Fall 2024

RESOURCE MANAGEMENT AND OPTIMIZATION OF INTERACTIVE MICROSERVICE AND MPI-BASED ENSEMBLE APPLICATIONS IN THE CLOUD

Md Rajib Hossen
University of Texas at Arlington

RESOURCE MANAGEMENT AND OPTIMIZATION OF INTERACTIVE MICROSERVICE AND MPI-BASED ENSEMBLE APPLICATIONS IN THE CLOUD

by

MD RAJIB HOSSEN

DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy at

The University of Texas at Arlington

December 2024

Arlington, Texas

Supervising Committee:

Dr. Mohammad A. Islam, Supervising Professor

Dr. Leonidas Fegaras

Dr. Jia Rao

Dr. William Beksi

To my beloved wife, Most Akhi Chowdhury, whose steadfast support and encouragement have been an enduring source of strength and inspiration throughout this journey.

ACKNOWLEDGEMENTS

I would like to extend my heartfelt gratitude to my supervising professor, Dr. Mohammad A. Islam, for his constant motivation, encouragement, and invaluable advice throughout the course of my doctoral studies. I am deeply thankful to my dissertation committee members, Dr. Leonidas Fegaras, Dr. Jia Rao, and Dr. William Beksi, for their genuine interest in my research and for generously serving on my dissertation committee. My sincere thanks go to my mentors from Lawrence Livermore National Laboratory, Dr. Daniel Milroy, Dr. Vanessa Sochat, and Dr. Abhik Sarkar, for their unwavering support and mentorship. Their valuable insights, guidance, and direction have been instrumental in shaping my work.

I am forever grateful to my high school teachers, whose inspiration, support, and encouragement instilled in me the courage to dream big and pursue those dreams. Without their guidance, I would not be where I am today. I also want to express my deep appreciation to my brother figure, Mr. Mainuddin Chowdhury, for his unwavering belief in me and for inspiring me to think beyond the limits of what is possible.

Finally, I wish to express my profound gratitude to my cousin Ahmad Foysal, who has always been there for me during my academic and professional journey. I am truly fortunate to have such a blessing in my life. I am also deeply thankful to my mother and wife for their sacrifice, encouragement, and patience. Additionally, I extend my appreciation to several of my friends who have supported me throughout my career.

August 19, 2024

ABSTRACT

RESOURCE MANAGEMENT AND OPTIMIZATION OF INTERACTIVE MICROSERVICE AND MPI-BASED ENSEMBLE APPLICATIONS IN THE CLOUD

MD RAJIB HOSSEN, Ph.D.

The University of Texas at Arlington, 2024

Supervising Professor: Mohammad A. Islam

As user-interactive applications in the cloud transition from monolithic services to agile microservice architectures, efficient resource management becomes a key challenge. The multitude of loosely coupled components and fluctuating traffic patterns make traditional cloud autoscaling methods ineffective. Existing machine learning-based approaches, while attempting to address this, often require extensive training data and can lead to intentional violations of service level objectives (SLOs). To tackle these challenges, I propose PEMA (Practical Efficient Microservice Autoscaling), a lightweight resource manager for microservices. PEMA aims to optimize resource allocation through opportunistic resource reduction, considering the intricate dependencies between microservices.

On another front, scientific workflows are evolving to accommodate the increasing diversity and parallelism of modern computing systems. The integration of multi-scale simulations with Artificial Intelligence and Machine Learning (AI/ML) methods has made interdisciplinary workflows increasingly complex and challenging to man-

age using traditional high-performance computing (HPC) infrastructure. Converged computing, a growing movement that integrates HPC and cloud technologies into a seamless environment, can provide a means to bridge the gap between the needs and capabilities of modern scientific workflows. Ensemble-based HPC workflows, particularly those leveraging MPI-based (Message Passing Interface) workflows, stand to benefit from the efficiency improvements enabled by cloud-native orchestration. While these workflows have been demonstrated to scale in Kubernetes, limited work has explored the combined impact of autoscaling and elasticity on MPI-based workflows. To address this, we leveraged the Flux Operator, a Kubernetes operator of the Flux framework, and developed a workload-driven autoscaling strategy that outperforms traditional CPU utilization-based autoscaling for MPI-based ensembles. This approach enhances efficiency and reduces ensemble completion time by up to 4.7× compared to CPU utilization-based methods.

Additionally, significant power consumption remains a critical challenge for current and future HPC systems. Despite this, HPC systems often remain power underutilized, making them ideal candidates for power oversubscription to reclaim unused capacity. To mitigate the risk of system overload during oversubscription, I propose MPR (Market-based Power Reduction), a scalable, market-driven approach that incentivizes HPC users to reduce power consumption during overloads in exchange for rewards. Real-world trace-based simulations show that MPR consistently benefits both users and HPC managers by balancing resource gain and performance loss. We also demonstrate the effectiveness of MPR on a prototype system, highlighting its potential as a sustainable power management solution.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

Background: Cloud computing has revolutionized modern IT infrastructure by offering organizations the flexibility, scalability, and cost-effectiveness needed to handle diverse computational workloads. This shift has reduced capital expenditures and enabled businesses to access computational resources on-demand, transforming both public and private sectors. With the acceleration of cloud adoption, projections indicate that the cloud computing sector will generate nearly $1 trillion in revenue by 2025 [7]. This growth is not only reshaping traditional IT infrastructure but also driving new paradigms, such as the convergence of high-performance computing (HPC) with the portability and automation of the cloud.

One key architectural shift enabled by cloud computing is the rise of microservices, which break down monolithic applications into smaller, independently deployable services [8]. This architecture enhances the scalability and flexibility of applications, making it a preferred approach in cloud-native environments. However, the widespread adoption of microservices introduces new challenges in resource management, especially in balancing performance, cost-efficiency, and resource contention in cloud infrastructures.

Similarly, HPC systems are experiencing increased demands for more efficient resource management as they approach exascale computing. Large-scale scientific computations place tremendous strain on power and computational resources, requiring sophisticated strategies to manage resources while minimizing costs, particularly as the power consumption of supercomputers reaches megawatts.

Motivation: As cloud computing becomes ubiquitous, managing the resources of interactive applications built on microservices architectures has emerged as a critical challenge. Effective resource management is necessary to ensure performance, avoid resource contention, and optimize cost in dynamic cloud environments. Traditional rule-based resource management systems often fall short in efficiently managing cloud-native microservices, leading to unnecessary resource wastage or performance degradation under varying workloads.

In HPC systems, the power consumption and complexity of scheduling diverse workloads across heterogeneous resources pose significant challenges. Maximizing system utilization while minimizing energy consumption has become paramount, as the operational costs of supercomputers continue to escalate. There is also a growing need to integrate the elasticity and automation of cloud computing into HPC systems, allowing for real-time resource scaling based on workload demands.

To address these concerns, new resource management strategies that can dynamically adjust to workload variations and provide efficiency in both cloud and HPC environments are needed. This includes the development of lightweight, adaptive approaches that can operate without extensive training or over-reliance on static configurations.

Contribution Summary: This thesis contributes to addressing the challenges of resource management in both cloud and HPC environments through the following:

Power-Efficient Microservice Allocation (PEMA): We present a novel microservice resource manager, PEMA, which optimizes resource allocation for cloud-native applications. Unlike traditional methods, PEMA employs iterative, feedback-based tuning that starts with abundant resource allocation to meet Service Level Objectives (SLOs) and then gradually reduces resources based on real-time workload per-

formance. The method ensures resource efficiency without violating SLOs, leading to up to 33% savings compared to rule-based systems [9].

Market Mechanism-Based Power Management in HPC: We introduce a power management strategy for HPC systems that employs a market-based approach to power oversubscription. This method allows HPC systems to operate beyond traditional power limits while optimizing resource utilization. By involving users in real-time power management decisions, the system can dynamically allocate resources based on current needs, resulting in more efficient power usage during periods of high demand (3).

Elasticity in HPC Workloads with Cloud Integration: We explore the integration of cloud computing elasticity into HPC environments through the development of a workload-driven autoscaling algorithm. By leveraging Kubernetes (K8s) and the Flux Framework, we enable real-time resource scaling for HPC applications based on workload fluctuations, reducing operational costs and enhancing system performance. This convergence of cloud and HPC introduces a hybrid computing environment that supports the growing complexity of scientific workflows (4).

Author Contributions

Chapter 2: Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. 2022. Practical Efficient Microservice Autoscaling with QoS Assurance. In Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22). Association for Computing Machinery, New York, NY, USA, 240–252. https://doi.org/10.1145/3502181.3531460

Dr. Mohammad Islam provided supervision for the research direction, reviewed the research questions and experimental design, and made significant contributions

to improving the paper. My responsibilities included developing the entire system, creating the algorithm, conducting experiments, and writing the findings.

Chapter 3: Md Rajib Hossen, Kishwar Ahmed and Mohammad A. Islam, "Market Mechanism-Based User-in-the-Loop Scalable Power Oversubscription for HPC Systems," 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 2023, pp. 485-498, doi: 10.1109/HPCA56546.2023.10071

Dr. Mohammad Islam and Dr. Kishwar Ahmed supervised the research direction, reviewed the research questions and experimentation setup, and significantly improved the paper. My responsibilites entailed the development of high performance systems, designing and running experiments, and evaluation of results.

Chapter 4: Md Rajib Hossen, Vanessa Sochat, Abhik Sarkar, Mohammad A. Islam, and Daniel Milroy, "Enabling Workload-Driven Elasticity in MPI-based Ensembles", 2024 IEEE International Conference on Cluster Computing (CLUSTER), Japan.

- Dr. Vanessa Sochat, Dr. Abhik Sarkar, and Dr. Daniel Milroy supervised the work, offering valuable insights, guiding the experimental direction, and making significant improvements to the project. My responsibilities included developing the algorithm, setting up the cloud and HPC environment for experiments, conducting the experiments, and writing the paper.

CHAPTER 2

Practical Efficient Microservice Autoscaling with QoS Assurance

2.1 Introduction

Motivation. Microservices architecture is enjoying a growing penetration in user-facing cloud applications where an ensemble of loosely-coupled and small service components (i.e., microservices) work together to serve user requests [4, 10, 11]. As illustrated in Fig. 2.1, microservices architecture is a significant departure from traditional monolithic deployments with a few large application layers such as user-facing front-end, back-end business logic, and database [12]. Unlike monolithic applications, the small microservices can be easily managed and kept updated by small dedicated DevOps teams [13]. Moreover, microservices are typically stateless and communicate using lightweight APIs [14,15]. Hence, they offer agile resource management and scaling, better fault tolerance, and great platform agnostic compatibility among different microservices that cannot be matched by monolithic applications [13, 16].

