findings, in Section 3.7 and Section 3.8, I propose and evaluate a network-data-driven job allocation policy. The policy uses network counter data to quantify the network traffic intensity on routers, and allocate jobs according to the collected traffic intensity.

3.2 Level-Spread Allocation Policy

Dragonfly (Kim et al., 2008) is a network topology utilizing high-radix routers, and has attracted attention in recent years (Belka et al., 2017; Fuentes et al., 2016; Faizian et al., 2016; Jain et al., 2016; Groves et al., 2016; Yang et al., 2016; Kaplan et al., 2017). Dragonfly networks are composed of interconnected *groups* that behave as virtual routers. A group contains a set of compute nodes, network links, and routers. One example of a group is shown in Fig. 3·1. Different groups are connected by optical *global links*, and routers in each group are connected by electrical *local links.* Each router is connected to several nodes. Owing to the all-to-all connections among its groups, dragonfly is a constant-diameter network (i.e., the diameter does not scale up with the increase in the number of nodes). A message sent from one node to another following the shortest-path route passes through at most two local links and one global link. The dragonfly topology has been implemented in some of the latest HPC systems (Faanes et al., 2012; Chakaravarthy et al., 2012).

An important factor that affects the communication time in HPC systems is job allocation (Leung et al., 2002), which is the procedure of selecting a set of compute nodes to run a parallel program. The hierarchy of the dragonfly topology brings new complexity to job allocation, and to date, job allocation on dragonfly has not

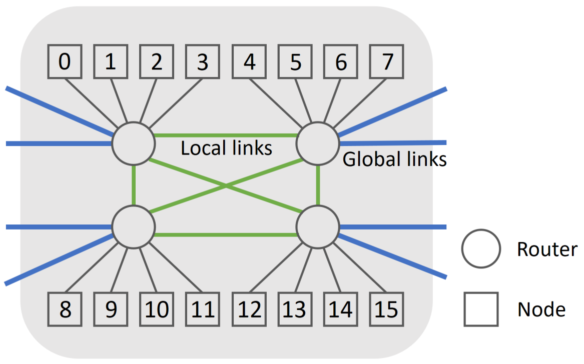


Figure 3·1: A group in a dragonfly network. The group is composed of 4 routers that are connected to each other by local links (green). Each router is connected to 4 nodes, which are labeled by numbers. The entire dragonfly machine (not shown here) consists of a number of such groups. Between each pair of groups there is a global link (blue) connecting them together.

been fully explored. Most studies on dragonfly networks only consider two allocation policies: *simple* allocation (i.e., selecting nodes by following the label order) and *random* allocation (Faanes et al., 2012). Jain et al. (Jain et al., 2014) proposed six different job allocation policies for dragonfly machines and concluded that random allocation is generally better than the others. Yang et al. (Yang et al., 2016) further demonstrated that random allocation cannot guarantee the best performance for every job.

Inspired by the prior work, we examine the shortcomings of existing allocation policies and conclude that a size-aware job allocation policy is necessary to further improve the performance on dragonfly networks. To this end, we propose the *Level-Spread allocation policy* (Sec. 3.2) and compare it with several state-of-the-art allocation policies (Sec. 3.3.4). We perform extensive simulations (Sec. 3.3) and show that our proposed policy outperforms the existing policies (Sec. 3.4). In addition, we discuss the influence of scheduling on the performance of allocation policies (Sec. 3.4.5). The specific contributions of our work are as follows:

- We demonstrate that on dragonfly networks, harnessing node locality and balancing link congestion are two complimentary approaches to reduce communication latency. Existing allocation policies only resemble either one of these two approaches. We show that combining them intelligently provides better performance.

- We design a novel allocation policy, *Level-Spread*, for dragonfly networks. This policy spreads jobs within the smallest network level that a given job can fit in. Through packet-level network simulations, we show that our proposed policy reduces the communication time by 16% on average compared to the state-of-the-art policies by harnessing node locality and balancing link congestion.

Our work distinguishes from the related work by the following points. First, we propose a size-aware job allocation policy, *Level-Spread*, specifically for the dragonfly topology. Prior work on size-aware allocation policies either focus on other network topologies or demonstrate performance tradeoffs without providing a concrete policy. Second, we evaluate our allocation policy on a broad range of workloads, machine configurations, and communication characteristics, instead of focusing on a specific job type or network setting.

Before describing our *Level-Spread* policy, in Sec. 3.2.1, we first use a motivational experiment to demonstrate that the best-performing allocation strategy is different for jobs of different sizes, and we explain why spreading tasks of a job inside a suitable network level can be beneficial to their MPI (Message Passing Interface) communication. We then present our *Level-Spread* in Sec. 3.2.2.

3.2.1 Motivational Experiment

Intuitively, allocating jobs compactly reduces communication times as it leads to small network distances. However, in dragonflies, allocating large jobs compactly (i.e., prioritizing using nodes from the same group when allocating a job) leads to

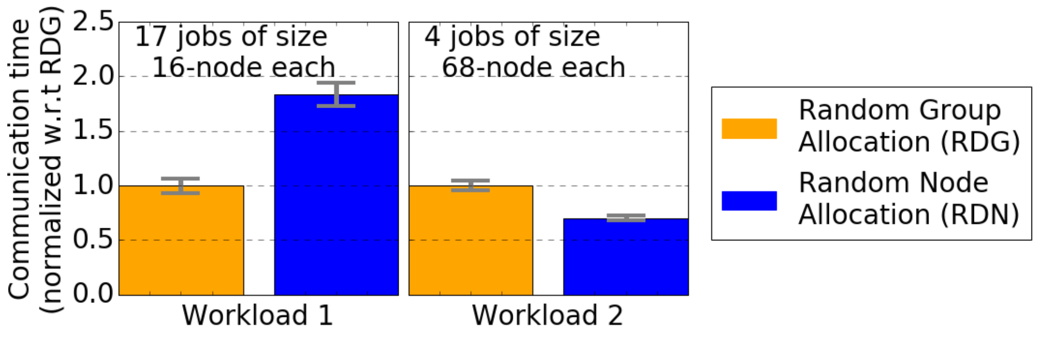


Figure 3·2: We compare random group allocation (RDG), which prioritizes selecting nodes from the same group when allocating a parallel job, and random node allocation (RDN), which selects nodes randomly across the entire network, and we simulate two different workloads. Workload 1 (small jobs) benefits from RDG, whereas workload 2 (large jobs) benefits from RDN.

hot spots on network links. Hence, several studies suggest *random* allocation for dragonfly networks (Yang et al., 2016; Jain et al., 2014; Prisacari et al., 2016).

Our simulation results in Fig. 3·2 confirm this phenomenon. The figure compares the communication time by running two workloads on a 272-node dragonfly machine with 16 nodes per group. With each workload, we compare two allocation policies proposed by Jain et al. (Jain et al., 2014): (1) random group allocation (RDG), which randomly selects a group and allocates the job to all the idle nodes in that group, and repeats this process if more nodes are required; (2) random node allocation (RDN), which randomly selects nodes in the entire machine for a job. Each simulation is further repeated 10 times to reduce the bias due to randomness.

In workload 1, seventeen 16-node jobs are running simultaneously, fully utilizing the dragonfly machine. RDG outperforms RDN as RDG allocates each job to a single group, harnessing group-level locality and avoiding interference on global links. Contrary to workload 1, in workload 2, where four 68-node jobs are running simultaneously on the same machine, RDN reduces the communication time by 30% compared to RDG. Although more packets travel on global links with RDN than with RDG, the RDN policy benefits from a balanced load on the network.

These observations inspire us to design a new allocation policy combining the advantages of both RDG and RDN.

3.2.2 Level-Spread Allocation Policy

Our goal is to minimize the communication time of jobs running on HPC systems with dragonfly topologies through job allocation. Here, a job consists of multiple parallel tasks, and each task is defined as an MPI rank. In contrast to existing dragonfly-specific allocation policies that apply the same allocation strategy (grouping or spreading) for all jobs (Jain et al., 2014), our *Level-Spread policy* selects specific allocation strategies based on the job size along with the machine state.

The principle of our policy is to allocate a job into the smallest network level (router, group, or machine) where that job can fit in and to spread the parallel tasks of the job within that level. Selecting the smallest network level benefits a job by letting it harness node locality and by reducing the communication interference with other jobs. Spreading the tasks within that level balances the network communication on the links, and thus, reduces network hot spots. The specific steps are as follows:

- If a job fits within the available nodes that are connected to a single router, we select *the router with the largest number of idle nodes* and allocate the job there.

- If a job cannot fit within a single router but fits within the available nodes in a single group, we select *the most idle group* and allocate the job there. To further reduce load imbalance on local links in this group, we select nodes connected to different routers in a round-robin manner.

- If a job cannot fit within a single group, we *spread the job throughout the entire network*, where we select nodes in different groups in a round-robin manner.

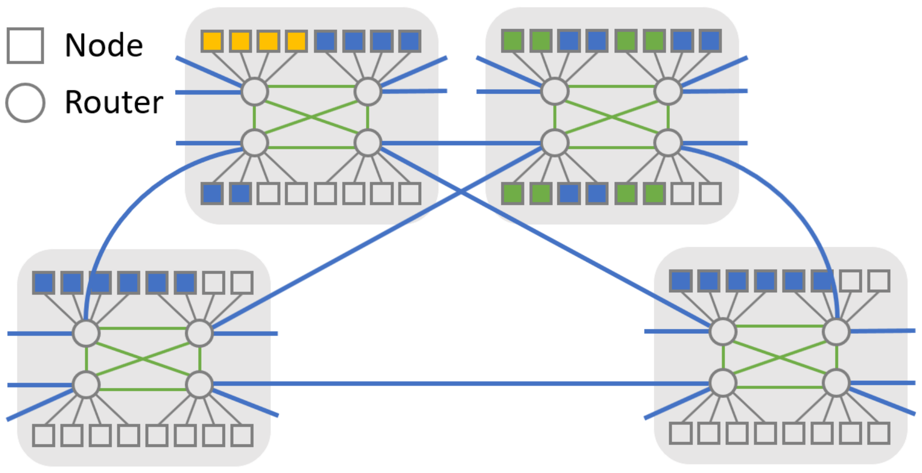


Figure 3·3: Three jobs are allocated to a dragonfly machine ($g = 9$, $a = 4$, $p = 4$) by *Level-Spread*. Only four groups are drawn for simplicity. The first job (orange) is allocated to the nodes connected to a single router. The second job (green) is allocated to different routers in the same group. The third job (blue) is spread to all groups in a round-robin manner.

Following the terminology used by Kim *et al.* (Kim et al., 2008), in this article, p represents the number of nodes connected to a router; a represents the number of routers in a group; g represents the number of groups in the entire dragonfly machine.

Fig. 3·3 illustrates an example allocation for three jobs. The first 4-node job is allocated to a single router. The second 8-node job (green) is spread within the group with the largest number of idle nodes. The third job (blue) is spread throughout the entire machine, occupying the first six available nodes of every group.

In the implementation of our algorithm, we scan through all routers/groups when searching for the router/group with the maximum availability. The time complexity of our algorithm is linear with the number of nodes in the machine, i.e., $\mathrm{O}(a \times g \times p)$, as follows: Allocating jobs that fit to a single router, is performed in $\mathrm{O}(a \times g + p)$ time, where selecting a router is $\mathrm{O}(a \times g)$ and allocating nodes within the router is $\mathrm{O}(p)$. The allocation of the jobs that do not fit in a single router but do fit within a group is performed in $\mathrm{O}(a \times g + a \times p)$. For the remaining jobs, the algorithm simply

Table 3.1: Configurations of simulated dragonfly machines.

Machine Parameters	Values
Processors per node	2
Nodes per router (p)	4
Routers per group (a)	4 or 8
Number of groups (g)	17 or 33
Total number of nodes ($p \times a \times g$)	From 272 to 1056
Local link bandwidth	8 Gbit/s
Global link bandwidth	2 Gbit/s, 8 Gbit/s, or 32 Gbit/s
Routing algorithm	Adaptive routing
Machine utilization level	90% or 70%

scans through all nodes in $O(a \times g \times p)$. Owing to its linear time complexity, our policy can be easily implemented on real dragonfly machines.

3.3 Simulation Methodology

A commonly-used method to examine the performance of dragonfly networks and to explore various network configurations is to run network simulations (Jain et al., 2016; Bhatele et al., 2016; Mubarak et al., 2012; Yang et al., 2016; Prisacari et al., 2014b; Jain et al., 2014; Prisacari et al., 2014a). To evaluate the performance of different job allocation policies on dragonflies, we run network simulations using the Structural Simulation Toolkit (SST) (Rodrigues et al., 2011) with different network configurations and job communication patterns.

3.3.1 Structural Simulation Toolkit (SST)

The SST (Rodrigues et al., 2011) is an open-source architectural framework developed to model and simulate HPC systems. It supports packet-level network simulations and has been verified and used in recent studies (Rodrigues et al., 2011; Wilke et al., 2014; Groves et al., 2016; Hsieh et al., 2011; Underwood et al., 2007). We have extended SST by adding new allocation policies for the dragonfly network: the baseline policies and our *Level-Spread* policy.

3.3.2 Simulated Environments

We simulate dragonfly networks with various configurations, listed in Table 3.1. We select these parameters following previous studies (Prisacari et al., 2014b; Kaplan et al., 2017; Yang et al., 2016). Our dragonfly machines have 16 or 32 ($= p \times a$) nodes per group. Thus, *Level-Spread* will spread jobs that are larger than 16 or 32 nodes to the entire machine. Following the designed structure of dragonfly network in Ref. (Kim et al., 2008), the routers inside each group are connected to each other by local links in an all-to-all fashion.

For message routing, we use the *adaptive routing* algorithm (Kim et al., 2008), which has been shown to provide good performance in dragonfly networks (Kim et al., 2008; Yang et al., 2016; Jain et al., 2014). Based on link congestions derived from the local queue information, *adaptive routing* dynamically chooses between shortest-path routing and *Valiant routing*, which first directs each packet to a randomly-selected intermediate group and then to the destination group.

Different from job allocation, *task mapping* controls the order of task placement onto the processors of the allocated nodes. We use random task mapping to reduce the bias caused by the task mapping process. We assume that each node has two processors where each processor executes one task.

We also explore various machine utilization levels, defined as the number of busy nodes divided by the total number of nodes in the machine. As real HPC systems are heavily utilized, we test utilization levels of 90% and 70%.

3.3.3 Parallel Workloads

We examine both homogeneous and heterogeneous workloads listed in Table 3.2. For each homogeneous workload, the number of jobs is determined by machine size and utilization. For example, for a 1056-node dragonfly machine and 90% utilization

Table 3.2: Parallel workloads in our experiments.

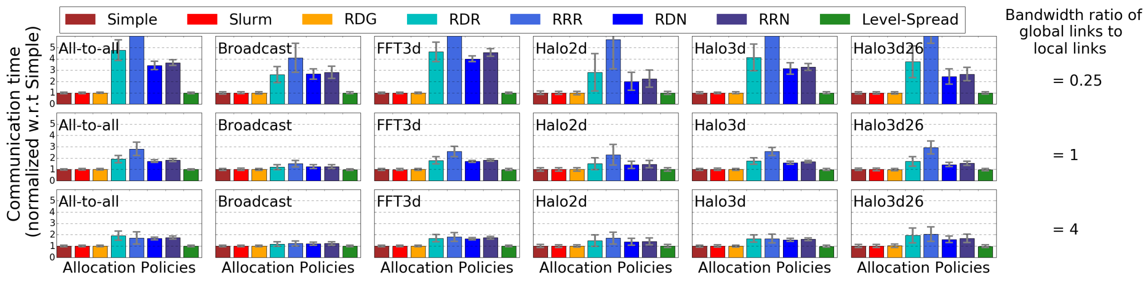
Type	Workload
Homogeneous workloads	Multiple 16-node jobs
	Multiple 64-node jobs
	Four quarter-machine-size jobs
Heterogeneous workloads	16-node jobs and 64-node jobs
	Jobs with randomly-selected sizes

level, the homogeneous workload of 16-node jobs consists of $\lfloor 1056 \times 90\%/16 \rfloor = 59$ such jobs. For each heterogeneous workload, we set the number of 16-node jobs the same as the number of 64-node jobs. To explore a broader range of job sizes and numbers, we also generate random workloads each composed of two types of jobs with randomly-generated sizes (see Sec. 3.4.4).