Microservices come with their own sets of challenges, and in this paper, we focus on its resource management. In principle, microservice resource management is same as monolithic applications - achieve the desired performance (e.g., end-to-end response latency) with the minimum resource allocation [17–19]. Resource management for microservices-based applications, however, is more challenging because these applications have a much larger configuration space due to the sheer number of microservices responsible for the application performance. For example, if we consider an application with m microservices where each microservice can be configured with n different CPU allocations, there will be n^m possible resource configurations. Moreover, mi-

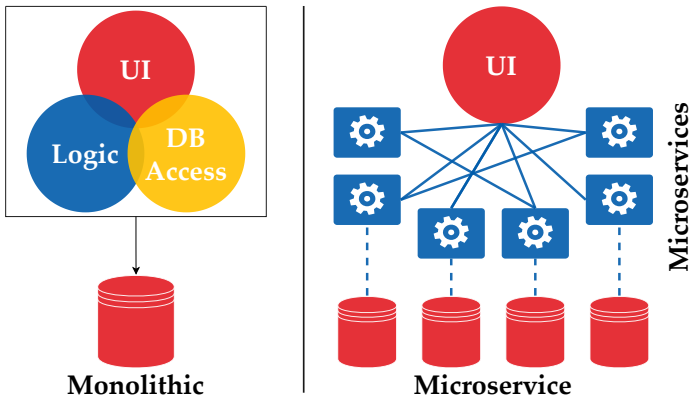


Figure 2.1: Comparison between monolithic and microservices architecture. Microservices consists of many small loosely coupled systems.

croservices have complex communication topology and inter-dependencies that make it harder to identify and mitigate Quality of Service (QoS) violations [4, 16]. A single user request may traverse through several microservices, and if any microservice in the critical path becomes a bottleneck, the end-to-end response time will increase significantly [20]. Our motivating experiments on three prototype microservices show that the same amount of CPU allocation can result in more than 250% increase in application latency based on how the resource is distributed among different microservices. Meanwhile, existing resource management techniques developed for monolithic applications with a few service layers cannot readily capture the complex microservice interactions to make effective resource allocation choices [21–24]. Nevertheless, addressing these resource management challenges for microservices is of paramount importance as an increasing number of production cloud services have been adopting microservice architectures [12, 25].

Limitation of state-of-art approaches. Owing to the growing interest, several recent works try to address the resource management challenges in microservices [12, 20, 25–27]. They focus on utilizing machine learning (ML) techniques to capture the complex relationship between microservice resources and performance.

For instance, FIRM [20] uses a combination of support vector machines (SVM) and reinforcement learning to localize root causes of SLO violations, and apply resource autoscaling to avert these violations. Sage [26], on the other hand, uses supervised training to identify dependencies between different microservices using a Causal Bayesian Network, and a graph encoder to track the QoS violating microservices to adjust their resources. However, this line of works built on ML are fundamentally limited by their extensive training requirements, both in terms of training time to capture the dynamics of the microservices and data resolution (e.g., request level traces to build dependency graphs). More importantly, to learn from the data, some ML-based techniques intentionally cause or allow SLO violations which is undesirable in production systems [20, 25–27]. Also, any changes in the microservices architecture and inter-dependencies will require retraining the system. This ML retraining can become a barrier for real world microservices applications which go through frequent software/code updates. ML retraining can also be triggered by changes in underlying cloud hardware due to server migrations and upgrades. On the other hand, the resource demand of microservices changes with the workload on a daily basis. However, existing approaches focusing on SLO violations do not directly incorporate dynamic workload in their learning [20, 25–27].

Key insights and contributions. To avoid the hurdles of the approaches mentioned above, we propose PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling), a lightweight microservice resource manager that does not need extensive training. PEMA utilizes iterative feedback-based tuning to find efficient resource allocations that satisfy the SLO. Instead of finding the best resource configuration, PEMA first allocates abundant resources to all microservices to satisfy SLO and then tries to exploit resource reduction opportunities. Allocating abundant resources for the microservices can be easily accomplished as cloud native applications enjoy a great

degree of resource scalability. The initial (and inefficient) resource allocation can be achieved using existing rule-based resource managers [28]. Using this opportunistic resource reduction approach, PEMA avoids causing intentional SLO violations as it always allocates enough resources for microservices, even when performing poorly (i.e., missing resource reduction opportunities). To enable PEMA's approach, we introduce the notion of "monotonic resource reduction" where we either reduce the resource of a microservice or keep it unchanged. In contrast, a non-monotonic resource reduction can be made through resource reduction for some microservices and resource increase for some other microservices with an overall total resource reduction (i.e., a greater total reduction than total increase). We observe that monotonic resource reductions result in a monotonic increase in response time. Hence, we can use the response time as feedback to identify resource reduction opportunities to make gradual monotonic resource changes to reach efficient allocations. In addition, based on experiments on prototype microservice implementations, we identify that we can avoid resource reduction in bottleneck services using only two microservice-level performance metrics - CPU utilization and CPU throttling time.

Our feedback-based design also allows us to seamlessly adapt a workload-aware design where we implement a novel approach of using dynamic workload ranges with a dynamic response time target. More specifically, to avoid time-consuming learning of the efficient allocation for different workload levels independently, we use dynamic ranging where PEMA starts resource allocation for a large workload range (e.g., 100$\sim$1000 requests-per-second) and then gradually splits them into smaller ranges (e.g., 100$\sim$200 requests-per-second). We retain the resource allocation learned by the parent workload range during the range split to bootstrap the tuning for the new workload range. Based on the workload, we also dynamically alter the feedback from response time to allow headroom for response time change due to workload change.

Our performance evaluation reveals that PEMA can attain resource efficiency close to the optimum [1] with high probability. We also show that PEMA can save as much as 33% resource compared to rule-based resource allocation strategies of commercially available cluster managers. We demonstrate that PEMA can seamlessly adapt to changes in microservice deployment due to changes in underlying cloud hardware. Moreover, we show that adaptability of PEMA allows its integration with opportunistic resource management where variable SLO is used for trading performance for resource savings.

Experimental methodology and artifact availability. We use three prototype microservices implementations widely used in academic research on microservices [20, 25,26]. We implement `TrainTicket` from [2] consisting of 41 microservices, `SockShop` from [1] with 13 microservices, and `HotelReservation` from [4] with 18 microservices. We deploy these services in Docker [29] containers managed by Kubernetes [30]. Our Kubernetes cluster consists of five nodes with one master node and four worker nodes. Each node is equipped with two 10-core Intel Xeon processors, and 128 GB of Memory running the Ubuntu 20.04.3 operating system. Our software artifacts are available at our GitHub repository [31].

Limitations of the proposed approach. We share our insight on the limitations of PEMA on two different fronts - the fundamental limitations in PEMA's design approach and the limitations of PEMA's current implementation. Due to its non ML-heavy approaches, PEMA's design loses on capturing complex interdependencies between microservices, and therefore is limited on the absolute best resource efficiency it can achieve. However, PEMA makes up for this loss of optimization potential through its simplicity and adaptability to change (e.g., workload variation).

[1] Optimum resource allocation refers to the minimum resource required to satisfy SLO. We describe how we identified the optimum resource allocation in Section 2.4.2.

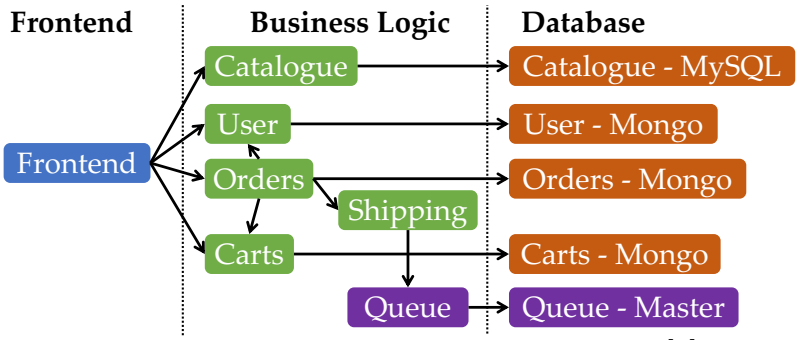


Figure 2.2: Architecture of the `SockShop` [1].

Also, due to our randomized exploration based search, PEMA offers provably efficient management and can result in arbitrarily inefficient resource allocations at times. We defer the discussion on the limitations of PEMA's current implementation to the end of our paper in Section 2.6 to make it more meaningful to the reader.

2.2 Preliminaries

2.2.1 Microservice Prototypes

`SockShop` [1]. `SockShop` implements the user-facing microservices of an e-commerce website. `SockShop`'s functionalities include searching, order placement, and shipping. Its functionalities can be divided into three parts - front-end, business-logic, and databases. The user requests arriving at the front-end are routed to appropriate microservices to serve the requests. The business-logic interact with each other and the databases as needed. The front-end is implemented using NodeJS, orders and carts microservices are implemented using Java, and the rest of the services are implemented with Go. Shipping service uses RabbitMQ to propagate messages to Queue-Master which is implemented in Java. The databases are implemented using MySQL and MongoDB. For `SockShop`, we set the SLO response time to 250 milliseconds. The overall architecture is shown in Fig.2.2.

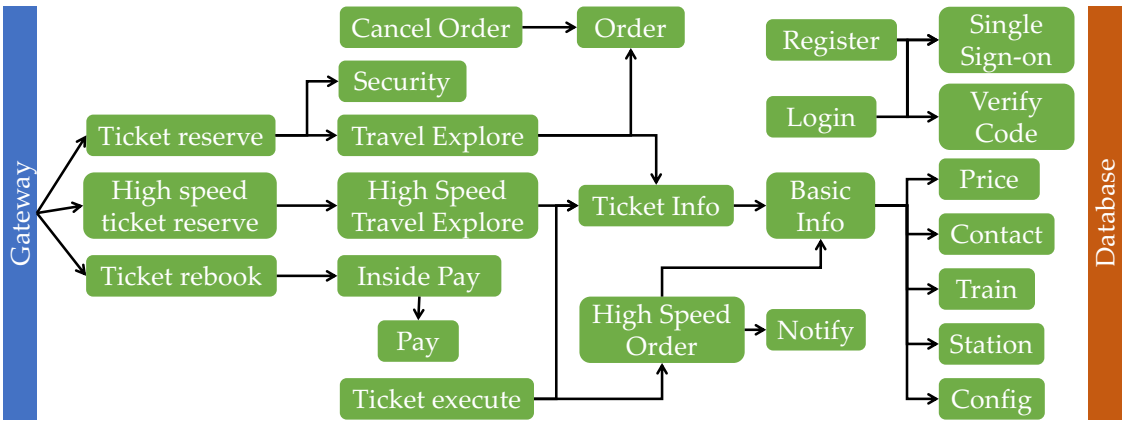


Figure 2.3: Architecture of the `TrainTicket` [2, 3].

`TrainTicket` [2]. `TrainTicket` implements a complete train ticket booking system consisting of 41 microservices. Its functionalities include ticket search with date and destination filtering, seat booking, ordering food, payment, and consignment service. The business logic of `TrainTicket` is implemented using 24 microservices divided into five layers where the microservices in the upper layers depend on the microservices of the lower layers. There are some intra-layer communications as well. The overall architecture is shown in Fig. 2.3. `TrainTicket` covers many features of microservices such as synchronous invocations, asynchronous invocations, and message queues. The `TrainTicket` business logics and front-end are built using NodeJS, Java, Python, and Go. The databases are implemented using MongoDB, and MySQL. For `TrainTicket`, we set the SLO response time to 900 milliseconds.

`HotelReservation` [4]. `HotelReservation` application is adopted from DeathStarBench microservices benchmark applications. It has 18 microservices. `HotelReservation` lets users get nearby hotel information and reserve rooms. All the services in `HotelReservation` are written in Go, and they communicate with each other via gRPC [32]. The backend uses Memcached for in-memory caching to provide faster searches while the persistent databases are implemented using MongoDB. The application is pre-populated

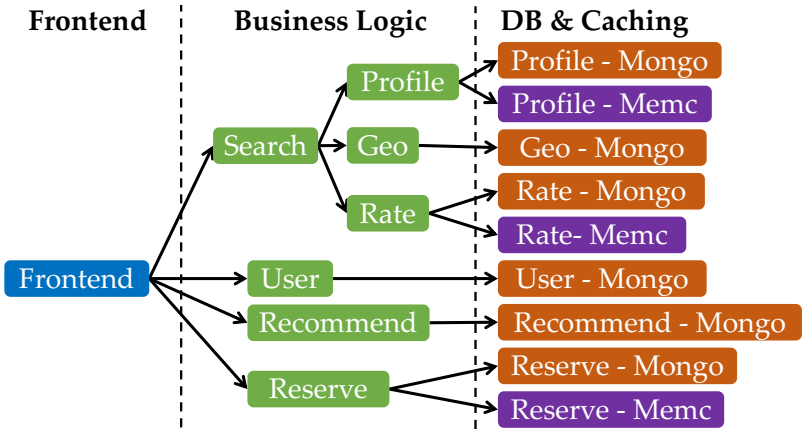


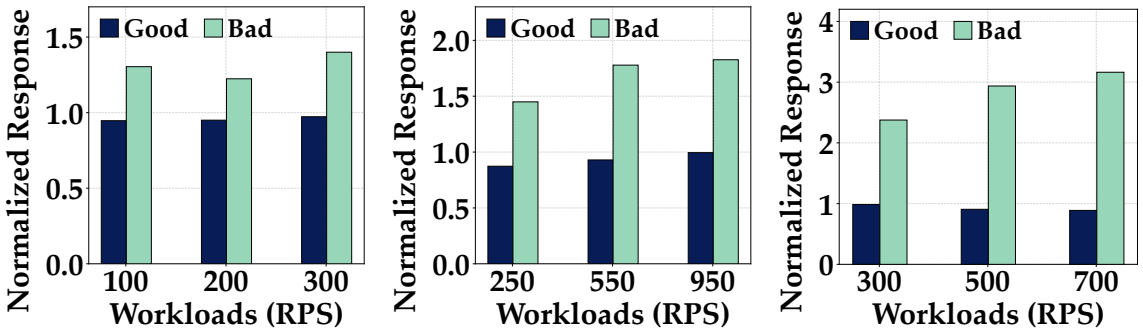
Figure 2.4: Architecture of the `HotelReservation` [4].

with 80 hotels and 500 registered users. This application consists of 18 microservices. For `HotelReservation`, we set the SLO response time to 50 milliseconds.

2.2.2 Performance Monitoring and Resource Allocation

For performance monitoring of our container-based microservice implementation, we use Prometheus [33] to collect container-specific metrics such as CPU utilization and CPU throttling. For collecting end-to-end latency performance and workload (i.e., requests per second), we use Linkerd [34]. We also use Jaeger [35] which provides detailed tracing of each request showing its service path through different microservices. Note that, our resource manager does not utilize Jaeger.

We use the 95-th percentile end-to-end response latency as a performance metric and refer to it as the application performance unless specified otherwise. For our cloud-based microservice applications which exploit request-level-parallelism, end-to-end response latency is the popular choice of performance metric [36]. For microservice resources, we only consider the total CPU allocation to a microservice with the assumption that the memory is not a bottleneck resource. Furthermore, we do not



(a) `TrainTicket` (b) `SockShop` (c) `HotelReservation`

Figure 2.5: Impact of "Good" (i.e., satisfies SLO) and "Bad" (i.e., violates SLO) resource distribution on the response time normalized to the SLO at different workloads levels. In Fig. (a), for workloads 100, 200, and 300, total CPU allocations are 40.5, 42, and 47 respectively. In Fig. (b), for workloads 250, 550, and 950, total CPU allocations are 6.3, 7.7, and 14.1, respectively. In Fig. (c), for workloads 300, 500, and 700, total CPU allocations are 5.1, 6.9, and 9.4, respectively.

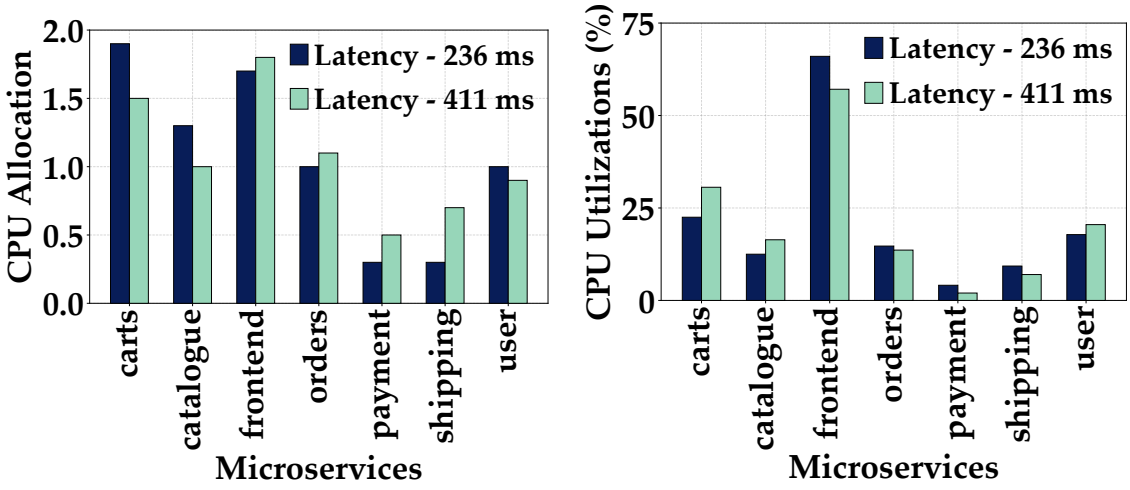
explicitly address the number of container replicas and consider homogeneous settings for each microservice.

2.2.3 Challenges in Microservice Resource Management

As in any general computing system, the performance of microservices applications depends on their resource allocation. Various theoretical and practical tools have been developed over the years to establish a mathematical relationship between computing resource and performance [17]. However, they are not equipped to capture complex interactions between different microservices. Any request's end-to-end response time (i.e., performance) is the aggregation, often non-linearly due to parallel processing, of time spent in many microservices. Consequently, the presence of any microservice with a resource bottleneck on the service path affects the end-to-end response time. Meanwhile, the resource demand for different microservices can be widely different based on their service. Hence, the distribution of resources among different microservices plays a crucial role in application performance.

To demonstrate the importance of resource distribution, we run a few experiments on our microservices prototypes. We first identify "good" resource allocations that satisfy the SLOs for the prototypes for different workload levels. We then change these to "bad" distributions by randomly altering resource allocations while keeping the total resource the same. Fig. 2.5 shows the impact of this resource distribution - even with the same amount of resources, the performance varies significantly because of changes in distribution. For, `TrainTicket` we see as much as 43.88% increase in response time while `SockShop` and `HotelReservation` suffer up to 91.3% and 256.2% increase, respectively.

Due to the large configuration space, the "good" resource distribution cannot be readily determined for microservices. Also, the nature of processing done in different microservices is different and cannot adhere to any general resource allocation principle, such as keeping utilization lower than a certain level [18, 19, 28, 37]. To illustrate this, we show the resource distributions of `SockShop`'s microservices for the good and bad configurations with the same amount of total resource in Fig. 2.6(a) and the corresponding CPU utilization in Fig. 2.6(b). We see that there is no readily identifiable root cause (e.g., microservice with bottleneck resource) in response latency in Fig. 2.6(b) for the 74% increase (236 milliseconds to 411 milliseconds). Also, while we see an increase in utilization for the `cart`, `catalogue`, and `user` services for the bad configuration, their utilization remains below the `frontend's` utilization, making it impossible to employ any common utilization-based resource allocation policy. Furthermore, we see that the utilization change due to resource change is different for different services. For example, the `frontend's` utilization changed more than `orders` even though they experienced similar resource change. This indicates that resource allocation policies that try to increase overall utilization [20], may not be the most efficient.



(a) CPU allocation

(b) Utilization

Figure 2.6: (a) Total CPU allocation of 7.5 distributed among different microservices of `SockShop`. (b) CPU utilization.

To summarize, *for efficient microservice management, it is crucial to identify how resources should be distributed among different microservices as the same amount of resources can result in significantly different performance based on which microservice gets how much resources. However, finding the efficient resource distribution is very hard as there are no easily generalizable markers (e.g., high utilization) to assist in the resource allocation.*

2.3 Design of PEMA

We have two design goals for our resource manager - (1) assure QoS (i.e., avoid SLO violations), and (2) find efficient resource allocation. Using a discrete-time model with a time step Δt (e.g., one minute) where the microservice resource allocation decisions are updated at the beginning of each time step, we formalize our re-

source management as the following optimization problem ORA (**O**ptimum **R**esource **A**llocation)

$$\mathsf{ORA}: \underset{\mathbf{x}^t}{\text{minimize}} \quad \sum_{i=1}^{N} x_i^t \tag{2.1}$$

$$\text{subject to} \quad \mathcal{F}(\mathbf{x}^t) \leq R \tag{2.2}$$

Here, at time step t, $\mathbf{x}^t = (x_1^t, x_2^t, \cdots, x_N^t)$ is the resource allocation vector of the N microservices, $\mathcal{F}(\mathbf{x}^t)$ is the end-to-end latency response of the application for resource allocation $\mathbf{x}^t$, and R is the response latency threshold defined in the SLO. In what follows, we develop PEMA (**P**ractical **E**fficient **M**icroservice **A**utoscaling) - a practical microservices resource manager that finds a provably efficient solution to ORA. We first discuss the design principles of PEMA to achieve our goals (i.e., the solution to ORA), followed by the rationale for our choices and implementation details of PEMA.

Note here that, instead of minimizing the total resource allocation, ORA can also adopt cost minimization as its goal by replacing x_i^t in Eqn. (2.1) with $\mathcal{C}(x_i^t)$ which represents the cost of resource x_i^t. Moreover, resource allocation vector $\mathbf{x}^t$ is not restricted to CPU allocations only. We can incorporate other types of cloud resources such as memory allocation and I/O bandwidth in $\mathbf{x}^t$. Nevertheless, our general solution principle still applies, albeit the opportunistic resource reductions need to be conducted on multiple resource dimensions.

2.3.1 Design Principles of PEMA

A learning-based approach. Achieving either of our design goals for a microservice-based application is non-trivial due to their complex topology and inter-dependency between different microservices. Moreover, the relation and interaction with each other for these microservices varies with applications and deployments, even among different versions of the same application. Not to mention, the under-

lying cloud hardware (e.g., processor type/model) hosting these applications also affects the microservice performance and resource allocation. Consequently, our resource manager needs to identify resource allocation strategies for each microservice implementation and at the same time be able to adapt as the application evolves. Hence, we take a learning-based approach where PEMA iteratively interacts with the application through a feedback loop to navigate towards efficient resource allocations.

Provably efficient resource allocation. Solving PEMA can be interpreted as tuning the application resources that will make the response latency exactly equal to the SLO specified level. However, since the resource distribution across different microservices affects the latency and microservice-based applications usually consist of many microservices, there could be many different resource allocations that result in a latency equal to the SLO. Consequently, in PEMA, instead of finding the best resource allocation (i.e., the lowest aggregate resource), our goal is to find a resource allocation close to the optimum with fewer iterations.