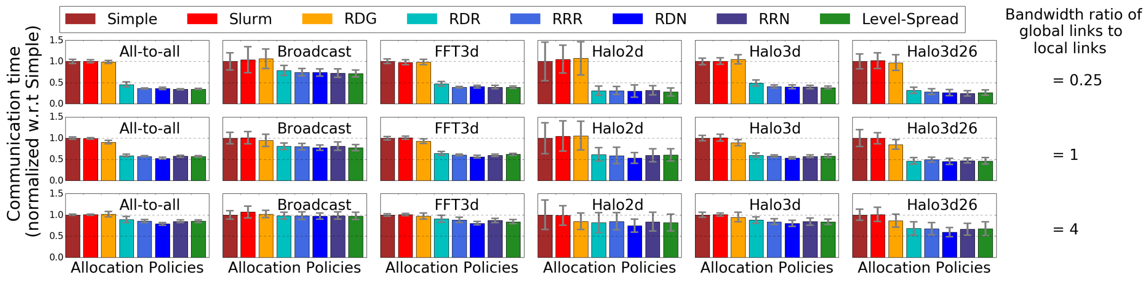
All jobs in our workloads arrive at the same time. The order of jobs to be allocated, which is determined by a scheduling algorithm, influences the outcome of allocation. As allocation is typically performed independent of scheduling, in Sec. 3.4.5 we investigate two job ordering scenarios: small-job first and large-job first.

The communication structure among the tasks of a job defines the job's communication pattern. To explore the influence of job communication patterns on the performance, we use the following six communication patterns, which represent common communication characteristics in parallel HPC applications (Hertel et al., 1995; Plimpton, 1995; Antypas et al., 2008; Sreepathi et al., 2016; Asanovic et al., 2009):

- **All-to-all**: each task sends messages to all other tasks.
- **Broadcast**: one central task broadcasts messages to all other tasks.
- **FFT3d**: the communication pattern of doing 3-D fast Fourier transform.
- **Halo2d**: each task sends messages to its 4 nearest neighbors, forming a communication graph in a 2-D grid.
- **Halo3d**: each task sends messages to its 6 nearest neighbors, forming a communication graph in a 3-D grid.



(a) Workload composed of 16-node jobs. As the jobs fit in a single group, this workload benefits from the grouping-strategy policies (*Level-Spread*, Simple, Slurm, RDG), which, in this case, allocate the tasks of a job into the same group.



(b) Workload composed of 64-node jobs. As the jobs do not fit in a single group, this workload benefits from spreading-strategy policies (*Level-Spread*, RDR, RRR, RDN, RRN), which, in this case, spread tasks of a job into different groups.

Figure 3·4: Communication time of homogeneous workloads on a machine with 16 nodes per group, 17 groups in total, at 90% utilization, and using an application message size of 1 KB. Results are normalized with respect to the Simple allocation policy. Error bars represent the standard deviation. Both (a) and (b) shows reduced difference among these allocations when the global link bandwidth increases relative to the local link bandwidth.

- **Halo3d26**: each task sends messages to its 26 neighbors in 3 dimensions, including 6 nearest neighbors and 20 diagonal neighbors.

In order to focus on the communication overhead due to job allocation, we simulate jobs with only communication without any computation. We also explore the influence of communication intensity by using 1KB and 100KB job message sizes.

To reduce the influence of randomness introduced by task mapping, routing, and allocation policies with randomization, we repeat each experiment ten times.

We explore all the combinations of the machine configurations listed in Table 3.1, workloads listed in Table 3.2, and communication characteristics discussed above. In total, we conduct more than one hundred thousand simulations.

3.3.4 Baseline Allocation Policies

We select the following seven allocation policies (Jain et al., 2014; Slurm, 2016) for dragonfly networks as baselines for comparison:

- **Simple** selects idle nodes by following the node label order, which is defined as follows: the nodes in the first group are labeled as in Figure 3·1 and the labeling continues with the next group in the network following the same rationale. Simple policy allocates jobs in a compact manner in general.

- **Slurm** allocates jobs following the policy for dragonfly implemented in the Slurm Workload Manager (Slurm, 2016). It first attempts to allocate a job to the nodes connected to a single router. The first available router with a sufficient number of idle nodes is chosen, and the nodes connected to that router are allocated following the label order. If there are no routers with a sufficient number of idle nodes, it searches for the router with the fewest number of idle nodes and selects the idle nodes connected to that router following the label order, and repeats this step as necessary. This allocation policy does not utilize the group structure of a dragonfly network.

- **Random Nodes (RDN)** chooses nodes completely randomly from the entire machine.

- **Random Routers (RDR)** randomly selects a router and then selects idle nodes connected to that router following the label order, and repeats this step as necessary.

- **Random Group (RDG)** randomly chooses a group and selects the nodes in that group following the label order, and repeats this step as necessary.

- **Round Robin Nodes (RRN)** starts from the first group, selects the first idle node following the label order in that group, then moves to the next group and repeats the same process as necessary.
- **Round Robin Routers (RRR)** starts from the first group, chooses the first available router following the label order and selects the idle nodes connected to that router following the label order. It then moves to the next group and repeats the same process as necessary.

We also compare our policy with a job-size-aware policy for fat-tree networks, proposed by Jokanovic et al. (Jokanovic et al., 2015), and adapt their policy to dragonflies. Their policy places a virtual boundary dividing the machine into two partitions. When allocating a job whose size is smaller than the number of nodes per group, the policy allocates the job to a group in the first partition that has enough idle nodes. When allocating larger jobs, the policy places the job in the second partition, starting from the last nodes (according to the label order). In this way, this policy reduces system fragmentation as well as interference between small and large jobs. The boundary between two partitions is updated dynamically based on the allocation history. Results for Jokanovic's policy are discussed in Sec. 3.4.6.

Our *Level-Spread* policy distinguishes from the existing policies by combining the grouping and spreading allocation strategy intelligently based on job sizes. Existing policies only rely on either the grouping or the spreading strategy, and cannot take full advantage of the specific hierarchical structure of dragonflies.

3.4 Simulation Results for Level-Spread Policy

In this section, we provide our experimental results and analyze the strengths and weaknesses of the baseline policies as well as our proposed policy. We first display the results from running homogeneous and heterogeneous workloads. Next, we analyze the impact of communication intensity on performance. In Sec. 3.4.4, we examine the performance on a broad range of random workloads. In Sec. 3.4.5, we discuss the influence of scheduling on the performance of *Level-Spread*. In Sec. 3.4.6, we compare *Level-Spread* with a size-aware allocation policy proposed by Jokanovic et al. for fat-tree networks.

We focus on communication time, which is a better metric than throughput to evaluate the network performance because it has a more direct impact on job execution time (Prisacari et al., 2014b). As users generally care about the relative degree of the delay of HPC jobs, we use normalized values in our evaluation.

3.4.1 Homogeneous Workloads

Figure 3·4 shows the average communication time of the homogeneous workloads. The results in each subfigure is normalized with respect to the average communication time using the Simple allocation policy. We simulate workloads that consist of the same communication pattern, for all six communication patterns described in Sec. 3.3.3. Each column of subfigures focuses on a communication pattern, while each row focuses on a different ratio of global link bandwidth to local link bandwidth.

For the 16-node-job workloads in Fig. 3·4(a), because the smallest network level that a single job can fit in is a dragonfly group, *Level-Spread* allocates the tasks of each job within a single group. *Level-Spread* and policies that select nodes compactly (Simple, Slurm, and RDG) outperform the policies that spread the tasks across the entire machine (RDR, RRR, RDN, and RRN) by reducing the communication time by 15% to 89%. The degree of reduction depends significantly on link bandwidth.

Figure 3·4(b) shows the normalized communication time in workloads composed of 64-node jobs. As a 64-node job cannot fit in a single dragonfly group, *Level-Spread* spreads the tasks of each job across the entire machine. These results demonstrate that *Level-Spread* and the policies that spread the tasks (RDR, RRR, RDN, RRN)

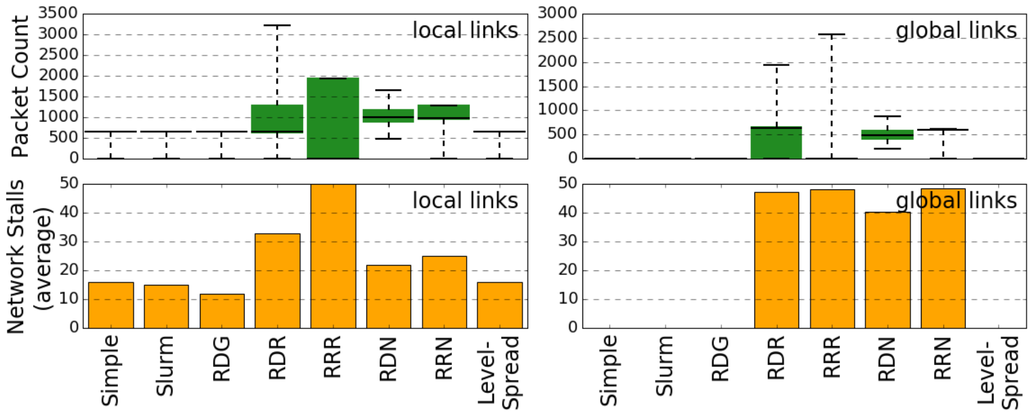


Figure 3·5: Packet count and stalls at the output port to global or local links when running the 16-node-job homogeneous workload.

outperform the other policies that select nodes compactly. When the bandwidth ratio of global-to-local links is 1, which is close to the values in existing dragonfly machines, *Level-Spread* reduces the communication time by 22% to 54% compared to Simple allocation.

By comparing different columns in Fig. 3·4, we see that the performance of the eight allocation policies is consistent for the six communication patterns. By comparing different rows, we see that the increase of global link bandwidth reduces performance difference among allocations.

For all machine parameters and communication patterns we use, we have also examined the impact of machine utilization. For the two utilization levels, 90% and 70%, our results show negligible impact of utilization level on the relative performance of different allocation policies.

We collect network statistics in these simulations to understand the underlying causes of the performance difference. Fig. 3·5 shows the number of packets and stalls on network links when the machine is running the 16-node-job All-to-all workload. The green boxes represent 25% to 75% percentiles, the central line represents median, and the whiskers represent min/max. In this case, the global-to-local link bandwidth

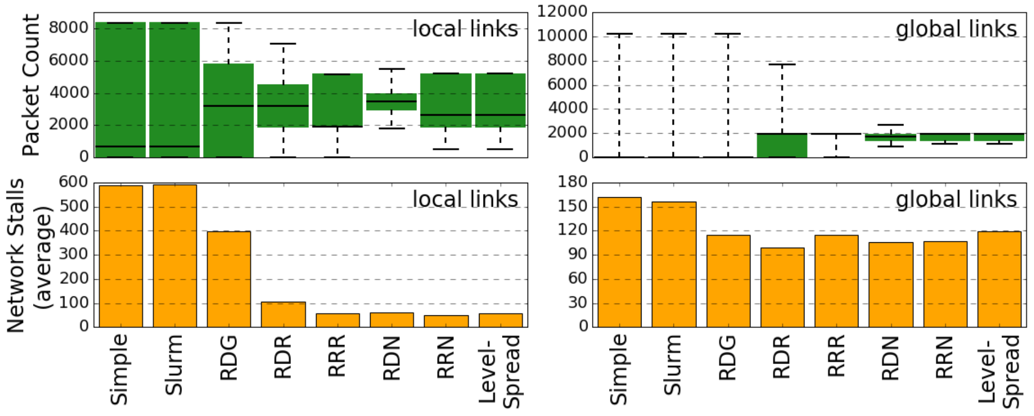
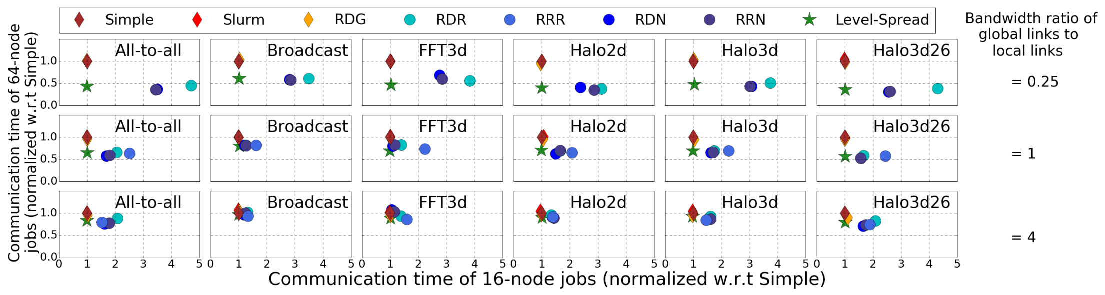


Figure 3·6: Packet count and stalls at the output port to global or local links when running the 64-node-job homogeneous workload.

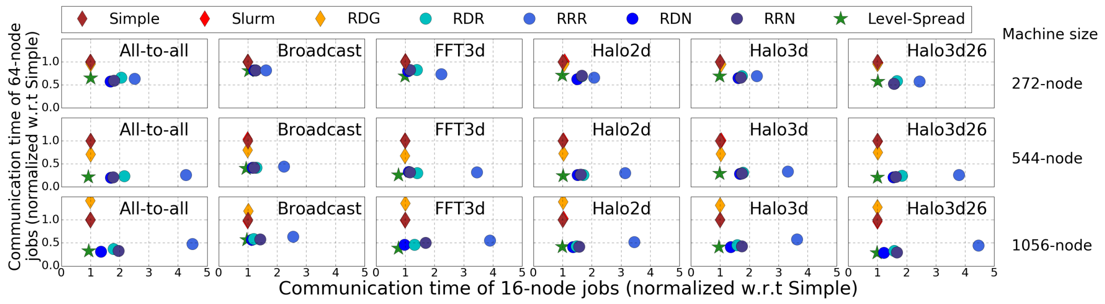
is 1, job message size is 1KB, and the machine has 17 groups and 16 nodes per group. Fig. 3·5 shows that *Level-Spread* and the grouping-strategy policies (Simple, Slurm, RDG) indeed stress the links less than other policies. Consequently, grouping-strategy policies lead to fewer stalls, leading to better performance.

Similarly, Fig. 3·6 shows network statistics when the machine is running the 64-node-job All-to-all workload. *Level-Spread* and the spreading-strategy policies (RDR, RRR, RDN, RRN) lead to more packets on both local and global links in terms of median. However, spreading-strategy policies create smaller variations on packet count, in agreement with our expectation that spreading-strategy policies balance load on the links, and thus, reduce hot spots. Therefore, spreading-strategy policies result in fewer stalls compared to grouping-strategy policies, which explains their better performance.

To examine the performance of running jobs larger than 64-nodes, we also ran homogeneous workloads that consist of four jobs, each of quarter-machine-size. For the four machines ranging from 272 nodes to 1056 nodes, the performance of different allocations on these workloads are very close to Fig. 3·4(b); so we omit these results due to space constraints.



(a) Each row uses a different global-to-local link bandwidth ratio in a machine with 17 groups and 16 nodes per group. Simple, Slurm, and RDG overlap with each other. In the first row, RRR's X-axis position is beyond 5, thus not in the figure.