QoS preserving learning. An unwanted pitfall of the learning-based approach in the existing literature is that the system needs to learn "bad" resource allocations that cause SLO violation by causing/creating these violations [20, 25–27]. While our approach too cannot completely eradicate the possibility of SLO violations, unlike prior works, we do not cause them intentionally. Instead, we adopt a QoS conservative approach where we start from with sufficient resource for all microservices to satisfy SLO, and then iteratively search for resource reduction opportunities based on the application's performance statistics. During the search/learning, PEMA always tries to maintain latency performance better than the SLO. Moreover, we dynamically tune how much resource we reduce based on how close our performance is to the SLO and stop tuning if the performance is at the SLO level. For example, with a response time SLO of 250ms, PEMA will try to reduce more resources when the response time

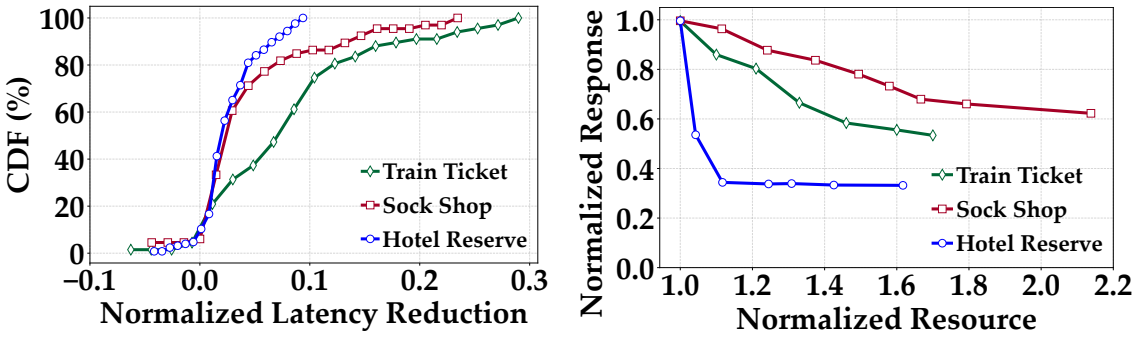
is 150ms than when the response time is 200ms. Hence, during resource allocation navigation, PEMA does not set a resource allocation to violate the SLO intentionally.

Feedback-based navigation. Starting with ample resources for each microservices to comfortably satisfy SLO, PEMA uses the difference between current application performance and the SLO as an indicator of resource reduction opportunity. However, it does not tell us on which microservice(s) we should exercise the resource reduction. Hence, PEMA uses microservice-wise performance metrics to determine the target microservices. More specifically, PEMA uses the microservice-wise performance metrics to filter out the microservices approaching their bottleneck resource configuration and then implements a randomized selection process where the probability of picking a microservice is determined by its performance metrics. With unknown relation between a microservices resource allocation with the overall application performance, a guided randomized selection allows PEMA to explore various possible combinations of resource allocation.

2.3.2 Supporting Results for Design Rationales

Here, we provide corroborating observations for PEMA's design using our prototype microservices implementations. We first show why application's performance can be a safe yet effective indicator of resource reduction, followed by how microservice-wise performance metrics can help PEMA navigate.

Gradual resource reduction for efficiency. In PEMA, we use the difference between SLO specified response time and current system response time to determine how much resource-saving opportunity is available. Our design choice is motivated by our observation that, in general, *monotonic resource changes across microservices result in monotonic changes in the end-to-end response time.* We say a resource reduction is monotonic if some microservice resources are decreased while other mi-



(a) CDF of latency change

(b) Response time change

Figure 2.7: (a) Distribution of end-to-end response time increase (normalized to SLO) due to monotonic resource reduction. (b) Change in response time (normalized to SLO) with resource (normalized to optimum).

croservices' resources are unchanged. A resource change is not monotonic if some microservices receive greater resources while some others have their resource reduced, regardless of what happens to the aggregate resource allocation. Fig. 2.7(a) shows the CDF of increase in end-to-end response time for monotonic resource reduction for our applications. Note that there is no direct relationship between resource reduction and the amount of change in response time. This is because, the same amount of resource reduction on different microservices will have different impact on the end-to-end response time. The CDF is showing distribution of latency increase for random amounts of monotonic resource reduction on random numbers of microservices at random initial (before resource reduction) resource allocations. The CDF highlights the most likely impact of a monotonic resource reduction - an increase in the response latency regardless of the state of the microservice, i.e., its total resource allocation. The CDFs also show that the opposite, i.e., response latency decreasing with resource reduction, happens an only handful of times (10.2% for `TrainTicket` and 6.1% for `SockShop`). We attribute these cases as transient anomalies based on our observation of the application's performance metric fluctuations.

The key take away from Fig. 2.7(a) is that *by making monotonic resource reductions, we can gradually increase the latency to the SLO level.* In Fig. 2.7(b), we show examples of such monotonic resource reduction steps and its impact on latency. Here, we normalize the resource to the optimum resource allocation and the latency to the SLO level. PEMA's goal in Fig. 2.7(b) is to reach coordinate (1, 1) by gradually making monotonic resource changes. Note that the resource reduction steps in Fig. 2.7(b) is not unique. Moreover, monotonic resource reduction alone does not guarantee to reach the optimum resource allocation keeping the response latency within the SLO. Instead, it offers a QoS preserving approach of navigation to find efficient resource allocation.

Microservice-wise augmentation. While the response latency tells us about the resource reduction opportunities, it does not tell us from which microservices we should reduce the resources. We need to avoid microservices that may create a bottleneck during this resource reduction. We define a microservice's "bottleneck resource" as the resource allocation that makes the microservice a bottleneck. In PEMA, we use microservice-level performance metrics to identify the microservices with imminent bottleneck resources. However, as opposed to prior works where complex machine learning models are applied to determine such bottleneck services, we use only two performance metrics - CPU utilization and CPU throttling time [38].

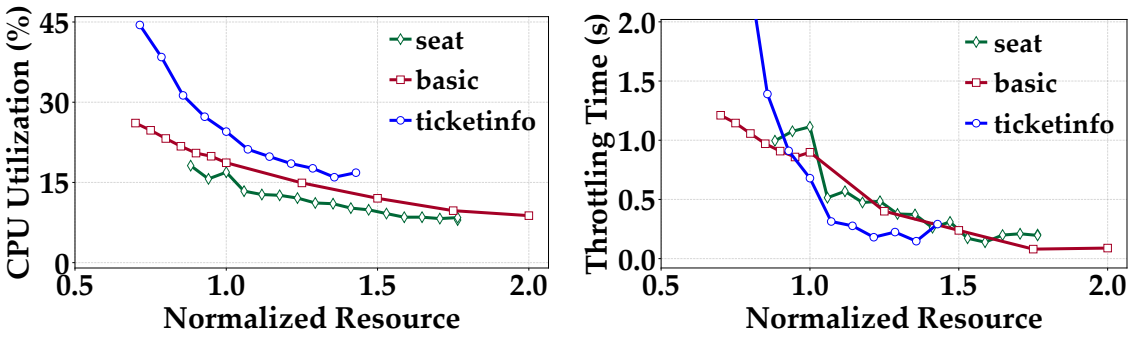
Our choice of these performance metrics is based on our experiments. We intentionally create bottlenecks and use feature extraction to identify which performance metrics can be used to identify the bottleneck services reliably. Note that these experiments are done to assist in our design. PEMA does not need any offline experiments or pre-training. For each microservice, we collect the following performance metrics - `cpu_usage_seconds_total`, `memory_usage_bytes`, `cpu_cfs_throttled_seconds_total`, Jaeger tracing - `self_time`, and `duration`. We then run classification with various

Table 2.1: Classification accuracy with CPU utilization and CPU throttling time as features to detect bottleneck microservices.

Microservice Name	Botteleneck Services	Accuracy (%)
TrainTicket	seat	94.18
TrainTicket	seat, ticketinfo	96.2
SockShop	carts	100.0
SockShop	carts, orders	98.3
HotelReservation	front-end	97.8
HotelReservation	front-end, search	95.6

combinations of the performance metrics as features. We find that, when used as the classification features, CPU utilization and CPU throttling time give us the highest classification accuracy. Table 2.1 shows the classification accuracy for different applications with various bottleneck services.

To better understand the role of CPU utilization and CPU throttling time as bottleneck indicators, we track these metrics for three different microservices in TrainTicket- seat, basic, and ticketinfo, as we reduce their resources to create bottlenecks. To identify the bottleneck, we allocate sufficient resources to all other microservices. Fig. 2.8 shows the change in CPU utilization and CPU throttling as we reduce the resource of the microservice under investigation. We normalize the microservice resource allocations to their respective bottleneck resources. We make a few important observations here. *First,* the CPU utilization (Fig. 2.8(a)) changes gradually as the microservice approaches and eventually crosses the bottleneck resource. We also see that the utilization corresponding to bottleneck is different for different microservices. For example, ticketinfo's bottleneck utilization is around 25%, whereas seat's bottleneck utilization is around 15%. *Second,* CPU throttling time changes rapidly at bottleneck resource. The bottleneck CPU throttling time also varies with microservices.



(a) CPU utilization (b) CPU throttling time

Figure 2.8: Changes in CPU utilization and CPU throttling time with resource allocation for three bottleneck microsservices in `TrainTicket`- `seat`, `basic`, and `ticketinfo`.

2.3.3 PEMA

Here we present the details of PEMA's implementation that builds on our design principles and experimental observations.

Resource reduction opportunity. In PEMA, similar to gradient descent, we start with sufficient resources for all microservices and gradually decrease their resource based on how our resource change affects the end-to-end response time. We update resource allocation in regular intervals based on the response time observed in the previous interval. Since we rely on the response time statistics, we set sufficiently long update intervals to have stable response time statistics. For instance, in `TrainTicket`, `SockShop`, and `HotelReservation`, we use update interval of two minutes. For resource reduction at time step t, we first decide the number of microservices n^t to reduce resources from using

$$n^t = N \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right), \tag{2.3}$$

where $r^{t-1} = \mathcal{F}(\mathbf{x}^{t-1})$ is the response time in the previous time step. $\alpha \leq 1$ is a user-defined non-negative parameter that determines how aggressively we want to reduce the resource. A smaller α will reduce resource more aggressively and vice versa.

Next, using similar approach as Eqn.(2.3), we decide how much resource we reduce in the n^t microservices in percentage using

$$\Delta^t = \beta \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) \cdot 100\%, \tag{2.4}$$

where $\beta \leq 1$ is another user defined parameter that decides the maximum resource reduction for any microservice in one time step. A high value of β makes PEMA aggressively change the resource between update intervals and vice versa. We analyze the impact of α and β in our evaluation in Section 2.4.3.

Using Eqns. (2.3) and (2.4), PEMA dynamically adjusts the amount of monotonic resource reduction as our response time r^t approaches SLO limit R. We can also set the values of α and β dynamically to have more aggressive reduction when $R - r^{t-1}$ is high and reduce the amount of reduction per interval as r^t approaches R. In addition, to avoid triggering resource change for transient perturbation in response time, we can keep a response time buffer by scaling down R, for instance, to 95%, in Eqns. (2.3) and (2.4).