(b) Each row uses a different machine size. The global-to-local link bandwidth ratio is 1.

Figure 3·7: Communication time of heterogeneous workloads each composed of n small jobs (16-node) and n large jobs (64-node) with a message size of 1KB. The number n is determined by the target utilization level of 90%. In each subfigure, a point represents the results from running the heterogeneous workload using a specific allocation. The X-axis of the point represents the average communication time of the small jobs, and the Y-axis represents that of the large jobs. The star corresponds to the *Level-Spread* allocation policy; the diamonds correspond to the grouping-strategy policies; the circles correspond to spreading-strategy policies. Values are normalized with respect to the Simple allocation policy in each subfigure.

3.4.2 Heterogeneous Workloads

We evaluate the benefits of size awareness in our *Level-Spread* allocation policy using heterogeneous workloads that consist of jobs with different sizes. Fig. 3·7 shows the communication time of jobs in heterogeneous workloads composed of n 16-node and n 64-node jobs, where n is determined by the machine size and the target utilization level. We simulate six such workloads, each using jobs from one of the six communication patterns. We allocate small jobs first; the impact of the allocation order is studied in Sec. 3.4.5.

In every subfigure, the X-axis of a point represents the average communication time of the 16-node jobs, and the Y-axis represents that of the 64-node jobs. Values are normalized with respect to the Simple allocation policy in each subfigure. The grouping-strategy policies (Simple, Slurm, RDG) are marked with warm-colored diamonds, and the spreading-strategy policies (RDR, RRR, RDN, RRN) are marked with cool-colored circles. *Level-Spread* is marked with a green star.

All subfigures in Fig. 3·7 show that *Level-Spread* lies in the bottom-left part, which demonstrates that *Level-Spread* reduces the communication time for both small and large jobs. The X-axis of *Level-Spread* coincides with the X-axis of diamonds, showing that the communication time of the 16-node jobs allocated by *Level-Spread* remains similar to the Simple, Slurm, and RDG policies that allocate jobs compactly. Meanwhile, the Y-axis of *Level-Spread* coincides with the Y-axis of circles in general, showing that the communication time of the 64-node jobs allocated by the *Level-Spread* remains similar to the RDR, RRR, RDN, and RRN policies that spread the jobs across the machine.

On the other hand, the diamonds in Fig. 3·7 lie in the top-left part of each subfigure, showing that these grouping-strategy policies do not work well for the large jobs. Circles lie in the bottom-right part of each subfigure, showing that these spreading-

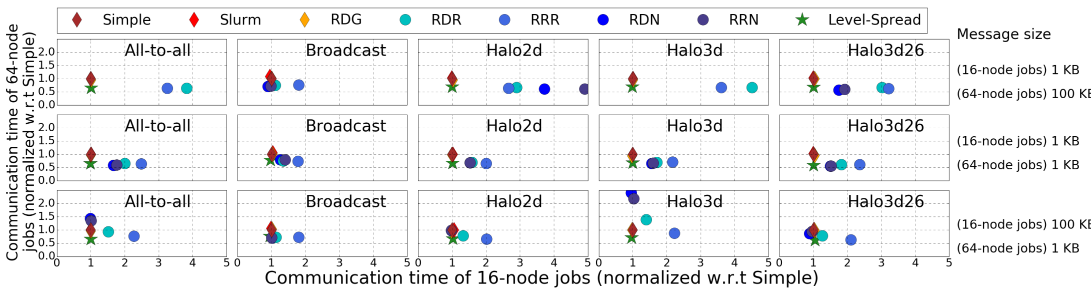


Figure 3·8: Varying the communication intensity in terms of message size of the jobs. Here, we simulate a heterogeneous workload composed of three 16-node jobs and three 64-node jobs on a 272-node machine. We repeat the simulations with six communication patterns and different message sizes. From top to bottom, we gradually increase the ratio of communication intensity between the 16-node jobs and the 64-node jobs. In the first row, some circles are beyond the range of the X-axis and thus not displayed.

strategy policies do not work well for the small jobs.

Comparing the columns in Fig. 3·7, we observe the consistency of the benefits of our *Level-Spread* policy across different communication patterns. Different rows in Fig. 3·7(a) illustrate the impact of global link bandwidth on performance. Increasing the global link bandwidth reduces the difference among the policies but does not change their ranking.

Different rows in Fig. 3·7(b) explore the impact of machine size on performance. *Level-Spread* consistently outperforms the others in all three machine sizes and all communication patterns. Increasing the machine size magnifies the difference among allocations. In the case of a 1056-node machine (33 groups), *Level-Spread* policy reduces the communication time of the 16-node jobs by 32% (Broadcast) to 64% (All-to-all).

In these simulations, changing machine utilization level from 90% to 70% has negligible influence on the relative performance of the eight allocation policies, so we omit the results for 70% utilization.

3.4.3 Impact of Communication Intensity

To explore the influence of communication intensity on the performance of different allocation policies, for all our workloads and parameter sets listed, we run simulations using 1KB and 100KB job message sizes. We find that as long as the communication intensity is homogeneous for all jobs in the workload, the impact of job message size on the relative performance of different allocation policies is negligible. Simultaneously increasing message size of all jobs from 1KB to 100KB only increases the communication time of every job by approximately 100 folds. Due to the similarity of these results to Fig. 3·4 and Fig. 3·7, we do not depict these results.

In Fig. 3·8, we explore the cases where the communication intensity of 16-node jobs is different from the intensity of 64-node jobs. We experiment on a 272-node machine and simulate different combinations of message sizes.

From top to bottom in Fig. 3·8, we gradually increase the ratio of communication intensity between the 16-node jobs and the 64-node jobs. These plots demonstrate a clear trend that increasing the communication intensity of the large (64-node) jobs worsens the performance of RDR, RRR, RDN, and RRN policies on the small (16-node) jobs. This is because when the tasks of the small jobs are spread into many groups, the intensive communication among the tasks of the large jobs drastically delays the communication of the small jobs. Conversely, increasing the communication intensity of the small jobs significantly worsens the performance of RDR, RRR, RDN, and RRN policies on the large jobs. This is because in these spreading policies, the communication among the tasks of the large jobs are delayed by the communication-heavy small jobs. However, for Simple, Slurm, RDG, and *Level-Spread* policies, the small jobs are less affected by the interference with the large jobs because the small jobs by these policies do not stress global links. These results agree with the findings of Yang *et al.* (Yang et al., 2016).

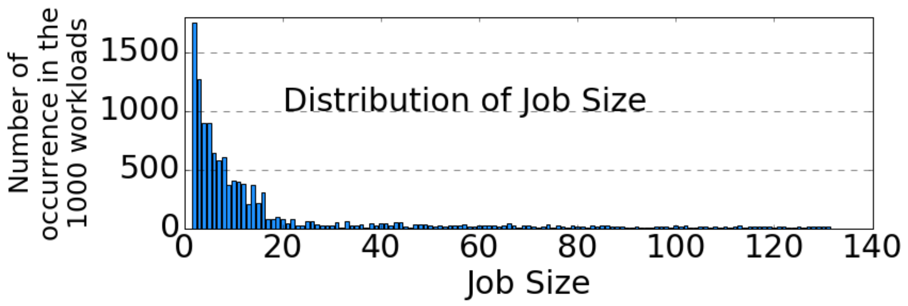


Figure 3·9: Counted occurrence of job sizes in the 1000 randomly generated workloads used in Sec. 3.4.4.

The results in Fig. 3·8 demonstrate the effectiveness of *Level-Spread* in various communication intensities. We have also verified that these conclusions on communication intensities are valid for different machine parameters listed in Table 3.1.

3.4.4 Mixed Job Sizes and Communication Patterns

In Sec. 3.4.2, we have examined the performance of different allocation policies using workloads composed of 16-node jobs and 64-node jobs. To verify that our results can be generalized to other job sizes and mixed communication patterns, we use a broader set of 1000 randomly generated workloads, each composed of two types of jobs (small and large). The following simulations use a 272-node machine with 16 nodes per group (4 nodes per router and 4 routers per group) and a global-to-local links bandwidth ratio of 1. Machine utilization level is not fixed and depends on the generated workload.

To generate the workloads, we first randomly select an integer between 17 and 136 (= 272/2) as the size of the large jobs. We choose the upper limit as 136-node because an individual job is commonly restricted from occupying more than half of a machine. Next, a random integer between 2 and 16 is chosen as the size of small jobs. Then, the number of the small jobs and the number (which can be different) of the large jobs are selected randomly under the restriction of machine size. Fig. 3·9 shows the overall

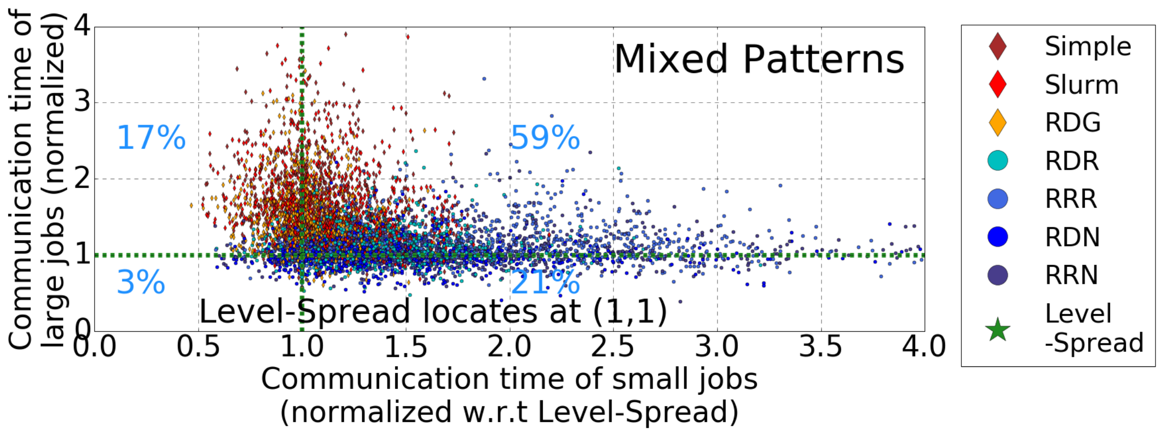


Figure 3·10: Results from 1000 randomly generated workloads with mixed job sizes and communication patterns. Each point represents one allocation policy in one workload. Values are normalized with respect to *Level-Spread*.

distribution of job sizes in these 1000 random workloads, which qualitatively matches the distribution of job sizes in real HPC systems (e.g., (Feitelson et al., 2014)). The communication pattern for each of the two types of jobs is randomly selected. In these simulations, we allocate small jobs before the large jobs in a given workload, and start running all jobs simultaneously. All jobs use a message size of 1KB.

In Fig. 3·10, the X-/Y-axis of each point represents the average communication time of the small/large jobs in one workload using one allocation policy. For each workload, the values are normalized with respect to the average communication time achieved using the *Level-Spread* policy. The dashed green lines split the graph into four parts, and a blue number shows the percentage of points in each part.

Fig. 3·10 shows that the grouping-strategy allocation policies (Simple, Slurm, and RDG) benefit the small jobs over the large jobs. Conversely, the spreading-strategy policies (RDR, RRR, RDN, RRN) benefit the large jobs at a cost of higher small-job communication overhead. *Level-Spread*, located at coordinate (1,1), combines the advantages of both grouping and spreading. Only in 3% of all cases, a baseline policy is strictly better than *Level-Spread*, i.e., for both the small and the large jobs. Meanwhile, in 59% of all cases, *Level-Spread* is strictly better than the baselines. In the best case, *Level-Spread* reduces communication time by 71%. Averaging over both small and large jobs and over all 1000 workloads, *Level-Spread* reduces communication time by 16%.

In addition, we run random-workload simulations where only a single communication pattern is used. For each communication pattern, we generate 100 workloads composed of small and large jobs. Our results show that for All-to-all, Halo3d, Halo3d26, Halo2d and Broadcast workloads the percentage of cases where a baseline policy is strictly better than *Level-Spread* is 1%, 1%, 3%, 4%, and 8%, respectively.

3.4.5 Influence of Scheduling on the *Level-Spread* Policy

In HPC systems, the order of allocating pending jobs is decided by a job scheduler, and this order affects where the jobs are allocated. For our *Level-Spread* policy, the scheduled order of pending jobs may influence the performance due to the job-size-awareness of *Level-Spread*. For example, assume a 15-node job and a 50-node job are pending on an empty 272-node dragonfly system whose nodes per group is 16. Using *Level-Spread*, if the 50-node job is scheduled first, it will be spread into different groups, occupying 2 or 3 nodes in each group. In this case, the 15-node job won't be able to fit in any group and will be also spread, converging to the RRN policy.

To evaluate the performance of *Level-Spread* in an adverse scheduling decision, we generate 1000 random workloads composed of two types of jobs similar to Sec. 3.4.4, but this time, we schedule the large jobs before the small jobs. To clarify, we allocate all jobs following the scheduled order and then start running them simultaneously. Fig. 3·11 shows that prioritizing large jobs in scheduling slightly moves all the points of baseline policies toward the left compared to Fig. 3·10, making the performance of *Level-Spread* closer to the four spreading-strategy allocations. This shows that with an adverse scheduling decision where large jobs are scheduled first, *Level-Spread* performs at least as well as the baselines.

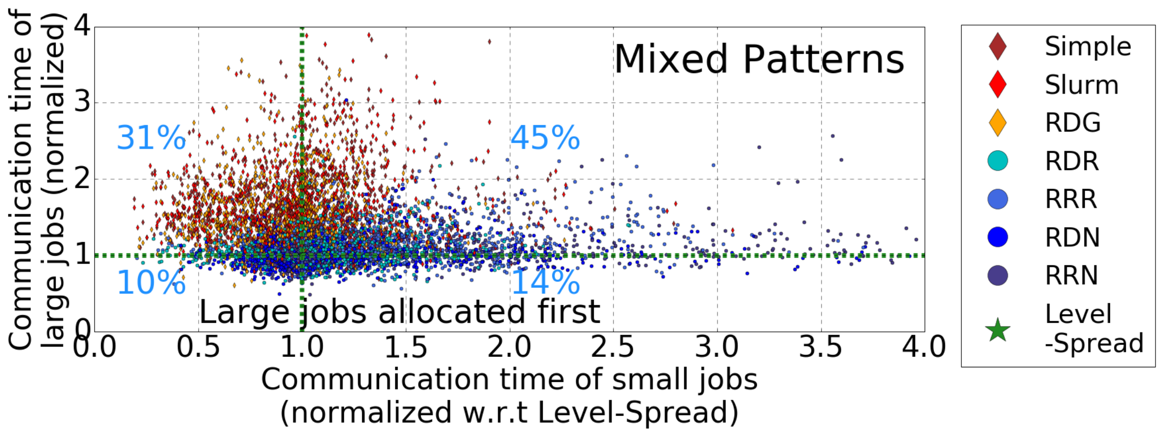


Figure 3·11: In an adverse scheduling decision where the allocation of large jobs (jobs that cannot fit in a single group) are prioritized, *Level-Spread* at least performs as well as the baselines.

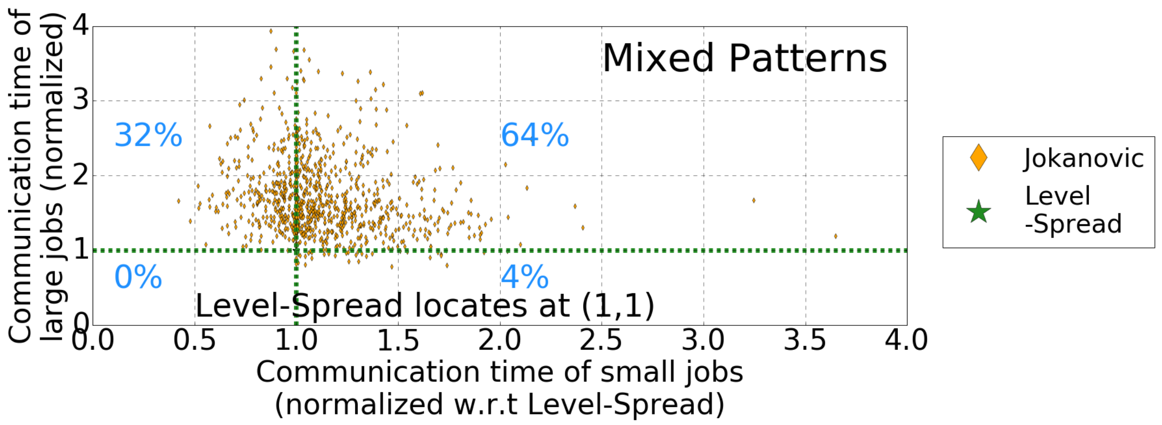


Figure 3·12: Comparing *Level-Spread* policy with Jokanovic's policy using 1000 random workloads with mixed job sizes and communication patterns.

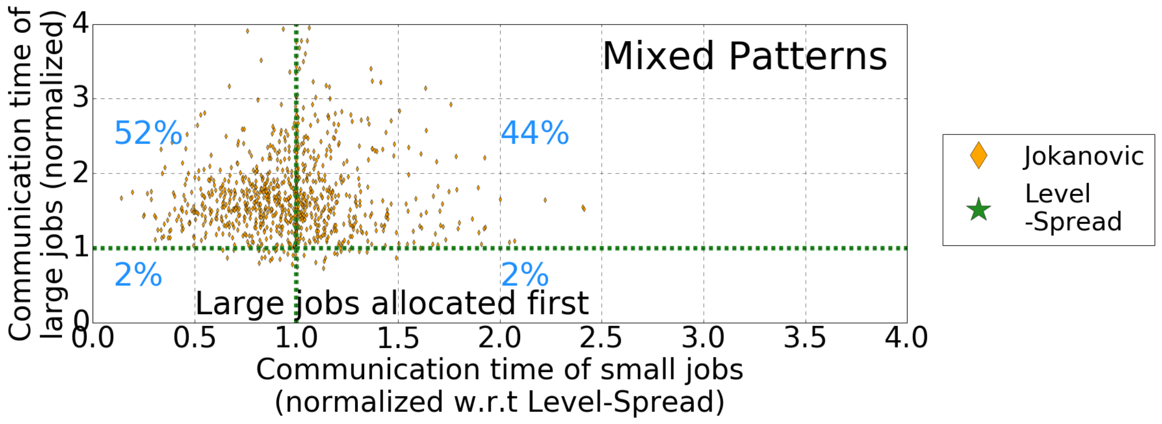


Figure 3·13: Comparing *Level-Spread* policy with Jokanovic's policy using 1000 random workloads. Here, large jobs are allocated prior to small jobs.

3.4.6 Comparison with Jokanovic's Allocation Policy

In the previous sections, we compare *Level-Spread* with state-of-the-art policies for dragonfly networks. There are also allocation policies for other network topologies such as Jokanovic's policy (Jokanovic et al., 2015) for fat-tree networks (see Sec. 3.3.4). To compare *Level-Spread* with Jokanovic's policy, we simulate 1000 random workloads similar to Sec. 3.4.4 (parameters are kept the same). Fig. 3·12 shows that in 64% of the workloads, *Level-Spread* performs strictly better than Jokanovic's policy (i.e., for both small jobs and large jobs). Jokanovic's policy does not perform strictly better than *Level-Spread* in any of these workloads. While Jokanovic's policy is good for small jobs, it is significantly worse than *Level-Spread* for large jobs.

The reason why Jokanovic's policy does not perform well in dragonfly networks is that dragonflies are not very sensitive to system fragmentation, owing to its low diameter and the all-to-all inter-group connections. As discussed in Sec. 3.2.1, dragonfly networks benefit from a more balanced network traffic when we spread large jobs. Therefore, spreading large jobs, as done by *Level-Spread*, gives better performance than Jokanovic's policy, which groups large jobs together.

Similar to Sec. 3.4.5, we also run simulations on 1000 random workloads where the large jobs are scheduled prior to small jobs. Fig. 3·13 demonstrates that even with this adverse scheduling decision for *Level-Spread*, our *Level-Spread* policy continues to outperform Jokanovic's policy.

3.5 Quantifying Network Congestion Using Hardware Performance Counters

It has been commonly reported that network congestion is a major cause of performance degradation in HPC systems (Bhatele et al., 2013a; Smith et al., 2016; Bhatele et al., 2020), leading to extention on job execution time 6X longer than the optimal(Chunduri et al., 2017). Although performance degradation caused by congestion has been commonly observed, it is not well understood how that impact differs from application to application. Which network metrics could indicate network congestion and performance degradation is also unclear. Understanding the behavior of network metrics and application performance under network congestion on large HPC systems will be helpful in developing strategies to reduce congestion and improve the performance of HPC systems.

The contributions of this work are listed as follow:

- In a dragonfly-network system, we quantify the impact of network congestion on the performance of various applications. We find that while applications with intensive MPI operations suffer from more than 40% extension in their execution times under network congestion, the applications with less intensive MPI operations are negligibly affected.

- We find that applications are more impacted by congestor on nearby nodes with shared routers, and are less impacted by congestor on nodes without shared routers. This suggests that a compact job allocation strategy is better than a non-compact

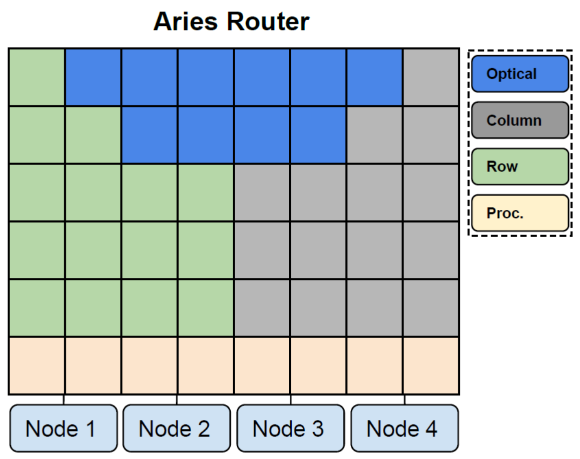


Figure 3·14: Aries router architecture in a dragonfly network.

strategy because sharing routers among different jobs is more common in a non-compact allocation strategy.

- We show that a *stall-to-flit ratio* metric derived from Aries network tiles counters is positively correlated with performance degradation and indicative of network congestion.

In the following, we first provide background on the Aries network router. Then, we introduce our network metrics derived from Aries counters. After that, we evaluate the value of these metrics in revealing network congestion.

3.5.1 Aries Network Router

Aries is one of the latest HPC network architectures (Alverson et al., 2012). Aries network features a dragonfly topology (Kim et al., 2008), where multiple routers are connected by row/column links to form a virtual high-radix router (called a "group"), and different groups are connected by optical links in an all-to-all manner, giving the network a low-diameter property, where the shortest path between any two nodes is only a few hops away.

Figure 3·14 shows the 48 tiles of an Aries router in a Cray XC40 system. The blue tiles include the optical links connecting different groups; the green and grey tiles include the electrical links connecting routers within a group; and the yellow tiles include links to the four nodes connected to this router. In the following, we call the 8 yellow tiles as processor tiles (**ptiles**); and we call the other 40 as network tiles (**ntiles**).

3.5.2 Network Metrics

In each router, Aries hardware counters collect various types of network transmission statistics (Cray Inc., 2018), including the number of flits/packets travelling on links and the number of stalls that represent wasted cycles due to network congestion.

In this work, we use a *stall-to-flit ratio* metric derived from ntile counters. As the number of network stalls represents the number of wasted cycles in transmitting flits from one router to the buffer of another router, we expect the stall/flit ratio to be an indicator of network congestion. For each router, we define the ntile stall/flit ratio as:

$$\begin{aligned}&\text{Ntile Stall/Flit Ratio}\\&= \text{Avg}_{r\in 0..4, c\in 0..7}\frac{\text{N\_STALL\_}r\text{\_}c}{\sum_{v\in 0..7}\text{N\_FLIT\_}r\text{\_}c\text{\_}v}\end{aligned}$$

Here, N_FLIT_r_c_v is the number of incoming flits per second to the v-th virtual channel of the r-th row, c-th column network tile. N_STALL_r_c is the number of stalls per second in all virtual channels on that ntile. As the stalls and flits collected from a specific ntile cannot be attributed to a certain node, we take an average over all the 40 ntiles (represented as "Avg") and use it as the ntile stall/flit ratio of the router. Because the 40 ntiles are the first 5 rows and all 8 columns in Fig. 3·14, the metric takes the average for $r \in 0..4$, and $c \in 0..7$.

In comparison to ntile counters, we also analyze ptile flits per second collected

Table 3.3: Aries network counters used in this work (Cray Inc., 2018).

Abbreviation	Full counter name
N_STALL_r_c	AR_RTR_r_c_INQ_PRF_ROWBUS_STALL_CNT
N_FLIT_r_c_v	AR_RTR_r_c_INQ_PRF_INCOMING_FLIT_VCv
P_REQ_STALL_n	AR_NL_PRF_REQ_PTILES_TO_NIC_n_STALLED
P_REQ_FLIT_n	AR_NL_PRF_REQ_PTILES_TO_NIC_n_FLITS
P_RSP_STALL_n	AR_NL_PRF_RSP_PTILES_TO_NIC_n_STALLED
P_RSP_FLIT_n	AR_NL_PRF_RSP_PTILES_TO_NIC_n_FLITS

by P_REQ_FLIT_n and P_RSP_FLIT_n, which are request and response flits received by a node, respectively. In this paper, we always take the sum of these two metrics when we refer to ptile flit-per-second. Similarly, we refer to the sum of P_REQ_STALL_n and P_RSP_STALL_n as the ptile stalls per second. In these metrics, $n \in 0..3$ corresponds to the four nodes connected with this router. Thus, ptile counters specify the contribution from a certain node. The full names of the counters we mentioned are listed in Table 3.3. The original counters record values cumulatively, so we take a rolling difference to estimate instantaneous values.

In addition, when we calculate stall/flit ratio, we ignore the samples where stall-per-second is smaller than a threshold. This is because when both the stall and the flit number in a second are too small, the stall/flit ratio could occasionally surge while it does not reflect influential congestion. We set the threshold as the median stall-per-second value of data taken over a three-month period from the entire system. For electrical link ntiles and optical link ntiles, the thresholds are 5410794 and 933, respectively.

3.6 Experiments for Quantifying Network Congestion

3.6.1 Design of Experiments

We conduct experiments on Cori, which is a 12k-node Cray XC40 system located at the Lawrence Berkeley National Laboratory, USA. On Cori, network counter data are collected and managed by the Lightweight Distributed Metric Service (LDMS) tool (Agelastos et al., 2014). LDMS has been continuously running on Cori and collecting counter data for years for every node. The data collection rate is once per second.

To characterize job execution performance, we experiment with the following real-world and benchmark applications:

- **Graph500**. We run breadth-first search (BFS) and single-source shortest path (SSSP) from Graph500, which are representative graph computation kernels (Murphy et al., 2010).

- **HACC**. The Hardware Accelerated Cosmology Code framework uses gravitational N-body techniques to simulate the formation of structure in an expanding universe (Heitmann et al., 2019).

- **HPCG**. The High Performance Conjugate Gradient benchmark models the computational and data access patterns of real-world applications that contain operations such as sparse matrix-vector multiplication (Dongarra et al., 2016).

- **LAMMPS**. The Large-scale Atomic/Molecular Massively Parallel Simulator is a classical molecular dynamics simulator for modeling solid-state materials and soft matter (Plimpton, 1995). Our experiments use the `in.vacf.2d` input from the package (Sandia Corporation, 2014) and we have only adjusted the number of steps for our experiments.

- **MILC**. The MIMD Lattice Computation performs quantum chromodynamics simulations. Our experiments use the `su3_rmd` application from MILC (Bauer et al., 2012). Its communication is characterized by overlapping nearest neighbor exchanges in 4 dimensions followed by small message Allreduce.

- **miniAMR**. This mini-application applies adaptive mesh refinement on an Eulerian mesh (Heroux et al., 2009).

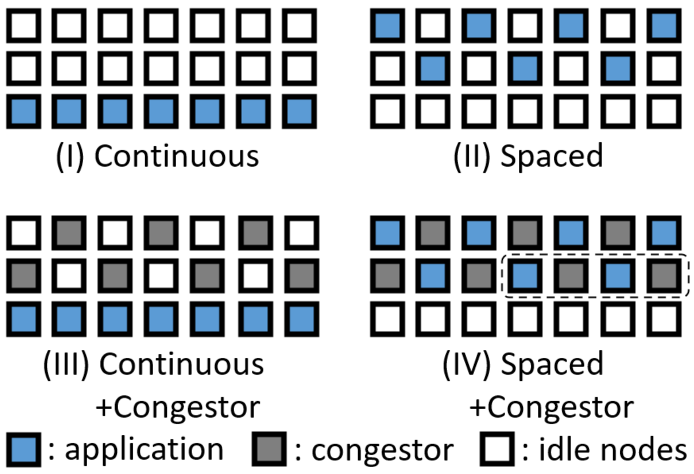


Figure 3·15: The four experimental settings. Each square is a node. Blue squares run a parallel application. Grey squares run the GPCNeT congestor. White ones are idle.

- **miniMD**. This molecular dynamics mini-application is developed for testing new designs on HPC systems (Heroux et al., 2009).

- **QMCPACK**. This is a many-body ab initio quantum Monte Carlo code for computing the electronic structure of atoms, molecules, and solids (Kim et al., 2018). Our experiments use the `simple-H2O` input from the package (Kim et al., 2014) and we have only adjusted the number of steps in the input.

To create network congestion on HPC systems in a controlled way, we use the Global Performance and Congestion Network Tests (GPCNeT) (Chunduri et al., 2019), which is a new tool to inject network congestion and benchmark communication performance. When launched on a group of nodes, GPCNeT runs congestor kernels on 80% of nodes, and the other 20% runs a canary test in a *random-ring* or *allreduce* communication pattern (Chunduri et al., 2019) to evaluate the impact of the congestor kernel. Our experiments run the RMA Broadcast congestor kernel. By comparing running the canary test in isolation with running the canary test together with congestor kernels, GPCNeT reports the intensity of congestion by the following impact factor metrics: bandwidth ratio, latency ratio, and allreduce bandwidth ratio. For example, bandwidth ratio represents the canary test's effective bandwidth when running with congestor, divided by the bandwidth when running in isolation.

We quantify the impact of network congestion on applications by comparing the execution time of the applications when running them with or without congestors. We also differentiate between *endpoint congestion* and *intermediate congestion*. Endpoint congestion refers to the congestion generated by traffic from other nodes that share the same routers as our application. Intermediate congestion refers to the congestion caused by traffic not from nodes sharing the same routers but from intermediate links. We design experiments as follows.