Avoiding bottleneck services. For the i-th microservice, we denote its utilization as u_i with a bottleneck threshold U_i^{th} and CPU throttling time as h_i with a bottleneck threshold H_i^{th}. To decide the n^t candidate microservices, we first take the set of microservices that has a CPU throttling time less than their respective thresholds. We denote the set of indexes of these microservices as $\mathcal{I}^t = \{i : h_i^{t-1} \leq H_i^{th}\}$. We then normalize the utilization of each microservice in $\mathcal{I}^t$ to their respective uti-

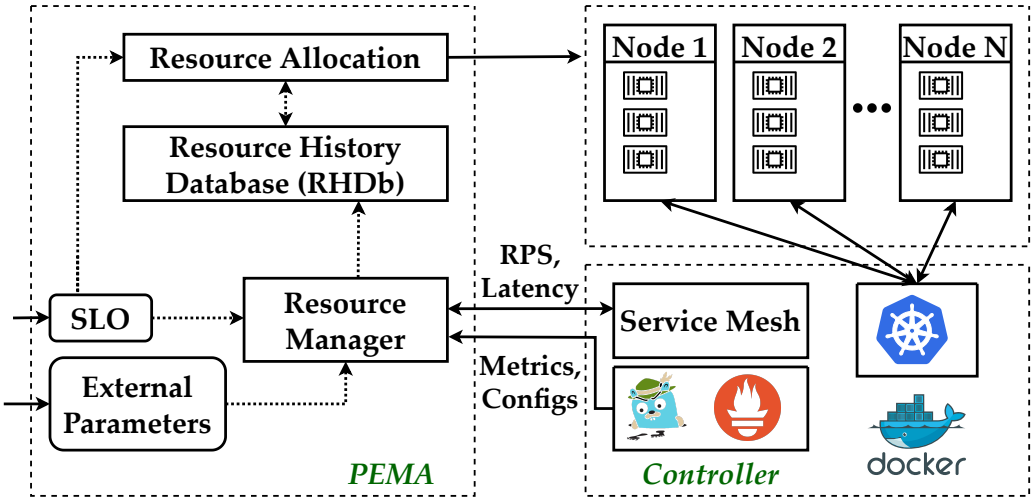


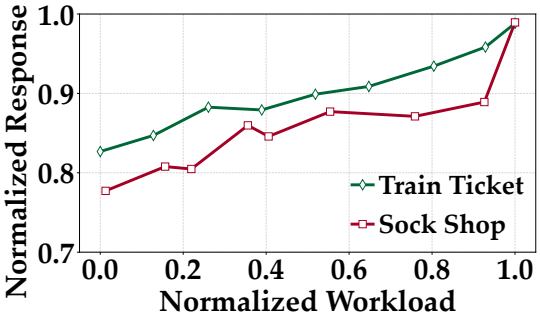
Figure 2.9: Block diagram of the PEMA.

lization threshold as $u_i^{*t-1} = \frac{u_i^{t-1}}{U_i^{th}}$ and update the probability of each microservice in $\mathcal{I}^t$ as follows

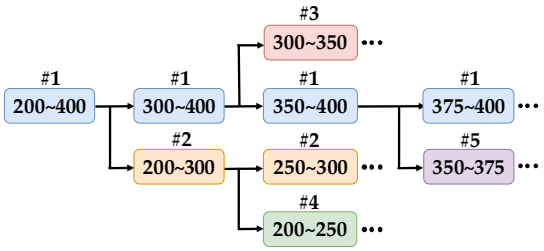
$$p_i^t = 1 - \frac{u_i^{*t-1} - \min_{i \in \mathcal{I}^t}(u_i^{*t-1})}{1 - \min_{i \in \mathcal{I}^t}(u_i^{*t-1})} \tag{2.5}$$

Here, $\min_{i \in \mathcal{I}^t}(u_i^{*t-1})$ means the minimum normalized utilization among all the microservices in $\mathcal{I}^t$. Eqn.(2.5) indicates that a microservice with utilization equal to its threshold, i.e., $u_i^{*t-1} = 1$ will result in a "zero" probability ($p_i^t = 0$), whereas the microservice with the lowest utilization, i.e., $u_i^{*t-1} = \min_{i \in \mathcal{I}^t}(u_i^{*t-1})$, will have the probability of "one" ($p_i^t = 1$). We populate a new candidate set $\mathcal{I}^{*t}$ with a inclusion probability of p_i^t for the i-th microservice. If the size of $\mathcal{I}^{*t}$ is equal to or smaller than n^t, we take the entire set $\mathcal{I}^{*t}$ and reduce each microservice in $\mathcal{I}^{*t}$ and reduce their resource by Δ^t. However, if the size of $\mathcal{I}^{*t}$ is greater than n^t we uniformly randomly choose n^t microservices from $\mathcal{I}^{*t}$.

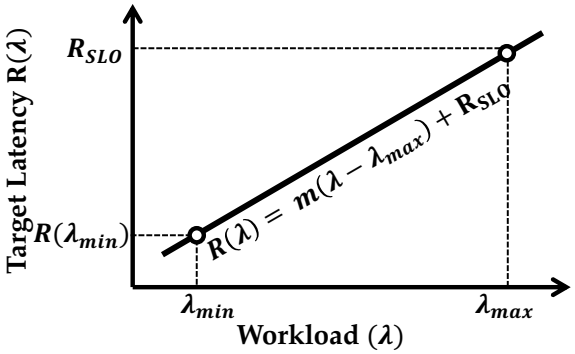
Dynamically updating bottleneck thresholds. As shown in Fig. 2.8, the bottleneck thresholds for utilization and CPU throttling time varies among microservices. Hence, we need to learn the appropriate threshold settings for each microservice. In PEMA, we begin with a conservative estimation of utilization threshold set



(a) Response time vs workload



(b) Dynamic workload range



(c) Dynamic response time target

Figure 2.10: (a) Response time change due to workload. (b) Dynamic workload range to bootstrap efficient resource allocation for different workloads. (c) Dynamically updating target response time to tackle response time change due to workload change.

at 15% and CPU throttling time threshold of "zero" (i.e., no CPU throttling) for all microservices. We expect all microservices to satisfy these thresholds as PEMA starts with ample resource allocation. Similar to our resource reduction approach, we opportunistically increase these thresholds. More specifically, at the beginning of every time step t, we update the utilization and CPU throttling time thresholds as follows

$$U_i^{th} = \max\left(U_i^{th}, u_i^{t-1}\right), \forall i \tag{2.6}$$

$$H_i^{th} = \max\left(H_i^{th}, h_i^{t-1}\right), \forall i \tag{2.7}$$

Iterative resource allocation. PEMA applies the resource reduction iteratively and saves all resource allocations, $\mathbf{x}^t$, and the response times, r^t, in a "resource allocation history database (RHDb)". The purpose of the RHDb is to allow PEMA to roll back to a previous SLO satisfying resource allocation for all microservices in case of an SLO violation. Even though the resource reduction slows down when the latency approaches the SLO, PEMA cannot guarantee that its opportunistic resource reduction will never cause an SLO violation. In addition, changes in microservice implementation or changes in its hardware configuration may also alter optimum resource allocation and cause SLO violations. In such cases, rolling back to a previous configuration allows PEMA to jump start on finding the new optimum, instead of resetting the resource allocation to the maximum and starting from scratch. While RHDb itself does not add significant overhead due to its lightweight single-table implementation, the action of rolling back may cause extra iterations for PEMA to find an efficient resource allocation. Nonetheless, the mechanism of roll back using RHDb is essential for PEMA's adaptability and QoS assurance.

Escaping sub-optimum configurations. The combination of monotonic resource reduction and probabilistic choice of microservices to reduce resource may cause PEMA to make unfavorable resource reductions early on (e.g., making particular microservice reach bottleneck and push response time close to SLO) and settle at inefficient resource allocation, even though other microservices have redundant resources. This can force PEMA to slow-down prematurely, even stop further resource reduction. To escape from such inefficient resource allocations, we implement random exploration where PEMA with a probability p_e^t rolls back to a uniformly random

previous resource allocation in RHDb. We set p_e^t based on the response latency as follows

$$p_e^t = A \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) + B \tag{2.8}$$

Here, A and B are exploration parameters that decide the maximum and the minimum probability of exploration, respectively, and satisfy $0 \leq B \leq A \leq 1$ and $A + B \leq 1$. The exploration probability decreases as PEMA's response time r^{t-1} approaches the SLO R. The random exploration also allows PEMA to "walk back" the resource reduction path it took and identify previously missed reduction opportunities. Naturally, the degree of exploration affects how quickly we reach an efficient resource allocation. Nonetheless, we do not anticipate this exploration to add significant overhead since PEMA can find an efficient resource allocation in a few tens of iterations.

Implementation of PEMA. We present the working principle in Algorithm 1 where PEMA takes performance metrics from the system using Prometheus and Linkered and then updates the resource allocation of the microservices while keeping a log of all resource allocations and response times in its database RHDb. The high-level architecture block diagram of PEMA is presented in Fig. 2.9.

2.3.4 Workload-Aware Resource Allocation

Our design of PEMA so far addresses how we can navigate to find an efficient resource allocation for our microservice-based application. Our design, through configuration rollback, can also handle changes in microservice implementation. Here we address how PEMA tackles the workload variations. For any cloud application, the workload intensity (i.e., requests per second) directly affects the response time, and hence, how much resource is needed [17–19]. In Fig. 2.10(a), we show the change in response time as the workload changes. As PEMA iteratively makes resource reduc-

tions based on the response time, a decrease in workload will falsely indicate resource reduction opportunities that do not work for high workloads, leading to many SLO violations when the workload increases. The same is true for prior ML-based approaches that do not explicitly address workload change [26, 27].

Hence, PEMA needs to identify efficient resource allocations at different workload levels. A straightforward way is to divide the workload variations into discrete workload ranges (e.g., a workload range from "X" requests-per-second to "Y" requests-per-second) and run multiple copies of PEMA in a "pseudo-parallel" fashion. We say pseudo-parallel as at any time only one PEMA is working on its corresponding workload range. Note here that the workload ranges need to be small enough to not significantly affect the response latency, requiring resource allocation changes, i.e., a single resource allocation should work for the entire range. For instance, a range of 25 requests-per-second in `TrainTicket` microservice is a suitable workload range.

Dynamic workload-range. While in principle multiple parallel PEMA works, it may take a long time to reach efficient allocations for every workload range. To accelerate the learning, we propose a novel approach where we start with a few (two/three) larger workload ranges and gradually split each range (i.e., parent range) into smaller ranges (i.e., child range) until we reach our target workload ranges. The goal here is to utilize learning from the parent ranges to bootstrap the learning process for the child ranges. During a range split, the parent range is divided into two equal child ranges. We attach PEMA of the parent range to the child range with a higher workload, whereas a new PEMA process is launched for the other child range. The new PEMA uses the resource allocations of the parent range as the starting point and requires fewer iterations to reach an efficient resource level. The intuition for this approach is that a resource allocation that satisfies SLO at a higher workload should also satisfy SLO for a lower workload. Fig. 2.10(b) illustrates the idea where

we start with a workload range of 200∼400 and then branch out to smaller ranges. The number on top of each range identifies the PEMA process attached to this range. The original PEMA process with id "#1" remains attached to the higher workload ranges (e.g., 300∼400, 350∼400, 375∼400) as we split each range into smaller ranges.

Dynamic response time target. While this approach benefits the learning time, we need to tackle the latency variation due to workload changes when the workload ranges are large (e.g., 200∼400 rps for `TrainTicket`). We use one PEMA process for each workload range, even during the initial stages with large ranges (e.g., PEMA #1 for 300∼400 range in Fig. 2.10(b)). Each PEMA process needs to make an SLO preserving resource allocation that works for its entire range. To achieve this, instead of setting it to the SLO specificity response time, we update R in Eqns. (2.3), (2.4), and (2.8) into a function of workload λ as follows

$$R(\lambda) = m \cdot (\lambda - \lambda_{max}) + R_{SLO} \tag{2.9}$$

Here, m is a parameter that determines the change in latency performance for a unit change in workload, λ_{max} is the upper limit of a workload range, and R_{SLO} is the SLO specified response time. Fig. 2.10(c) illustrates our approach of using a dynamic response time target. We see from Eqn. (2.9) that when the workload is low within a range, we set a conservative (i.e., lower than SLO) latency target to intentionally allocate more resource than needed and therefore allow headroom for higher workloads. This approach intentionally makes conservative inefficient resource allocations for lower workload levels within a range. However, as the ranges get smaller as we split them, the latency variation within a range also gets smaller, and so is the inefficiency. On the other hand, we learn m at the beginning of PEMA when we keep the resource allocation fixed for a few time steps while the workload changes. We then use linear regression on the workload vs response time (as in Fig. 2.10(a)) to

extract m. Note that we learn m only once at the beginning when the workload ranges are large. During range splits, we keep the m from the parent range. Now, m may change as we make the resource allocations change on the microservice. Nonetheless, as our range split reaches the final workload ranges, we no longer need the dynamic response target, and m becomes irrelevant.