Assume we are going to run an application on N nodes (we use $N = 7$ in Fig. 3·15 as an example, and we actually experiment with $N = 64$), then we reserve $3N$ nodes from the system. Most of these $3N$ nodes are groups of consecutive nodes as the Slurm scheduler on Cori is configured to prioritize reserving consecutive nodes. We have four experimental settings shown in Fig. 3·15 and described below:

- In Setting I ("Continuous"), we run the application on N nodes continuously selected from the list (shown in blue), and the other $2N$ nodes are left idle.
- In Setting II ("Spaced"), we run the application on N nodes selected by choosing every other node from the list.
- In Setting III ("Continuous+Congestor"), besides the application in a "continuous" way, we simultaneously run GPCNeT on another N nodes selected in a "spaced" manner (shown in grey). In this case, the application is mostly affected by intermediate congestion because the majority of blue nodes do not share routers with grey nodes.
- In Setting IV ("Spaced+Congestor"), the nodes for the application and the nodes

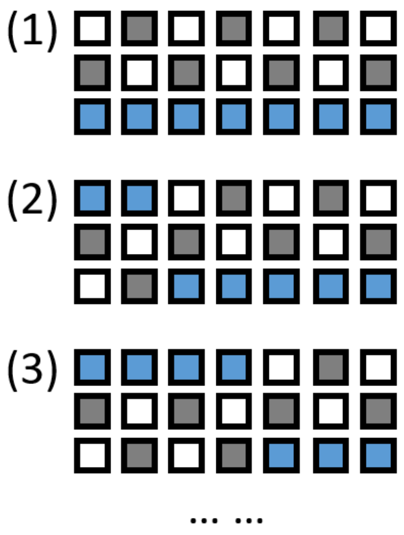


Figure 3·16: To mitigate variations from background traffic, we repeat experiments with the placement of application/congestor rotationally shifted (first three shifts for Setting III are drawn).

for the congestor are interleaved. In this case, the application is also affected by endpoint congestion because sharing router among application nodes and congestor nodes is common. As an example, assume the dashed line shows the four nodes connected to the same router, then, the two grey nodes create endpoint congestion on the other two blue nodes. Although every four nodes are not always connected to the same router, because Cori's scheduler prioritizes allocating contiguous nodes for us, nodes sharing the same router are common.

In our experiments, we always run an application on 64 nodes, and the congestor also occupies 64 nodes. We did not experiment with larger workloads to avoid too much impact on other users in a production system. All experiments run on Haswell nodes and use all 32 cores of each node. The same inputs are used during different runs of an application.

Since the experiments are done in a production system, network traffic generated by jobs from other users may go through the routers we use. To reduce the impact of

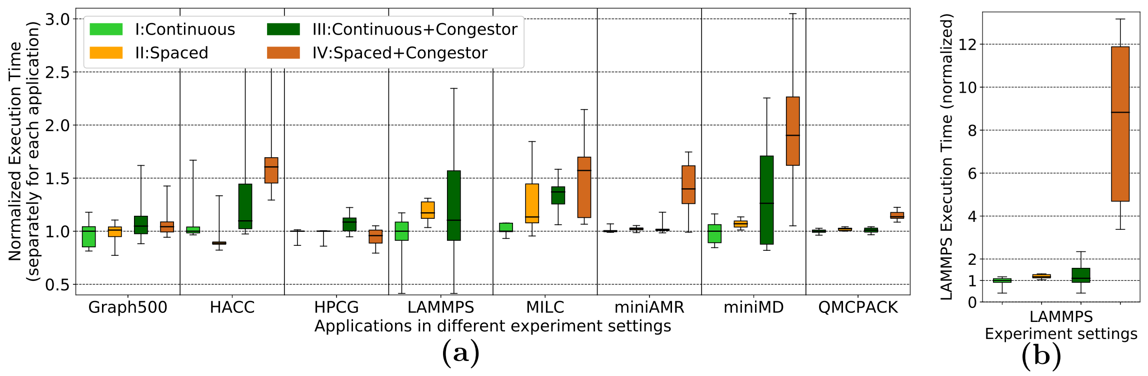


Figure 3·17: Normalized application execution time under four experimental settings. Normalization is done separately for each application. Each bar summarizes the 10 runs for an application. Errorbars are min/max; edges of box are the first and third quartiles; middle line is the median. Setting IV of LAMMPS exceeds the range and it is separately drawn in (b).

this background traffic, we repeat each setting 10 times by rotationally shifting the application's and congestor's placement, as illustrated in Fig 3·16. Each shift rotates 1/10 of the node list length.

In addition to the experiments discussed above, we also experiment with GPCNeT alone without our applications, and the details are discussed in Section 3.6.4.

In the following, we first analyze the impact of network congestion on applications in Section 3.6.2. Next, we show the correlation between network metrics and application execution time in Section 3.6.3. Then, we show that ntile stall/flit ratio is correlated with network congestion intensity in Section 3.6.4.

3.6.2 Impact of Network Congestion on Applications

Figure 3·17 summarizes the impact of network congestion on applications. The execution times are normalized separately for each application with regard to the median value in Setting I. Because LAMMPS's values exceed the range of Fig. 3·17(a), we also draw them separately in Fig. 3·17(b).

These results demonstrate that network congestion has diverse impact on applications. Some applications such as Graph500 and HPCG are negligibly affected by congestion, where the median execution time difference between with and without congestor is less than 10%. On the other hand, applications including HACC, LAMMPS, MILC, miniAMR, and miniMD are significantly affected by congestion, where the median performance with endpoint congestion (Setting IV) is 0.4X to 7X higher than the performance without congestion (Setting II). QMCPACK shows a medium-level impact from congestion, where endpoint congestion extends the median execution time by 11%. The slightly shorter execution time in Setting II of HACC and Setting IV of HPCG should be due to variations caused by background traffic.

To understand why applications are impacted differently, we use CrayPat (Cray Inc., 2020) to profile the MPI characteristics of the applications (in separate runs without congestor). Table 3.4 shows the percentage of time spent on certain MPI operations, and Table 3.5 shows the aggregate bytes transferred per second, average message size, and MPI call counts.

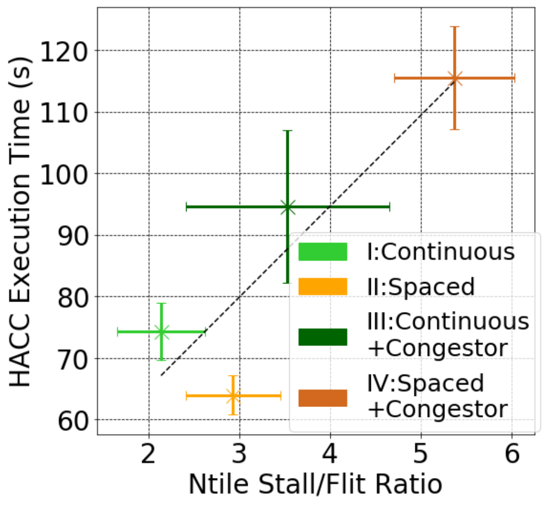
From Table 3.4, we see that the applications impacted more by congestion, including HACC, LAMMPS, MILC, miniAMR, and miniMD, share a common feature of more time spent on MPI operations. On the contrary, HPCG and QMCPACK have only 3.7% and 6.0% time on MPI operations, respectively. In addition, more MPI collective operations (such as MPI_Allreduce, MPI_Bcast) implies more intensive communication, making the application sensitive to congestion. Therefore, as LAMMPS spends 25.4% time on MPI_Allreduce, larger than any other applications, its execution time is extended by more than 7X in Setting IV. Similarly, QMCPACK has 5.4% time on MPI_Allreduce, higher than the 0.2% from HPCG, which explains why QMCPACK is impacted more by congestion than HPCG. On the other hand, Graph500 has only 2.3% time on MPI_Allreduce, and less than 8% time on all other MPI calls except for MPI_Test, which explains why it is only slightly affected by

Table 3.4: Application MPI profiles collected by CrayPat. "MPI Operation" shows the percentage of execution time spent on MPI operations, and the MPI call breakdown is shown in other columns. "MPI_(other)" is the sum of other MPI calls not specified here. Applications with more time spent on MPI operations, especially MPI collective operations (MPI_Allreduce, MPI_Bcast, MPI_Barrier, etc.), are impacted more by network congestion than applications with less intensive MPI operations.

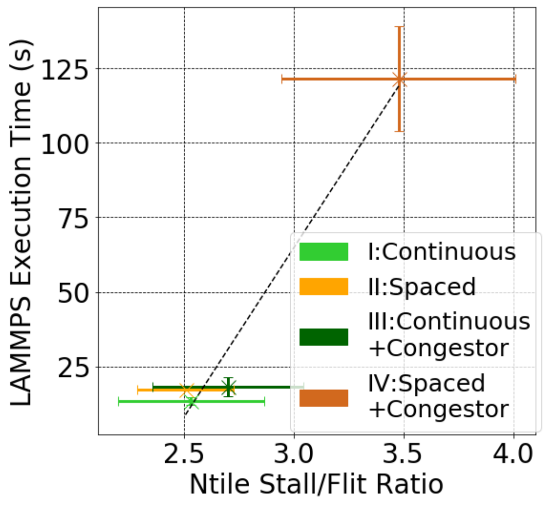
Application	MPI Operation	MPI _Allreduce	MPI_Sendrecv (or Send, Isend)	MPI _Bcast	MPI_Test (or Testany)	MPI_Wait (or Waitall)	MPI _Barrier	MPI_(other)
Graph500	31.4%	2.3%	2.6%	0.2%	21.3%	<0.1%	4.4%	0.6%
HACC	67.1%	<0.1%	0.2%	0	0	66.2%	0	0.7%
HPCG	3.7%	0.2%	2.1%	0	0	1.2%	0	0.2%
LAMMPS	47.3%	25.4%	8.6%	<0.1%	0	12.2%	<0.1%	1.1%
MILC	61.9%	1.9%	0.6%	<0.1%	0	58.5%	<0.1%	0.9%
miniAMR	26.8%	9.2%	0.5%	<0.1%	0	14.2%	0	2.9%
miniMD	83.4%	0.5%	82.5%	0	0	0	0.2%	0.2%
QMCPACK	6.0%	5.4%	<0.1%	<0.1%	<0.1%	<0.1%	0.5%	0.1%

Table 3.5: Execution time, aggregate data transfer rate, average message size, and MPI call counts collected by CrayPat. Columns starting with “MPI_” are breakdown of “MPI Call”. Non-dominant MPI call types are not listed. “Exec Time” is the median of the 10 runs in Setting I. “Agg Data Trans Rate” shows the aggregate bytes of data transferred per second.

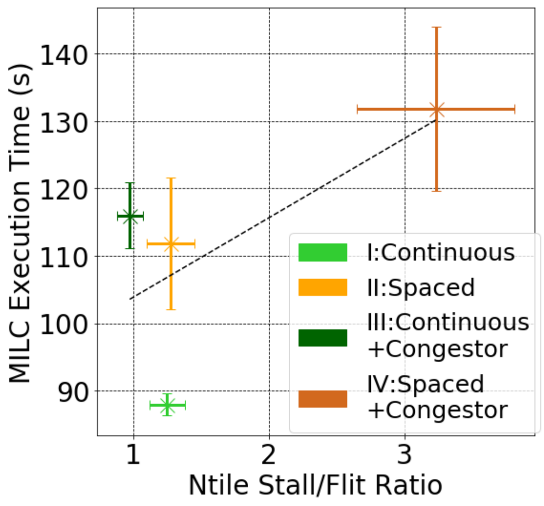
Application	Exec Time	Agg. Data Trans. Rate	Avg Msg Size	MPI Call	MPI _Allreduce	MPI_Sendrecv (or Send, Isend)	MPI_Test (or Testany)	MPI_Wait (or Waitall)
Graph500	122 s	50 MB/s	9 KB	45,724,042	7,251	607,409	43,852,063	2
HACC	69 s	100 MB/s	6 MB	7,252	41	1,504	0	2,748
HPCG	68 s	3 MB/s	4 KB	128,940	555	40,353	0	40,353
LAMMPS	15 s	5 MB/s	70 B	3,012,409	125,170	893,845	0	812,673
MILC	87 s	90 MB/s	16 KB	10,842,875	13,224	528,076	0	1,056,152
miniAMR	23 s	40 MB/s	22 KB	789,958	10,326	18,366	0	35,540
miniMD	65 s	90 MB/s	3 KB	2,736,074	4,810	2,064,036	0	0
QMCPACK	67 s	600 KB/s	13 KB	8,160	2,004	390	2,697	195



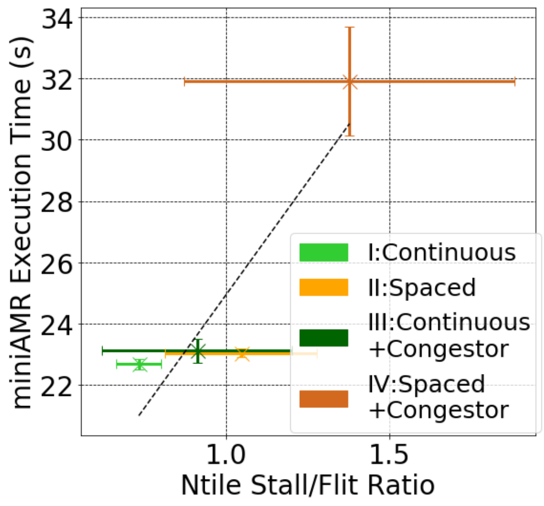
(a) HACC



(b) LAMMPS



(c) MILC



(d) miniAMR

Figure 3·18: There are positive correlations between ntile stall/flit ratio and application execution time. A cross represents the average of 10 runs for each setting. Errorbars are standard errors. The dashed line is a linear fit. These positive correlations suggest that ntile stall/flit ratio metric is indicative of performance degradation caused by network congestion.

congestion.

These findings suggest several key criteria for predicting congestion's impact on an application. The first is the amount of time an application spends performing MPI operations. Intuitively, an application not spending much time on communication will not be sensitive to congestion. Secondly, the type of communication matters. In our experiments, when collectives such as MPI_Allreduce, MPI_Bcast, and MPI_Barrier occupy more than 5% of time, we regard the application as having intensive MPI operation and expect it to be sensitive to congestion. Lastly, MPI_Wait(all) is important as they often indicate synchronization points where the slowest communication dominates performance (as is the case with MILC). Conversely, though Graph500 performs reasonable amounts of communication, the communications are uncoupled from each other as MPI_Test(any) calls indicate communication events that are completely independent of many other communications. Applying this understanding to Table II, we consider HACC, LAMMPS, MILC, miniAMR, miniMD, and QMCPACK as having intensive MPI operations.

From Table 3.5, we see the relationship between average message size and sensitivity to congestion is not clear. HACC, LAMMPS and MILC use very different message sizes but each seems sensitive to congestion. Other studies have found that small-size, latency-sensitive communications are more sensitive to congestion than bandwidth benchmarks typically with large message size (Chunduri et al., 2019). However, this relationship is not as clear cut for real applications.

Based on our results, aggregate data transfer rate is not indicative of congestion sensitivity either. For example, although Graph500 transfers data at 50 MB/s, it is less impacted by congestion than LAMMPS and QMCPACK which transfer data at merely 5 MB/s and 600 KB/s, respectively.

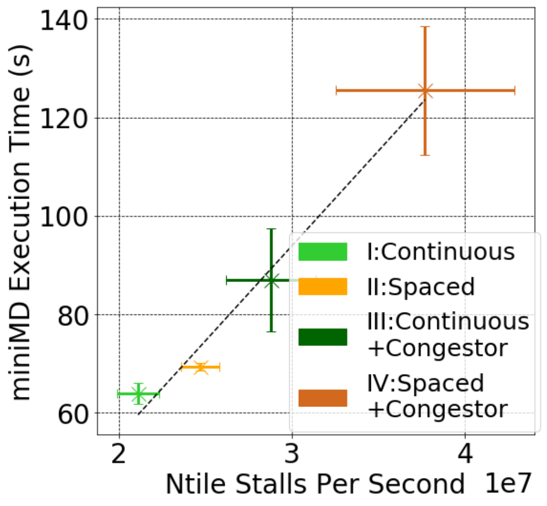
From Fig. 3·17, we also notice that the applications are more impacted by endpoint congestion than by intermediate congestion. Comparing Setting II with IV, we see HACC, LAMMPS, MILC, miniAMR, miniMD, and QMCPACK are all significantly impacted by endpoint congestion. Comparing Setting I with III, we see only MILC and miniMD are significantly impacted by intermediate congestion. This observation suggests that a compact job allocation strategy is better than a non-compact one because a non-compact allocation increases a job's probability to share routers with other jobs and are more likely to suffer from endpoint congestion.