2.3.5 Handling Transient Events

From our extended experiments we identify that PEMA is susceptible to unnecessary SLO violations due to transient dips in the response time. More specifically, after PEMA has already identified an efficient allocation, a momentary/transient dip in response time drives PEMA to make resource reductions only to meet with SLO violation in the next iteration. To circumvent this, we adopt a moving average approach where we take the average of the response time of K recent time steps and update the n^t and Δ^t as follows

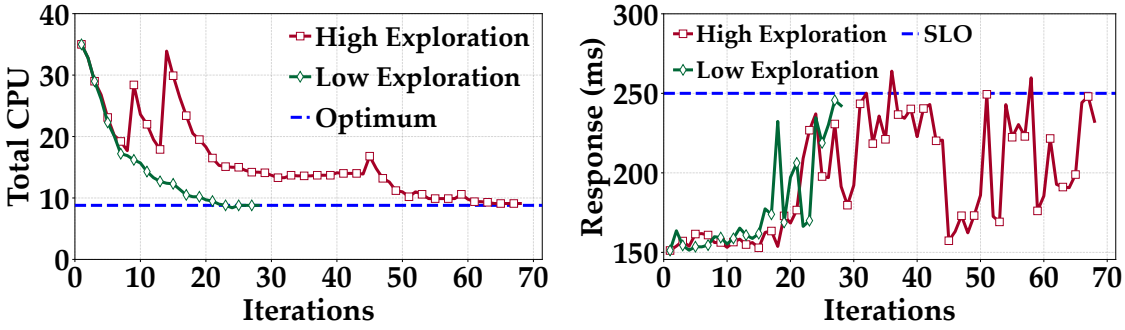
$$n^t = N \cdot \min\left(\frac{R - \frac{1}{K}\sum_{k=1}^{K} r^{t-k}}{\alpha R}, 1\right) \tag{2.10}$$

$$\Delta^t = \beta \cdot \min\left(\frac{R - \frac{1}{K}\sum_{k=1}^{K} r^{t-k}}{\alpha R}, 1\right) \cdot 100\% \tag{2.11}$$

Note that, to ensure QoS, we do not apply this moving averaging for detecting SLO violations. We still roll back resource allocations based on the most recent response time as in Line 4 in Algorithm 1.

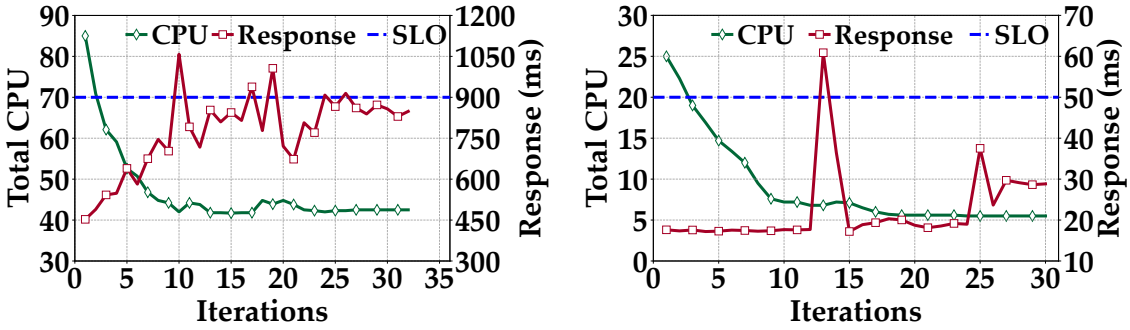
2.4 Evaluation

We use our microservice application prototypes, `TrainTicket`, `SockShop`, and `HotelReservation`, to evaluate PEMA. Here we first discuss details of PEMA's execution followed by performance evaluation against other resource allocation strategies.



(a) CPU allocation (b) Response time

Figure 2.11: Execution of PEMA on `SockShop` with different explorations. The exploration parameters in Eqn. (2.8) for high exploration are $A = 0.1, B = 0.01$, and for low exploration are $A = 0.05, B = 0.005$.



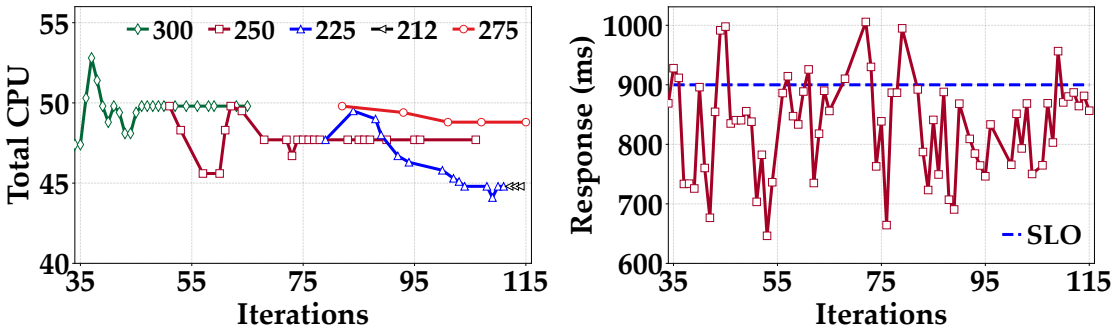
(a) `TrainTicket` (b) `HotelReservation`

Figure 2.12: Execution of PEMA for `TrainTicket` and `HotelReservation`.

We then present how different parameters affect PEMA, and finally show how PEMA can adapt to change in operating conditions.

2.4.1 Execution of PEMA

Here, we first show how PEMA finds efficient resource allocation using iterative resource reduction, where the duration of each iteration is two minutes. We then demonstrate how workload-aware PEMA utilizes the dynamic workload range and response time target. Finally, we present a 36-hour long experiment with PEMA making efficient resource allocation maintaining QoS.



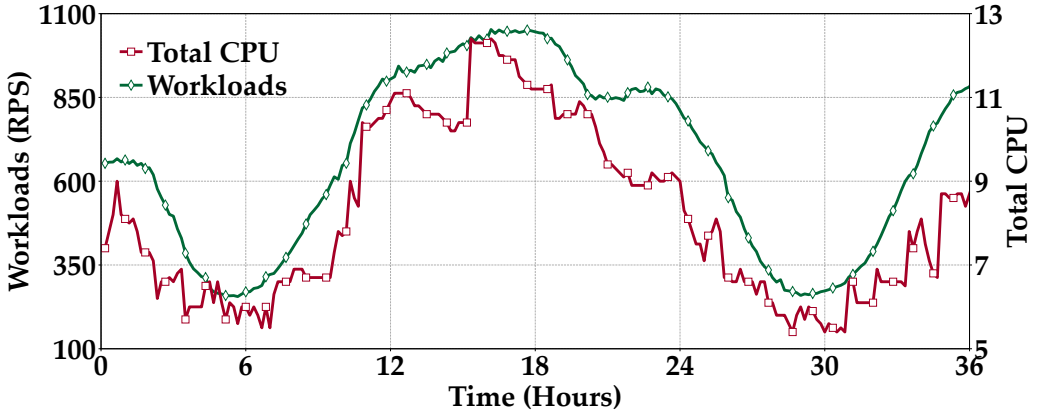
(a) Total CPU allocation over different ranges (b) Response time of the iterations

Figure 2.13: Execution of PEMA on `TrainTicket` with dynamic workload range. (a) CPU allocation. (b) Response time.

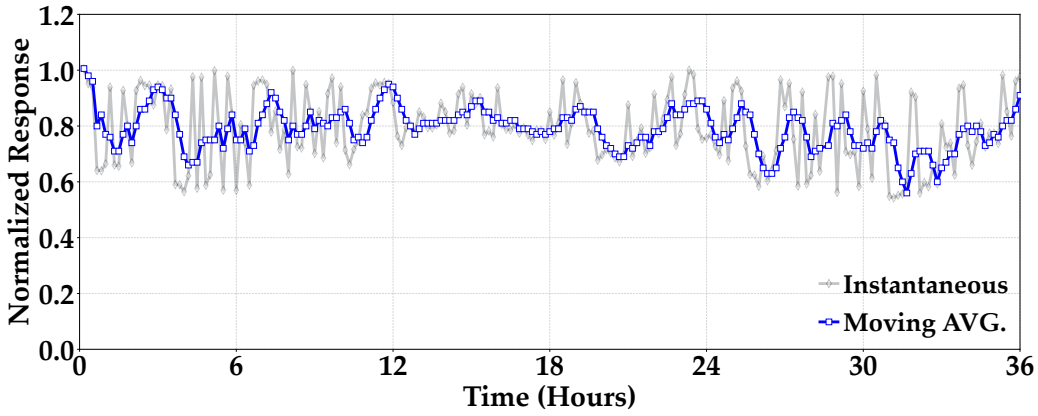
Efficient resource allocation. Fig. 2.11(a) demonstrates the iterative resource allocation and Fig. 2.11(b) shows the corresponding response times for `SockShop` under a workload of 700 requests per second for two different sets of exploration parameters. Here, the optimum total CPU allocation is 8.8 which is identified using extensive trial and error.

We see in Fig. 2.11(a) that when a higher exploration is used, PEMA intentionally increases the resource allocation twice around iteration 10 by going back to an older and higher CPU allocation. We also see that PEMA with high exploration settles at an inefficient allocation after 20 iterations as the response time reaches SLO (Fig. 2.11(b)). However, due to exploitation, we see that around iteration 45, it rolls back to an older allocation and finds its way to the efficient allocation. Incidentally, PEMA with low exploration also reaches the efficient resource allocation. We see a few SLO violations in Fig. 2.11(b) which are mitigated immediately by increasing the CPU resource. Figs. 2.12(a) and 2.12(b) show the iterative resource change and the corresponding response times for `TrainTicket` and `HotelReservation`, respectively.

Regardless of the microservice implementation, we see that PEMA can successfully find efficient resource allocations with only a few unintentional SLO violations.



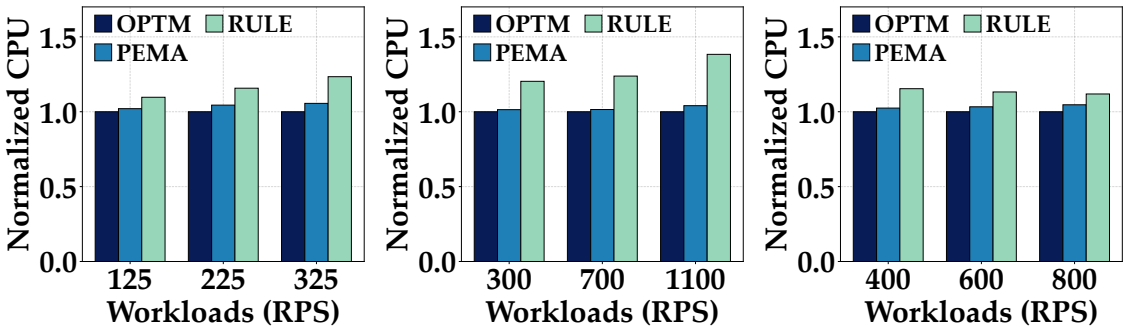
(a)



(b)

Figure 2.14: Extended execution of PEMA in `SockShop`. (a) Workload and CPU allocation. (b) Response time normalized to SLO.