3.6.3 Correlating Network Metrics with Application Performance

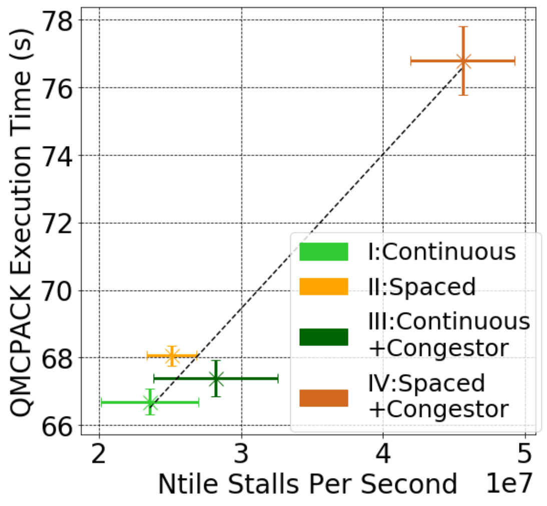
From the same experiments in Section 3.6.2, we correlate execution time with ntile stall/flit ratio in Fig. 3·18. Each cross represents the average value of the ten runs in each setting, and errorbars show their standard error. The ntile stall/flit ratio is calculated using the formula in Section 3.5.2, and averaged only over routers that contain nodes running our application. The metric is also averaged over the duration of the application.

In each case, we notice a positive correlation between job execution time and ntile stall/flit ratio, which demonstrates that this metric is indicative of application performance. Because the ntile counters collect not only local communications directly related to the host router but also communications that travel through the host router as an intermedium, our metric is only statistically correlated with job performance and suffers from variations caused by background traffic.

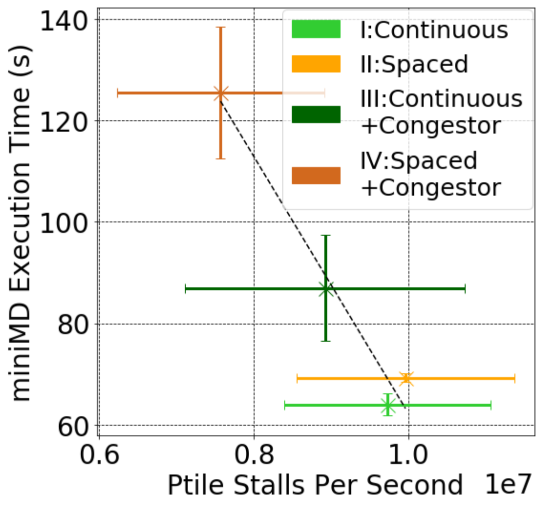
We also show the stall per second values on either ntiles or ptiles in Fig. 3·19. The stall count is averaged over routers and durations. While ntile stall per second shows a similar trend as ntile stall/flit ratio, the ptile stall per second shows a negative correlation with execution time. Although this negative correlation seems counter-intuitive at first thought, it in fact implies that ntile links, instead of the ptile-to-node links, are the communication bottleneck in these experiments.



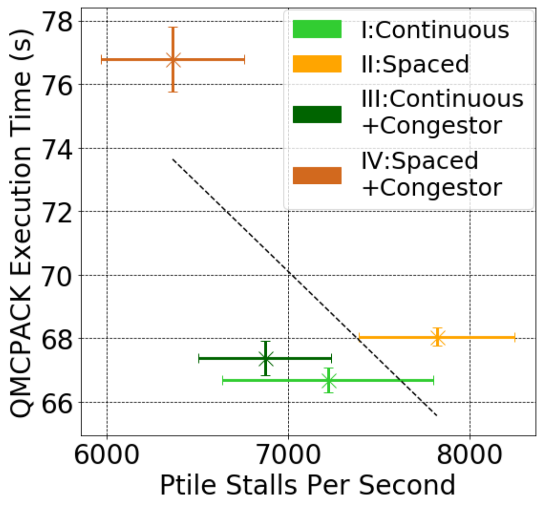
(a) miniMD - Ntile Stall



(b) QMCPACK - Ntile Stall



(c) miniMD - Ptile Stall



(d) QMCPACK - Ptile Stall

Figure 3·19: There are positive (negative) correlations between ntile (ptile) stalls per second and application execution time, respectively. The negative correlations in (c) and (d) imply that ptile-to-node links are not the bottleneck of the network.

When ntiles are the bottleneck, performance degradation causes an application to run slower and receive less messages per second. As a result, there are less flits per second in the ptile-to-node links. Less flits per second leads to less stalls per second on ptiles since these ptile-to-node links are not the bottleneck. Another way to explain the phenomenon is that the convergence of traffic occurs before the final hop within a switch. Once traffic makes it past the bottleneck, the rest of the path is relatively clear. This explains the negative correlation we see in Fig. 3·19(c,d). Therefore, we conclude that ntile metrics are better indicators for congestion than ptile metrics since ntiles links, rather than ptiles, are mostly the bottleneck.

3.6.4 Correlating Network Metrics with Network Congestors

We also conduct experiments that run GPCNeT on either 16, 32, 64, 86, 106, or 128 nodes without our applications. We use the impact factor metrics reported by GPCNeT to quantify the intensity of congestion created by GPCNeT. Figure 3·20 shows the correlation between impact factor (bandwidth ratio) and ntile stall/flit ratio. Each point represents an experiment run. The ntile stall/flit ratio is averaged similarly as before. We see a rough correlation between GPCNeT congestion intensity (quantified by impact factor) and ntile stall/flit ratio, which demonstrates that ntile stall/flit ratio is indicative of network congestion created by GPCNeT.

3.7 A Network-Data-Driven (NeDD) Job Allocation Policy

Section 3.6 reveals that certain network metrics are indicative of the appearance of network contention on HPC systems, which inspires us to design a job allocation policy that avoids network contention hot spots by extracting network congestion information from hardware counters at runtime.

Therefore, we propose the following Network-Data-Driven (NeDD) job allocation policy for HPC systems. This policy prioritizes allocating network-sensitive jobs to

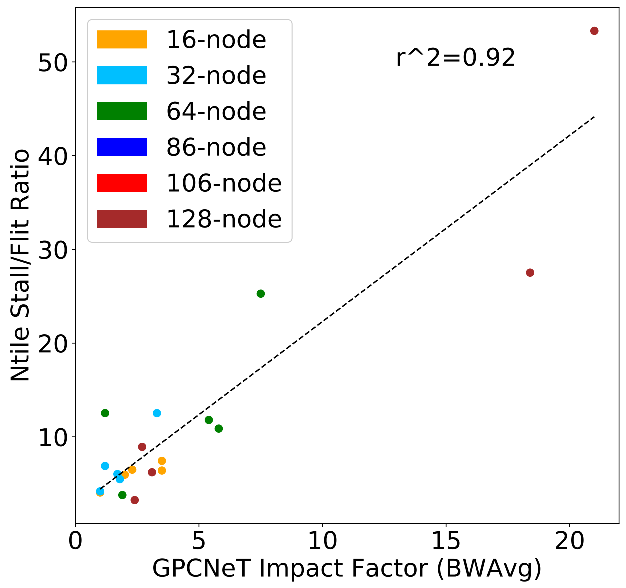


Figure 3·20: Correlating ntile stall/flit ratio with GPCNeT congestor impact factor. Different colors represent experiments where we run GPCNeT on different number of nodes.

routers with less network traffics. As shown in Fig. 3·21, at the time of allocating a job, our NeDD policy first reads the latest network traffic intensity on each node quantified by average bytes of communication per second. Next, the policy sorts all available routers by their network traffic summed over all nodes linked to the routers. For network-sensitive jobs, our policy will allocate the job to routers with less traffic, and for network-insensitive jobs, our policy will allocate the job to routers with more traffic.

To apply this policy, we assume the information regarding whether a job is network-sensitive or not is determined in advance and saved in a lookup table. Our experiments and analysis in Section 3.6.2 provide one approach to obtain this information by profiling the percentage of time a job spent on MPI collective operations (including MPI_Allreduce, MPI_Bcast, etc.).

Our NeDD policy quantifies the network traffic intensity of a router according to the traffic summed over all nodes linked to the routers, as depicted in Fig. 3·22. We

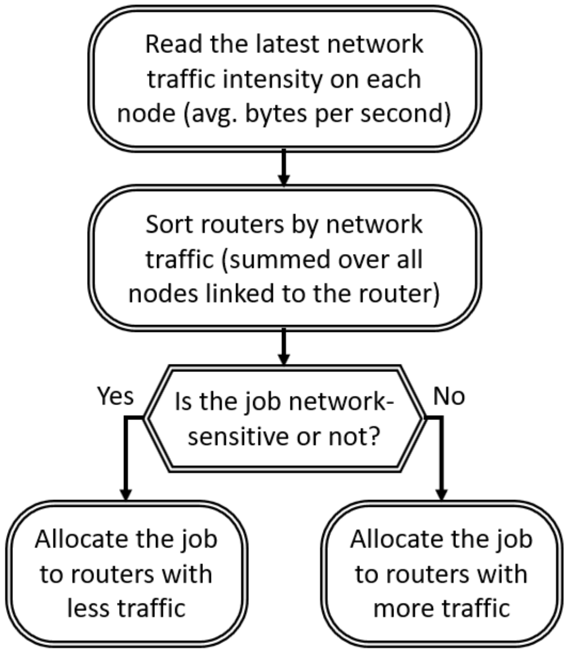

Read the latest network traffic intensity on each node (avg. bytes per second)
Sort routers by network traffic (summed over all nodes linked to the router)
Is the job network-sensitive or not?
Yes
No
Allocate the job to routers with less traffic
Allocate the job to routers with more traffic

Figure 3·21: The Network-Data-Drive (NeDD) job allocation policy for dragonfly systems.

design the policy in this way because our experiments in Section 3.6.2 demonstrate that network congestion created by neighbors (i.e., nodes connected to the same router) has much more significant impact on performance than non-neighbors. In Fig. 3·22, we show the case where four nodes are linked to each router, which follows the Aries network architecture implemented in the Cori system. Nonetheless, our NeDD policy can definitely be applied to systems where a router is connected with a different number of nodes.

3.8 Evaluting the NeDD Policy on Large HPC Systems

3.8.1 Experimental Design

We evaluate our NeDD policy on a large HPC systeml, Cori. The experiment design is shown in Fig. 3·23. To conduct the experiment, we first get N idle nodes (squares

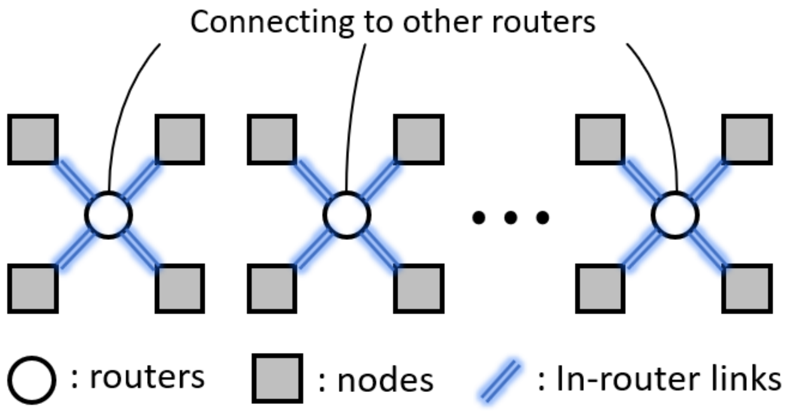


Figure 3·22: The NeDD policy quantifies the network traffic intensity of a router according to the traffic summed over all nodes linked to the routers.

in the figure) from the system[1]. Then, we run a network congestor using the GPC-NeT on C nodes (grey squares). After that, we run a parallel application on M nodes (green) using different job allocation policies and compare the execution time of the application under different allocation policies. A node the runs our experiment script and the LDMS storage daemon (purple) is prevented from running either the congestor or the application because the LDMS storage daemon is collecting the network counter data from all the other nodes and may creates network contention on this node.

We use the HPCG, LAMMPS, MILC, miniAMR, miniMD, QMCPACK applications detailed in Section 3.6 for this experiment.

We compare a number of different allocation strategies:

- **High-Traffic-Router** allocates a job to routers with high network traffics. This is what our NeDD policy will do to network-insensitive applications.

- **Random** policy allocates a job randomly among available nodes.

[1]The Slurm scheduler running on Cori is configured to prioritize selecting a contiguous set of nodes for a batch job. However, because we are experimenting on a production system, it is not easy to get a single contiguous set of nodes when N is large, and we actually get several sets of nodes for a batch job, where each set is a contigous set of nodes.

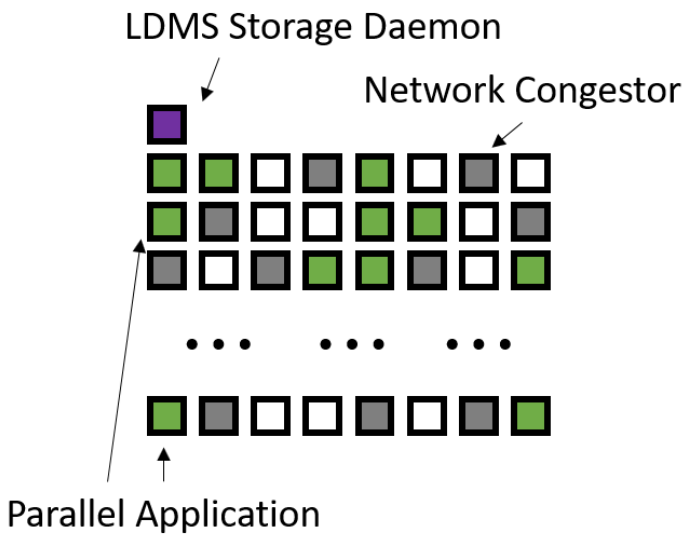


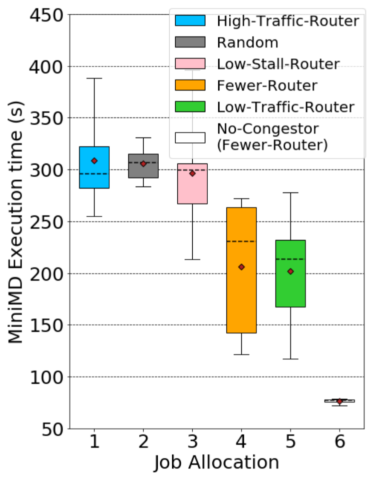
Figure 3·23: Our experimental design to evaluate the NeDD policy.

- **Low-Stall-Router** is a policy similar to NeDD. However, the Low-Stall-Router policy prioritizes allocating network-sensitive jobs to routers with lower network stall count (summed over all ports in this router) instead of routers with less network traffic.
- **Fewer-Router** allocates a policy into fewer number of routers. In other word, it prioritizes choosing routers that is connected to more idle nodes.
- **Low-Traffic-Router** allocates a job to router with low network traffics. This is what our NeDD policy will do to network-sensitive applications.

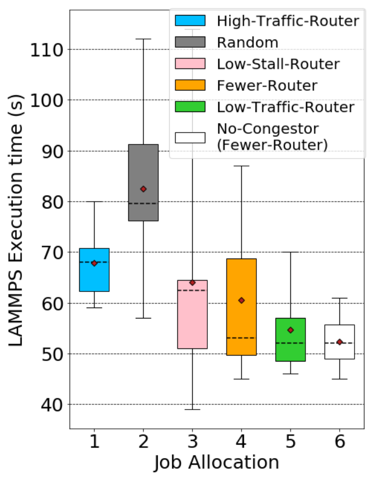
In our experiments, we also record the execution time of the applications when we do not run the network congestor (and allocated to the nodes following the Fewer-Router policy). This case is denoted as "No-Congestor".