Dynamic workload range. Next, in Fig. 2.13(a), we show the resource allocation of PEMA for `TrainTicket` as our workload varies between 200 and 300 requests per second. The legend in this figure indicates the upper limit on the workload range. The workload range 300 (i.e., 200∼300) first splits into ranges 300 and 250 around iteration 50. The 250 range splits into 250 and 225 around iteration 80, while the 300 splits into 300 and 275 right before iteration 85. We see that each workload range finds an efficient allocation within a few iterations as they start from an already good



(a) `TrainTicket` (b) `SockShop` (c) `HotelReservation`

Figure 2.15: Performance comparison of PEMA against optimum (OPTM) and commercial autoscaler (RULE). The CPU allocation is normalized to that of OPTM. PEMA is close to optimum and saves up to 33% resource compared to RULE.

allocation. Fig. 2.13(b) shows the corresponding the response time. We see some SLO violations, which are mitigated by PEMA.

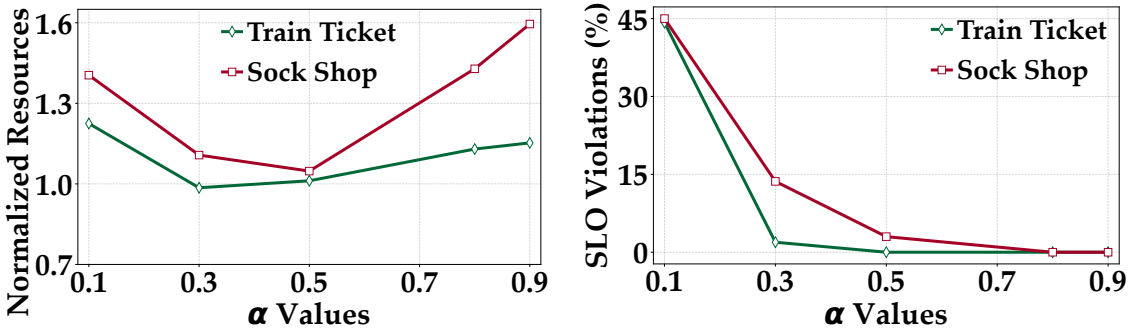
Extended execution. We run PEMA on `SockShop` for 36-hour where we change the workloads between 200 and 1100 requests per second following the workload pattern of Wikipedia collected from [39]. Fig. 2.14(a) shows the workload pattern and the corresponding resource allocation. We see that PEMA varies the total resource allocation with changing workload to maintain efficient allocation. Note here that simply varying scaling resource allocation based on workload does not work on microservices as the distribution of the resource plays an important role in performance. Fig. 2.14(b) shows the corresponding response times. We show both the instantaneous (i.e., most recent) and moving average responses with a window size of five. Recall that PEMA reduces resources based on the moving average to avoid transient changes while tackling SLO violation based on the instantaneous response time.

2.4.2 Performance evaluation

Benchmark strategies. We compare the resource allocation efficiency of PEMA against two benchmark strategies - optimum (OPTM) and rule-based (RULE). In OPTM, we use an exhaustive trial and error search to identify the best possible resource allocation. We identify a resource allocation as optimum if a small resource reduction (in our case 0.1 CPU) in any of the microservices results in a SLO violation. Note that, OPTM cannot be used in practice as it causes many SLO violations during trial and error. It acts as the upper limit of resource efficiency achievable by any resource manager. RULE is Kubernetes' rule-based resource scaling [40]. We chose RULE as a commercially available resource allocation algorithm to gauge PEMA's efficiency improvement. We do not compare PEMA to the ML-based resource allocation strategies as they do not focus on resource allocation efficiency.

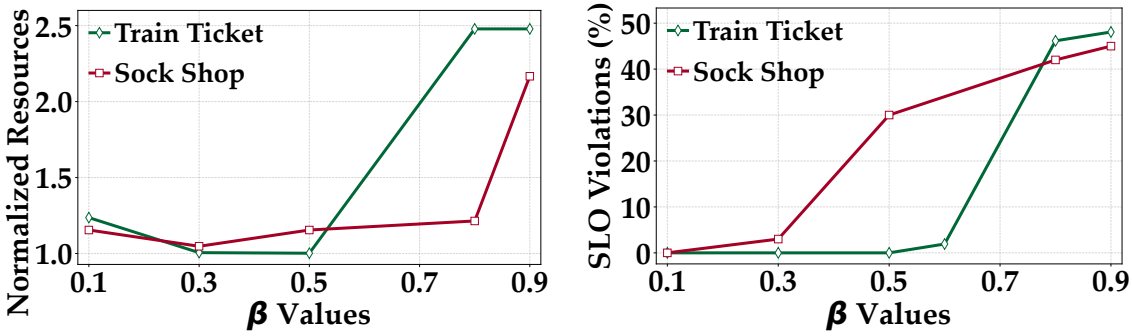
Comparison of resource allocation efficiency. We run each of the three microservices applications using PEMA and the two benchmark algorithms. Since OPTM requires extensive manual search, we evaluate these algorithms for three different workload levels for each microservice. Also, since PEMA is provably efficient, we run PEMA several times under each setting and show the average resource allocation. We normalize each resource allocation for each workload level using the resource allocation of OPTM.

Figs. 2.15(a), 2.15(b), and 2.15(c) show the resource allocations of `TrainTicket`, `SockShop`, and `HotelReservation`, respectively for the three different algorithms. We see that PEMA's resource allocation efficiency is very close to OPTM. We also observe that PEMA's efficiency drifts away with increasing workload. On the other hand, PEMA consistently beats RULE, saving as much as 33% on resource allocation for `SockShop` at high workloads.



(a) Resource allocation (b) SLO violations

Figure 2.16: PEMA's sensitivity to α for a $\beta = 0.3$ (a) Resource allocation normalized to optimum. (b) SLO violations.



(a) Resource allocation (b) SLO violations

Figure 2.17: PEMA's sensitivity to β for a $\alpha = 0.5$ (a) Resource allocation normalized to optimum. (b) SLO violations.

The performance comparison results demonstrate that despite being a lightweight resource manager, PEMA can deliver close to optimum resource allocation while retaining its capability to tackle workload variation without any significant overhead (e.g., ML training).

2.4.3 Parameter Sensitivity

Here we study how the two parameters α and β affect PEMA. Recall that α in Eqn. 2.3 determines how aggressively we reduce resource - smaller α makes PEMA reduce more resource for the same difference between response time and SLO. β,

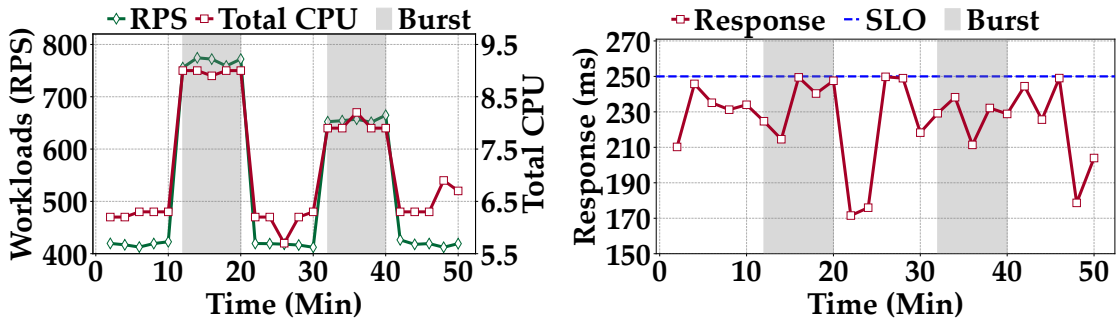
on the other hand, determines the maximum percentage resource reduction in each resource update iteration - smaller β results in smaller resource change and vice versa. For this study, we run experiments on `TrainTicket` and `SockShop` with workload 225 and 700 requests per second.

In Fig. 2.16(a), we show the change in resource allocation and in Fig. 2.16(b), we show the number of SLO violations as we change α. During this experiment, we keep $\beta = 0.3$. We see that both smaller and larger values of α result in sub-optimal resource allocations for `TrainTicket` and `SockShop`. This is because, for small α, PEMA is too aggressive making many SLO violations (as seen in Fig. 2.16(b)) and force to revert back to inefficient allocations. For high α, on the other hand, PEMA is slowed down prematurely at inefficient allocations, although it suffers much fewer SLO violations.

Next, in Figs. 2.17(a) and 2.17(b), we show the impact of change in β while we keep $\alpha = 0.5$. Similar to our observation for α we see that aggressive resource reduction due to higher values of β results in sub-optimal resource allocation while also suffering from many SLO violations. While PEMA is somewhat sensitive to both α and β, we can set α and β for any system by tuning based on SLO violation. We can take a conservative approach, start with large α and small β, and gradually change their values keeping a close eye on the SLO violations.

2.4.4 Adaptability

Workload bursts. PEMA can seamlessly handle sudden changes in workload. In Fig. 2.18, we show how PEMA handles workload bursts for `SockShop` by switching the resource allocation to the workload range corresponding to the workload burst. Here, we consider PEMA has already traversed through the resource reduction iterations for all workload ranges. As shown in Fig. 2.18(a), we create two workload burst



(a) (b)

Figure 2.18: Operation of PEMA with bursty workload in SockShop. (a) Workload and CPU allocation. (b) Response time.

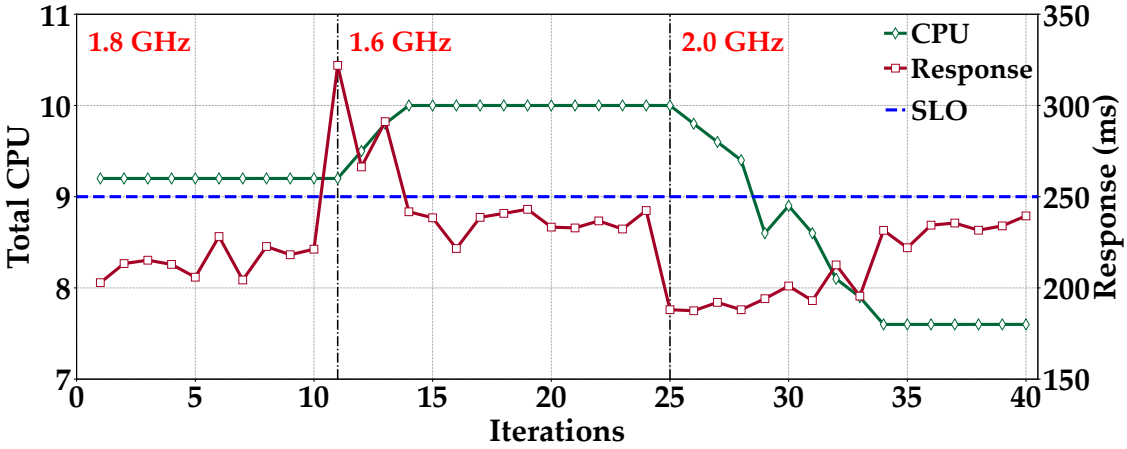


Figure 2.19: Adaptability of PEMA to changes in CPU speed for SockShop. The CPU speed change represents hardware or software updates that alters the resource demand.

of 10 minutes where the workload shoots up from 400 RPS to around 750 RPS and 650 RPS. We see that PEMA quickly changes the CPU allocation to keep the response time below SLO (in Fig. 2.18(b)). Note here that, since we update the resource allocation every two minutes, PEMA can react to a workload burst lasting less than two minutes. Nevertheless, we can adapt PEMA to respond to short-lived workload bursts by reducing the resource update interval.