3.8.2 Experimental Results

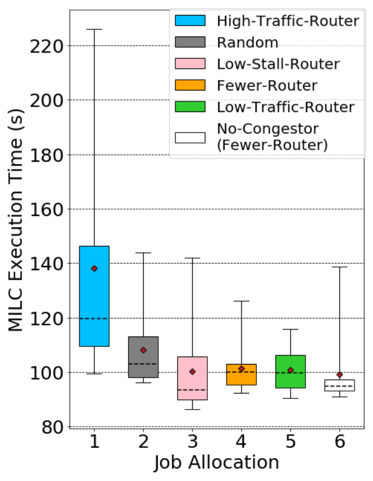
Our experiment results are shown in Fig. 3·24. These experiments use $N = 200$ nodes in total (not including the node running LDMS daemon), run network congestor on



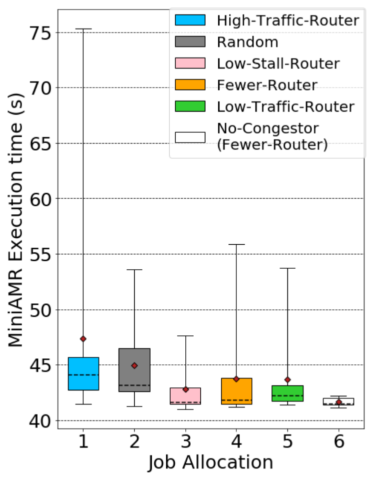
(a) miniMD



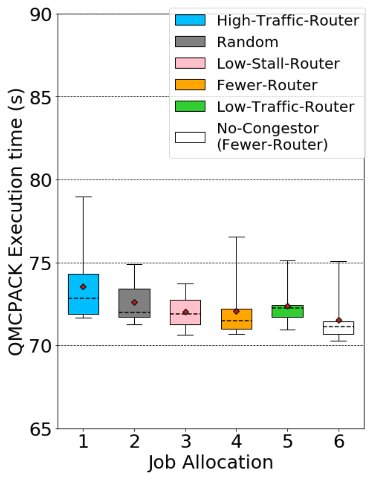
(b) LAMMPS



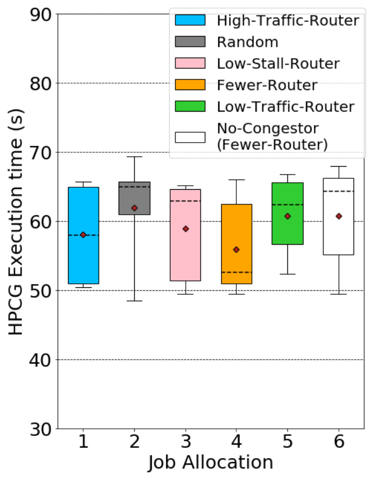
(c) MILC



(d) miniAMR



(e) QMCPACK



(f) HPCG

Figure 3·24: Results comparing different job allocation policies. Errorbars show the minimum and maximum execution time in the ten runs for each application. Colored area shows the first and third quartiles. The dashed black line shows the median, and the red point shows the average.

$C = 64$ nodes, and run a parallel application on $M = 32$ nodes.

As demonstrated in Fig. 3·24(f), for a network-insensitive application like HPCG, the impact of network congestion on the application performance is neglibile, and all allocation policies perform similarly. Especially, the variance of execution time with each allocation policy is similar to that in the No-Congestor case. This proves that our NeDD allocation policy can safely allocate the job to high-traffic-router without causing performance degradation while researving the low-traffic-routers to network-sensitive jobs.

On the other hand, as demonstrated in Fig. 3·24(a), for a network-sensitive application like miniMD, network congestion leads to an extention on job execution time by up to 5x (when comparing the average execution time for Random and No-Congestor cases), and the Low-Traffic-Router policy is performing 30% better than the Random policy, 29% better than the Lower-Router-Stall policy, and 7% better than the Fewer-Router policy. This proves that our NeDD policy which allocates network-sensitive jobs to low-traffic routers improves the performance of these jobs significantly.

In conclusion, we propose a Network-Data-Driven (NeDD) job allocation policy for HPC systems. Our NeDD policy is composed of three components: 1) a network measurement component that quantifies the network traffic intensity in the HPC system; 2) an application profiling component that determines whether an application is sensitive to network congestion; 3) a congestion-avoidance mechanism to allocate jobs that mitigate network congestion. Our current version of the policy applies a straightforward strategy to realize these three components. However, it is possible to improve our policy in future works by replacing one of these components by better algorithms. For example, other approaches to quantify network congestion in HPC systems can be considered (Jha et al., 2020).

Chapter 4

Conclusions and Future Work

The large energy cost and resource contention pose a serious challenge to the growth of HPC systems. To meet that challenge, in this work, we design intelligent middleware to improve the energy cost efficiency and performance of HPC systems. Our middleware enables HPC system to participate in demand response programs with QoS assurance of job performance, and the middleware also mitigates resource contention and improves performance through novel job allocation strategies.

4.1 HPC System Demand Response Participation

HPC systems are large power consumers, but the flexibility of job scheduling and the capability of server power capping enable them to regulate their total power consumption at a short time-scale. HPC systems' capability to significantly regulate their power renders them particularly good candidates for demand response programs, especially the regulation service where the participant needs to follow a target power that changes every few seconds. Prior works developed policies that enable HPC systems to participate in regulation service but without providing assurances on the QoS of computing jobs (Chen et al., 2014; Chen et al., 2019). To fill this gap, in this thesis, we demonstrate that, when equipped with well-designed power management policies and bidding policies, HPC systems benefit from participation in demand response programs while providing QoS guarantees on job performance. We propose QoSG and AQA policies that regulate HPC systems' power consumption to follow a power target when providing regulation service reserves. These policies provide QoS assurance by optimizing some scheduling parameters based on queueing theory. Through both simulations and real-system experiments with a broad set of workload traces, we demonstrate that our policies reduce the electricity cost of HPC systems by 10-50% while abiding by all QoS constraints, outperforming baseline policies proposed in prior works.

Future directions in this area include relaxing the assumptions in our work, enabling multi-site HPC system cooperation, and considering interaction with the electricity grid.

First, since our existing policies assume the knowledge of job arrival probability distributions, it would be helpful to understand and improve on situations where job arrival distributions deviate from the assumed models. If there are periodic trends or other predictable trends in job arrival patterns, a predictor could also be included to improve the performance of HPC demand response participation. In addition, since our policies assume the knowledge of jobs' expected execution time and power usage, it would be useful if this need for accurate job profiling can be removed or replaced by strategies that only need approximate knowledge.

Second, since the electricity price and the demand response dynamics among different geological locations are different, coordinating multiple HPC systems to participate in demand response together could be more beneficial than each HPC system participating in demand response individually. For example, when electricity price is low or consumption demand is high at one location, HPC systems at this location could accept jobs redistributed from other HPC systems. Designing some optimized job redistribution strategy in this way could further reduce the electricity cost of multi-site HPC systems. However, job redistribution does not come at no cost, thus, the monetary or other forms of cost of job redistribution should be considered when proposing these strategies.

Third, not only HPC systems need to optimize their cost, but independent system operators on the electricity grid side also need to optimize their cost, so exploring the interplay between the independent system operators and HPC systems could offer some insights in optimizing their cost together. Besides, some demand response programs offer a bidding mechanism where the price is set from bidding among multiple participants. Designing bidding strategies for HPC systems or improving the bidding mechanism from the electricity grid perspective are both interesting directions deserving exploration.

4.2 Mitigating HPC Network Contention through Job Allocation

Network contention is a major cause of performance degradation in modern HPC systems. Designing intelligent job allocation strategies could help in mitigating the impact of network contention on application performance.

On dragonfly networks, there are two allocation strategies to reduce communication latency: one is compactly allocating the tasks of a parallel application to harness locality, and the other is spreading the tasks across the machine to balance congestion on network links. Existing allocation policies resemble only one of these two strategies. To combine the benefits of these two strategies, we propose the *Level-Spread* allocation policy, which finds the lowest network level (router, group, or machine) a job can fit in and spreads the job throughout that level. Therefore, our allocation policy combines the advantages of existing policies for dragonfly networks. To evaluate *Level-Spread*, we conduct extensive simulations using SST with a broad range of workloads. We compare *Level-Spread* with eight baseline allocation policies, and conclude that *Level-Spread* outperforms the state-of-the-art by 16% on average (and up to 71%) in terms of communication time. To examine the applicability of our policy under different conditions, we conduct simulations with various dragonfly configurations, global link bandwidths, job communication intensities, and communication patterns. Our results validate the generality of *Level-Spread*, and we show that the performance gain from our policy increases when the machine size increases or the global link bandwidth decreases.

To mitigate network contention on HPC systems, we also conduct some experiments to quantify the impact of network congestion on job performance and network metrics. Our experiments show that applications demonstrate substantial difference under network congestion. Applications with intensive MPI operations (especially the applications that spend large portion of time on MPI collectives) suffer from 0.4X to 7X extension in execution times under network congestion, while applications with less intensive MPI operations are negligibly affected. By analyzing Aries network counters, we observe a positive correlation between application execution time and some network metrics counting stalls or flits, which demonstrates that these metrics are good indictors of network congestion and job performance.

Based on that finding, we design a Network-Data-Driven job allocation policy for HPC systems. The policy collects the latest network metrics and prioritize allocating the network-sensitive applications onto routers with less network traffics. Our evaluation on a large HPC system shows that the policy improve the performance of network-sensitive applications by up to 30% compared to baseline policies, while the performance of network-insensitive applications degrades for less than 3%.

Future directions in this area include generalizing our allocation strategies to other network topologies, co-optimizing job allocation with job scheduling strategies, and improving the Network-Data-Driven job allocation policy with more advanced algorithms.

First, although our Level-Spread and Network-Data-Driven job allocation policies are proposed and evaluated on dragonfly-topology HPC systems, they could be generalized to other network topologies following similar intuitions. For example, fat-tree network is similar to dragonfly network in its low-diameter property, so Level-Spread allocation could also help harness node locality and mitigate link congestion in fat-tree networks. Also, since network counters are deployed in many HPC systems other than dragonfly topology, our Network-Data-Driven policy could also be applied there.

Second, our work focuses on job allocation (where to allocate a job) and does not consider job scheduling (when to allocate a job). However, since job allocation and scheduling affect each other, co-optimizing the allocation and scheduling in HPC systems could offer more performance improvement than separately optimizing the two components. For example, when there are two jobs in the queue, it could be more beneficial the allocate the more-network-sensitive job first to routers with less network traffic instead of later. That is because if we allocate that job later than the other job, the other job may take the less-traffic routers while the job does not benefit much from those less-traffic routers.

Third, our Network-Data-Driven policy is proposed to simply avoid the high-traffic routers as it is an intuitive way to mitigate network contention. However, since existing technologies allow us to collect a large amount of network data from all nodes with high granularity, more advanced algorithms could be applied to extract information from the data and further improve job allocation and scheduling. For example, time-series analysis or machine learning could be applied to predict job performance and optimize job allocation and scheduling in HPC systems.

References

Agelastos, A. et al. (2014). The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 154–165.

Ahmed, K., Liu, J., and Wu, X. (2017). An energy efficient demand-response model for high performance computing systems. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 175–186.

Aksanli, B. and Rosing, T. (2014). Providing regulation services and managing data center peak power budgets. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4.

Alverson, B., Froese, E., Kaplan, L., and Roweth, D. (2012). Cray xc series network. `https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf`.

Amazon (2020). Amazon ec2 spot instances. `https://aws.amazon.com/ec2/spot`.

Antypas, K., Shalf, J., and Wasserman, H. (2008). NERSC - 6 workload analysis and benchmark selection process. `https://www.osti.gov/biblio/938789`.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67.

Bailey, D. H., Barszcz, E., Barton, J. T., et al. (1991). The nas parallel benchmarks. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165.

Bauer, G., Gottlieb, S., and Hoefler, T. (2012). Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 652–659.

Belka, M., Doubet, M., Meyers, S., Momoh, R., Rincon-Cruz, D., and Bunde, D. P. (2017). New link arrangements for dragonfly networks. *IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pages 17–24.

Bertsimas, D., Paschalidis, I. C., and Tsitsiklis, J. N. (1999). Large deviations analysis of the generalized processor sharing policy. *Queueing Systems*, 32(4):319–349.

Bhatele, A., Jain, N., Livnat, Y., Pascucci, V., and Bremer, P. (2016). Analyzing network health and congestion in dragonfly-based supercomputers. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 93–102.

Bhatele, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013a). There goes the neighborhood: Performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 41:1–41:12, New York, NY, USA. ACM.

Bhatele, A., Mohror, K., Langer, S. H., and Isaacs, K. E. (2013b). There goes the neighborhood: performance degradation due to nearby jobs. *Proceedings of the International Conf. on High Performance Computing, Networking, Storage and Analysis, (SC)*, pages 41:1—-41:12.

Bhatele, A., Thiagarajan, J. J., Groves, T., Anirudh, R., Smith, S. A., Cook, B., and Lowenthal, D. K. (2020). The case of performance variability on dragonfly-based systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Bhatele, A., Titus, A. R., Thiagarajan, J. J., Jain, N., Gamblin, T., Bremer, P., Schulz, M., and Kale, L. V. (2015). Identifying the culprits behind network congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 113–122.

Bohringer, C., Loschel, A., Moslener, U., and Rutherford, T. F. (2009). Eu climate policy up to 2020: An economic impact assessment. *Energy Economics*, 31, Supplement 2:S295 – S305.

Borghesi, A., Bartolini, A., Milano, M., and Benini, L. (2019). Pricing schemes for energy-efficient hpc systems: Design and exploration. *The International Journal of High Performance Computing Applications*, 33(4):716–734.

Brandt, J. M., Froese, E., Gentile, A. C., Kaplan, L., Allan, B. A., and Walsh, E. J. (2016). Network performance counter monitoring and analysis on the cray xc platform. `https://www.osti.gov/biblio/1422085`.

Budiardja, R., Crosby, L., and You, H. (2013). Effect of rank placement on Cray XC30 communication cost. `https://cug.org/proceedings/cug2013_proceedings/includes/files/pap153.pdf`.

Chakaravarthy, V. T., Kedia, M., Sabharwal, Y., Kumar Katta, N. P., Rajamony, R., and Ramanan, A. (2012). Mapping strategies for the PERCS architecture. *International Conf. on High Performance Computing (HiPC)*.

Chapman, B. et al. (2018). Rabbitmq. `https://github.com/rabbitmq`.

Chen, H., Caramanis, M. C., and Coskun, A. K. (2014). The data center as a grid load stabilizer. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, (i):105–112.

Chen, H., Hankendi, C., Caramanis, M. C., and Coskun, A. K. (2013). Dynamic server power capping for enabling data center participation in power markets. In *Intl. Conf. on Computer-Aided Design (ICCAD)*.

Chen, H., Zhang, Y., Caramanis, M. C., and Coskun, A. K. (2019). Energyqare: Qos-aware data center participation in smart grid regulation service reserve provision. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 4(1):2:1–2:31.

Chen, Z., Wu, L., and Li, Z. (2014). Electric demand response management for distributed large-scale internet data centers. *IEEE Transactions on Smart Grid*, 5(2):651–661.

Christian, B. (2011). *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University.

Chunduri, S., Groves, T., Mendygral, P., Austin, B., Balma, J., Kandalla, K., Kumaran, K., Lockwood, G., Parker, S., Warren, S., Wichmann, N., and Wright, N. (2019). GPCNeT: Designing a benchmark suite for inducing and measuring contention in hpc networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery.

Chunduri, S., Harms, K., Parker, S., Morozov, V., Oshin, S., Cherukuri, N., and Kumaran, K. (2017). Run-to-run variability on xeon phi based cray xc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 52:1–52:13, New York, NY, USA. ACM.

Cioara, T., Anghel, I., Bertoncini, M., Salomie, I., Arnone, D., Mammina, M., Velivassaki, T., and Antal, M. (2018). Optimized flexibility management enacting data centres participation in smart demand response programs. *Future Generation Computer Systems*, 78:330 – 342.

Clausen, A., Koenig, G., Klingert, S., Ghatikar, G., Schwartz, P. M., and Bates, N. (2019). An analysis of contracts and relationships between supercomputing centers and electricity service providers. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, ICPP 2019, pages 4:1–4:8, New York, NY, USA. ACM.

Cray Inc. (2018). Aries hardware counters (4.0). `http://pubs.cray.com/content/S-0045/4.0/aries-hardware-counters`.

Cray Inc. (2020). Cray performance measurement and analysis tools user guide (7.0.0). `https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/craypat`.

Cupelli, L., Schütz, T., Jahangiri, P., Fuchs, M., Monti, A., and Müller, D. (2018). Data center control strategy for participation in demand response programs. *IEEE Transactions on Industrial Informatics*, 14(11):5087–5099.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). Rapl: Memory power estimation and capping. *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194.

Dhiman, G., Marchetti, G., and Rosing, T. (2009). vGreen: a system for energy efficient computing in virtualized environments. In *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 243–248. ACM.

Dongarra, J., Heroux, M. A., and Luszczek, P. (2016). A new metric for ranking high-performance computing systems. *National Science Review*, 3(1):30–35.

EIA (2014). Annual energy outlook 2014. `http://www.eia.gov/forecasts/aeo`.

Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J. (2012). Cray cascade: A scalable hpc system based on a dragonfly network. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9.

Faizian, P., Rahman, S., Mollah, A., Yuan, X., Pakin, S., and Lang, M. (2016). Traffic pattern-based adaptive routing for intra-group communication in dragonfly networks. *IEEE Annual Symposium on High-Performance Interconnects (HOTI)*.

Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the Parallel Workloads Archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982.

Fuentes, P., Vallejo, E., Camarero, C., Beivide, R., and Valero, M. (2016). Network unfairness in dragonfly topologies. *Journal of Supercomputing*, 72(12):4468–4496.

Gandhi, A., Harchol-Balter, M., and Kozuch, M. A. (2012). Are sleep states effective in data centers? In *2012 International Green Computing Conference (IGCC)*, pages 1–10.

Garcia, M., Vallejo, E., Beivide, R., Odriozola, M., and Valero, M. (2013). Efficient routing mechanisms for dragonfly networks. *Proceedings of the International Conf. on Parallel Processing*, pages 582–592.

Govindan, S., Sivasubramaniam, A., and Urgaonkar, B. (2011). Benefits and limitations of tapping into stored energy for datacenters. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 341–352, New York, NY, USA. ACM.

Groves, T., Grant, R. E., Hemmer, S., Hammond, S., Levenhagen, M., and Arnold, D. C. (2016). (sai) stalled, active and idle: Characterizing power and performance of large-scale dragonfly networks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 50–59.

Groves, T., Gu, Y., and Wright, N. J. (2017). Understanding performance variability on the aries dragonfly network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 809–813.

Hansen, J., Knudsen, J., and Annaswamy, A. M. (2014). Demand response in smart grids: Participants, challenges, and a taxonomy. In *53rd IEEE Conference on Decision and Control*, pages 4045–4052.

Hastings, E., Rincon-Cruz, D., Spehlmann, M., Meyers, S., Xu, A., Bunde, D. P., and Leung, V. J. (2015). Comparing global link arrangements for dragonfly networks. In *2015 IEEE International Conference on Cluster Computing*, pages 361–370.

Heitmann, K., Finkel, H., Pope, A., Morozov, V., et al. (2019). The outer rim simulation: A path to many-core supercomputers. *The Astrophysical Journal Supplement Series*, 245(1):16.

Heroux, M. A. et al. (2009). Improving performance via mini-applications. `https://www.osti.gov/biblio/993908`.

Hertel, E. S., Bell, R. L., Elrick, M. G., Farnsworth, A. V., Kerley, G. I., McGlaun, J. M., Petney, S. V., Silling, S. A., Taylor, P. A., and Yarrington, L. (1995). *CTH: A software family for multi-dimensional shock physics analysis*, pages 377–382. Springer Berlin Heidelberg.

Hsieh, M.-Y., Rodrigues, A., Riesen, R., Thompson, K., and Song, W. (2011). A framework for architecture-level power, area, and thermal simulation and its application to network-on-chip design exploration. *ACM SIGMETRICS Performance Evaluation Review*, 38:63.

Jain, N., Bhatele, A., Ni, X., Wright, N. J., and Kale, L. V. (2014). Maximizing throughput on a dragonfly network. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 336–347.

Jain, N., Bhatele, A., Robson, M. P., Gamblin, T., and Kale, L. V. (2013). Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 95:1–95:12, New York, NY, USA. ACM.

Jain, N., Bhatele, A., White, S., Gamblin, T., and Kale, L. V. (2016). Evaluating hpc networks via simulation of parallel workloads. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165.

Jha, S., Brandt, J., Gentile, A., Kalbarczyk, Z., and Iyer, R. (2018). Characterizing supercomputer traffic networks through link-level analysis. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 562–570.

Jha, S., Patke, A., Brandt, J., Gentile, A., Lim, B., Showerman, M., Bauer, G., Kaplan, L., Kalbarczyk, Z., Kramer, W., and Iyer, R. (2020). Measuring congestion in high-performance datacenter interconnects. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 37–57, Santa Clara, CA. USENIX Association.

Jiang, N., Kim, J., and Dally, W. J. (2009). Indirect adaptive routing on large scale interconnection networks. *ACM SIGARCH Computer Architecture News*, 37:220.

Jokanovic, A., Sancho, J. C., Rodriguez, G., Lucero, A., Minkenberg, C., and Labarta, J. (2015). Quiet neighborhoods: key to protect job performance predictability. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 449–459.

Kambadur, M., Moseley, T., Hank, R., and Kim, M. A. (2012). Measuring interference between live datacenter applications. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Kaplan, F., Tuncer, O., Leung, V. J., Hemmert, S. K., and Coskun, A. K. (2017). Unveiling the Interplay between Global Link Arrangements and Network Management Algorithms on Dragonfly Networks. *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 325–334.

Kim, J., Baczewski, A. D., Beaudet, T. D., Benali, A., et al. (2018). QMC-PACK: an open sourceab initioquantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901.

Kim, J., Dally, W. J., Scott, S., and Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. *International Symposium on Computer Architecture (ISCA)*, pages 77–88.

Kim, J. et al. (2014). QMCPACK sample input. `https://github.com/QMCPACK/qmcpack/blob/develop/examples/molecules/H2O/simple-H2O.xml`.

Klawonn, A., Lanser, M., Rheinbach, O., Wellein, G., and Wittmann, M. (2020). Energy efficiency of nonlinear domain decomposition methods. *The International Journal of High Performance Computing Applications.*

Laurie, D. et al. (2018). An open-source tool for controlling ipmi-enabled systems. `https://github.com/ipmitool/ipmitool`.

Le, T. N., Liu, Z., Chen, Y., and Bash, C. (2016a). Joint capacity planning and operational management for sustainable data centers and demand response. In *Proceedings of the 7th International Conference on Future Energy Systems*, e-Energy '16, pages 16:1–16:12.

Le, T. N., Liu, Z., Chen, Y., and Bash, C. (2016b). Joint capacity planning and operational management for sustainable data centers and demand response. In *Proceedings of the Seventh International Conference on Future Energy Systems*, e-Energy '16, pages 16:1–16:12, New York, NY, USA. ACM.

Leung, V. J., Arkin, E. M., Bender, M. A., Bunde, D., Johnston, J., Lal, A., Mitchell, J. S. B., Phillips, C., and Seiden, S. S. (2002). Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. *IEEE International Conf. on Cluster Computing, (CLUSTER)*, pages 296–304.

Li, S., Brocanelli, M., Zhang, W., and Wang, X. (2014). Integrated power management of data centers and electric vehicles for energy and regulation market participation. *IEEE Transactions on Smart Grid*, 5(5):2283–2294.

Liu, Z., Liu, I., Low, S., and Wierman, A. (2014). Pricing data center demand response. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, page 111–123. Association for Computing Machinery.

Maiterth, M., Koenig, G., Pedretti, K., Jana, S., Bates, N., Borghesi, A., Montoya, D., Bartolini, A., and Puzovic, M. (2018). Energy and power aware job scheduling and resource management: Global survey — initial analysis. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 685–693.

Melo, D. and Carvalho, A. (2010). The new linux perf tools. In *Slides from Linux Kongress*, volume 18.

Mubarak, M., Carothers, C. D., Ross, R., and Carns, P. (2012). Modeling a million-node dragonfly network using massively parallel discrete-event simulation. *SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, pages 366–376.

Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. (2010). Introducing the graph 500. In *Cray Users Group (CUG)*, volume 19, pages 45–74.

National Energy Research Scientific Computing Center (NERSC) (2020). NERSC Queue Policy. `https://docs.nersc.gov/jobs/policy/`.

New York Independent System Operator (NYISO) (2020). Ancillary services manual, v6.0. `https://www.nyiso.com/manuals-tech-bulletins-user-guides`.

Niu, L. and Guo, Y. (2016). Enabling reliable data center demand response via aggregation. In *Proceedings of the Seventh International Conference on Future Energy Systems*, e-Energy '16, pages 22:1–22:11, New York, NY, USA. ACM.

Novoa, C. and Jin, T. (2011). Reliability centered planning for distributed generation considering wind power volatility. *Electric Power Systems Research*, 81(8):1654 – 1661.

Pahlevan, A., Zapater, M., Coskun, A., and Atienza, D. (2020). Ecogreen: Electricity cost optimization for green datacenters in emerging power markets. *IEEE Transactions on Sustainable Computing*, pages 1–1.

Parekh, A. K. and Gallager, R. G. (1993). A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357.

Paschalidis, I. C. (1999). Class-specific quality of service guarantees in multimedia communication networks. *Automatica*, 35(12):1951 – 1968.

Paschalidis, I. C., Li, B., and Caramanis, M. C. (2012). Demand-side management for regulation service provisioning through internal pricing. *IEEE Transactions on Power Systems*, 27(3):1531–1539.

Patel, T., Wagenhäuser, A., Eibel, C., Hönig, T., Zeiser, T., and Tiwari, D. (2020). What does power consumption behavior of hpc jobs reveal? : Demystifying, quantifying, and predicting power consumption characteristics. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 799–809.

Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19.

Prisacari, B., Garcia, M., Vallejo, E., and Beivide, R. (2014a). Performance implications of remote-only load balancing under adversarial traffic in Dragonflies. *International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip*.

Prisacari, B., Rodriguez, G., Heidelberger, P., Chen, D., Minkenberg, C., and Hoefler, T. (2014b). Efficient task placement and routing in dragonfly networks. *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 129–140.

Prisacari, B., Rodriguez, G., Minkenberg, C., Garcia, M., Vallejo, E., and Beivide, R. (2016). Performance optimization of load imbalanced workloads in large scale dragonfly systems. *IEEE International Conf. on High Performance Switching and Routing (HPSR)*.

Reda, S., Cochran, R., and Coskun, A. K. (2012). Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 32(5):64–75.

Rodrigues, A. F., CooperBalls, E., Jacob, B., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., and Rosenfeld, P. (2011). The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37.

Sandia Corporation (2014). LAMMPS sample input. `https://github.com/lammps/lammps/blob/master/examples/DIFFUSE/in.vacf.2d`.

Sandia National Laboratories (2020). Structural Simulation Toolkit (SST). `https://github.com/sstsimulator/sst-elements`.

Shehabi, A., Smith, S., Horner, N., Azevedo, I., Brown, R., Koomey, J., Masanet, E., Sartor, D., Herrlin, M., and Lintner, W. (2016). United states data center energy usage report. `https://www.osti.gov/biblio/1372902`.

Shi, Y., Xu, B., Zhang, B., and Wang, D. (2016). Leveraging energy storage to optimize data center electricity cost in emerging power markets. In *Proceedings of the Seventh International Conference on Future Energy Systems*, e-Energy '16, pages 18:1–18:13, New York, NY, USA. ACM.

Slurm (2016). Slurm's job allocation policy for dragonfly network. `https://github.com/SchedMD/slurm/blob/master/src/plugins/select/linear/select_linear.c`. [Line 1634-1932; accessed 30-March-2017].

Smith, S., Lowenthal, D., Bhatele, A., Thiagarajan, J., Bremer, P., and Livnat, Y. (2016). Analyzing inter-job contention in dragonfly networks. `https://www2.cs.arizona.edu/~smiths949/dragonfly.pdf`.

Sreepathi, S., D'Azevedo, E., Philip, B., and Worley, P. (2016). Communication Characterization and Optimization of Applications Using Topology-Aware Task Mapping on Large Supercomputers. *ACM/SPEC International Conf. on Performance Engineering (ICPE)*, pages 225–236.

Sun, Q., Ren, S., Wu, C., and Li, Z. (2016). An online incentive mechanism for emergency demand response in geo-distributed colocation data centers. In *Proceedings of the Seventh International Conference on Future Energy Systems*, e-Energy '16, pages 3:1–3:13, New York, NY, USA. ACM.

TOP500 (2020). The top 500 list. `https://www.top500.org/top500/lists/2020/11/`.

Tran, N. H., Pham, C., Ren, S., Han, Z., and Hong, C. S. (2016). Coordinated power reduction in multi-tenant colocation datacenter: An emergency demand response study. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6.

Tuncer, O., Zhang, Y., Leung, V. J., and Coskun, A. K. (2017). Task Mapping on a Dragonfly Supercomputer. *IEEE High Performance Extreme Computing Conf. (HPEC)*.

Underwood, K. D., Levenhagen, M., and Rodrigues, A. (2007). Simulating red storm: Challenges and successes in building a system simulation. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10.

Wang, W., Abdolrashidi, A., Yu, N., and Wong, D. (2019a). Frequency regulation service provision in data center with computational flexibility. *Applied Energy*, 251:113304.

Wang, Y., Zhang, F., Chi, C., Ren, S., Liu, F., Wang, R., and Liu, Z. (2019b). A market-oriented incentive mechanism for emergency demand response in colocation data centers. *Sustainable Computing: Informatics and Systems*, 22:13 – 25.

Wilke, J., Bennett, J., Kolla, H., Teranishi, K., Slattengren, N., and Floren, J. (2014). Extreme-Scale viability of collective communication for resilient task scheduling and work stealing. *IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN)*, pages 756–761.

Yang, X., Jenkins, J., Mubarak, M., Ross, R. B., and Lan, Z. (2016). Watch out for the bully! job interference study on dragonfly network. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 750–760.

Yu, Z., Guo, Y., and Pan, M. (2017). Coalitional datacenter energy cost optimization in electricity markets. In *Proceedings of the Eighth International Conference on Future Energy Systems*, e-Energy '17, pages 191–202, New York, NY, USA. ACM.

Zhang, L., Ren, S., Wu, C., and Li, Z. (2015). A truthful incentive mechanism for emergency demand response in colocation data centers. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2632–2640.

Zhou, Z., Liu, F., Chen, S., and Li, Z. (2020). A truthful and efficient incentive mechanism for demand response in green datacenters. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):1–15.

CURRICULUM VITAE

[redacted]

[redacted]

[redacted]

[redacted]