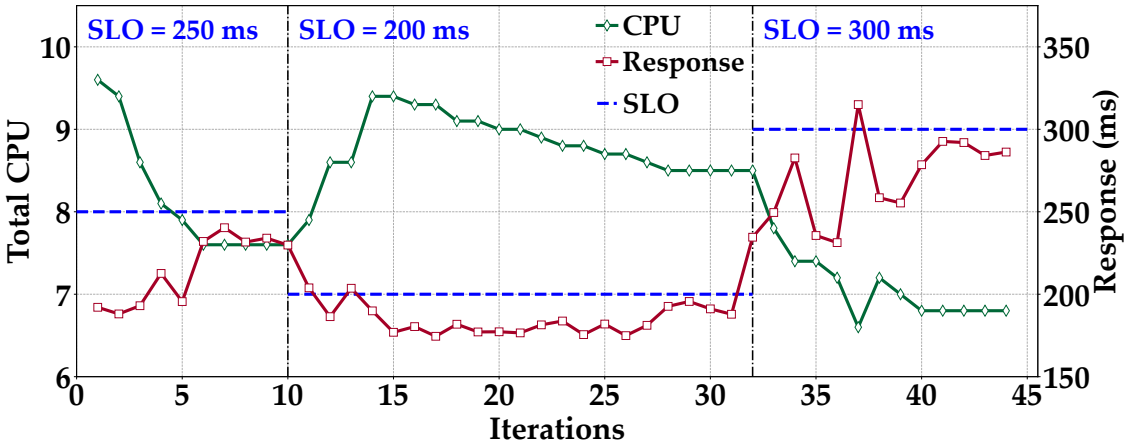


Figure 2.20: Adaptability of PEMA to changes in SLO for `SockShop`. Dynamic SLO can be used to trade performance for resource savings.

Operating environment. Our PEMA's lightweight design enables adaptability to operation condition changes. Such changes may lead to different response times even when the resource allocation is not altered. We change our server's CPU clock speeds from 1.8 GHz to 1.6 GHz and 2 GHz. These changes mimic a real-world scenario where a hardware or software change in the microservice alters the resource allocation dynamics. While we make the clock speed changes, we use PEMA to manage `SockShop`'s resource. A change in CPU frequency essentially changes the resource requirement for satisfying the SLO. Fig. 2.19 shows the CPU allocation and the corresponding response time as we change the CPU frequency. We see that PEMA can successfully change the resource allocation to satisfy the SLO demonstrating its capabilities to adapt.

Dynamic SLO change. In Fig. 2.20, we show that PEMA can also navigate towards efficient resource allocation as we change the SLO. Dynamically changing SLO can be a useful approach for applications that are willing to trade performance for resource savings to meet long-term goals such as cost budget [41]. Dynamic SLO essentially adds another control knob for managing the microservices application.

Unlike existing ML-based microservice managements, which will need to retrain with new SLO, PEMA can quickly adapt to SLO changes and tune the resource accordingly.

2.5 Related Works

Microservice autoscaling. Resource autoscaling has been extensively studied in the public cloud domain [17, 42–45]. The recent advancement of microservices has attracted a similar interest in autoscaling of microservice-based applications in academic settings [46, 47], as well as industrial settings [18, 28]. These autoscalers implement rule-based approaches in resource management. For example, Kubernetes [28] uses 90-th percentile resource usage in recent samples to set CPU and memory allocations with a 15% overprovisioning. Google Autopilot [18] uses 95-th percentile for CPU and maximum for memory in the recent samples as a marker for resource allocation in the upcoming interval. Alternative to the rule-based approach, Google also uses ML-based autoscaling using a combination of reinforcement learning and time series analysis [48]. [49] also proposes rule-based autoscaling based on CPU and memory utilization. However, rule-based autoscaling requires deep application knowledge to set up the thresholds that can vary with application. Meanwhile, [46] proposes hybrid autoscaling based on analytical modeling using a layered queue network.

SHOWAR [50], in spirit, is the closest to our design approach. It uses the variance in historical usage for vertical scaling and a proportional-integral-derivative (PID) controller for horizontal scaling. Nonetheless, SHOWAR still requires extensive tracing from the CPU scheduler for its scaling decision. On the other hand, similar to our opportunistic resource reduction, [51] utilizes "resource deflation" where preemptible virtual machines' resources are dynamically controlled. However, while

resource deflation gives away transient resources to avoid preemption, we use resource reduction as a mean to find efficient allocation by carving redundant resources.

SLO oriented resource management. In another line of work, ML-based approaches are used to identify and mitigate root causes of SLO violations in microservices [12, 20, 25–27]. For example, Sinan [25] uses a neural network to estimate short-term performance and a boosted trees model to estimate long-term performance to make per tier resource allocation. Sinan allows SLO violations to identify corner cases for resource allocation. Seer [27] requires fine-grained tracing for building its model and SLO violating cases to train its deep neural network to identify QoS violations. AlphaR [12], on the other hand, uses neural graph networks to capture the complex relationship between microservices and estimate application performance for resource allocation. Despite their impressive results in capturing minute details of microservices, they heavily depend on data and are slow to dynamically changing conditions for microservices. In designing PEMA, we depart from using complicated ML models and instead trade capturing microservice details for agility and adaptability in resource management.

2.6 Concluding Remarks

In this paper, we proposed PEMA, an iterative feedback-based approach to autoscaling microservices. PEMA is lightweight as it only requires the applications end-to-end performance and microservice-level CPU utilization and CPU throttling to navigate to efficient microservice resource allocation. Utilizing the lightweight design, we also developed a novel approach of dynamic workload-ranging to make workload-aware resource allocation with PEMA. Using three prototype microservice implementations, we showed that PEMA can achieve a performance close to the opti-

mum resource allocation and save as much as 33% resource compared to commercially used rule-based resource allocation.

Limitations of PEMA's current implementation. PEMA's implementation has several limitations that we plan to address in its future iterations. First, when PEMA causes an unintentional SLO violation, it rolls back the resource configuration in the next time step. Hence, the application suffers from bad performance during the entire resource update interval (e.g., 10 minutes). PEMA can be improved by implementing higher resolution performance monitoring (e.g., within 10 seconds), catching the SLO violations early, and rolling back configuration to mitigate it. Further, PEMA rolls back the configuration to the most recent configuration without SLO violation. It does not take into account the degree of SLO violation. For instance, a QoS violation where the response time is significantly higher than the SLO indicates that PEMA should roll back the configuration farther into the past to allocate more resources. On the other hand, while PEMA logs the resource allocation of all microservices and response times in its allocation history database, RHDb, for rollback and exploration purposes, it does not utilize this information in its decision. Finally, PEMA in this study only considers CPU resource allocation meanwhile memory and I/O resources allocation can also be important for microservices' performance depending on the nature of the application. Moreover, PEMA also does not explicitly address the impacts and trade-offs among vertical (i.e increasing resource in one node) and horizontal (i.e., increasing the number of nodes) resource scaling.

2.7 Acknowledgments

This work is supported in parts by the US National Science Foundation under grant number CNS-2104925.

Algorithm 1 PEMA

Input: SLO (R), affinity for resource reduction (α), maximum resource reduction limit (β), bottleneck utilization (U_i^{th}), and bottleneck CPU throttling time (H_i^{th}) for all microservices, exploration probability parameters A and B

Output: Resource allocation ($\mathbf{x}$)

1: **for** each time-step t **do**

2: **Performance metrics:** Collect end-to-end response time (r^{t-1}), CPU utilization u_i^{t-1}, and CPU throttling time h_i^{t-1}.

3: **Database update.** Insert x_i^{t-1}, r^{t-1}, U_i^{th}, and H_i^{th} to resource allocation history data base with key $t-1$.

4: **Handling SLO violation.** If $r^{t-1} > R$, update resource allocation to configuration from the resource allocation database with minimum resource and no SLO violation. Go to Line 11.

5: **Updating bottleneck thresholds.** For all microservices, update bottleneck thresholds for utilization, U_i^{th}, and CPU throttling time, H_i^{th}, following Eqns. (2.6) and (2.7), respectively.

6: **Exploration.** With a probability p_e^t defined in Eqn. (2.8), update resource allocation, $\mathbf{x}^t$ to a randomly chosen configuration from database without SLO violation. Go to Line 11.

7: **Resource reduction targets:** Determine number of microservice for resource reduction, n^t, using Eqn. (2.3) and resource reduction target for each microservice, Δ^t using Eqn. (2.4).

8: **Avoid bottleneck services:** Get the set $\mathcal{I}^t$ of microservices that do not exceed CPU throttling time threshold.

9: **Microservice-wise augmentation:** Build a new set $\mathcal{I}^{*t}$ from microservices in $\mathcal{I}^t$ with an inclusion probability of p_i^t defined in Eqn. (2.5).

10: **Resource reduction:** If $|\mathcal{I}^{*t}| > n^t$, uniformly randomly choose n^t microservices from $\mathcal{I}^{*t}$, else choose all microservices from $\mathcal{I}^{*t}$, and then update their resource to $x_i^{t-1} \cdot \Delta^t$.

11: **end for**

CHAPTER 3

Market Mechanism-Based User-in-the-Loop Scalable Power Oversubscription for HPC Systems

3.1 Introduction

Motivation. Advances in high-performance computing (HPC) systems have enabled scientists to perform large-scale computations quickly and efficiently. However, with the increasing computational requirement, HPC power consumption has also increased tremendously. The top supercomputers currently consume power in the megawatts range [52, 53]. Massive power consumption remains a central challenge as we move towards exascale and zettascale computing [54, 55].

Addressing the significant power consumption of HPC systems requires the adoption of energy-efficient techniques. *Power oversubscription* is a useful scheme to increase utilization by fitting the HPC system with more computing resources than its capacity. Power oversubscription has been widely adopted in hyperscale data centers of the likes of Google, Facebook, and Microsoft [56–61], which oversubscribes by as much as 20% [56]. Meanwhile, HPC systems are rife with oversubscription opportunities as these typically suffer from even greater underutilization. Approximately 30% of the power in mid-scale HPC systems remain underutilized [62], while in large-scale HPC systems, about $15 - 40\%$ of the power is never utilized [63]. This underutilization, however, is not due to a lack of demand but due to the HPC's highly specialized usage. To that end, *power oversubscription can reclaim unutilized power capacity and allow HPC expansion without additional infrastructure investment.*

Limitations of existing approaches. Power oversubscription comes with an unwanted pitfall of introducing the possibility of system overload (i.e., power consumption exceeding capacity). Several recent studies propose "power-aware job scheduling" where the HPC job scheduler allocates resources to keep the power consumption within the power budget while also targeting various efficiency improvements, such as increasing the overall system utilization, increasing throughput, and reducing job runtime [63–69]. However, optimizing such job scheduling with a peak power budget is a combinatorial bin-packing problem that is very hard to solve efficiently [70]. Moreover, the resulting power consumption from resource allocation varies depending on the job's characteristics. Not to mention, HPC jobs also go through different phases that consume different amounts of power [65–67]. Hence, power-aware scheduling faces the daunting task of estimating the power consumption over the period of each job's execution while also tracking phases of these jobs' progressions. Furthermore, HPC managers trying to maximize the system's performance (e.g., throughput) also need to consider different jobs' varying resource efficiency (e.g., work done per unit resource) during job scheduling. Hence, while existing approaches can proactively avoid overloading an oversubscribed HPC system, they also add a significant burden on job scheduling. More importantly, prior works on HPC oversubscription do not incorporate the HPC users whose job performance is adversely affected (because of power constraints) by oversubscription.

Our contribution. In stark contrast to proactive approaches, we propose a "reactive" approach for managing oversubscribed HPC systems. In our approach, the HPC manager allows the system's power to go beyond the power capacity, causing infrastructure overload. And when such an overload occurs, the HPC manager reduces the system's power consumption to mitigate it. The rationale for this reactive approach is that *first,* the HPC manager can quickly and reliably cutback the power

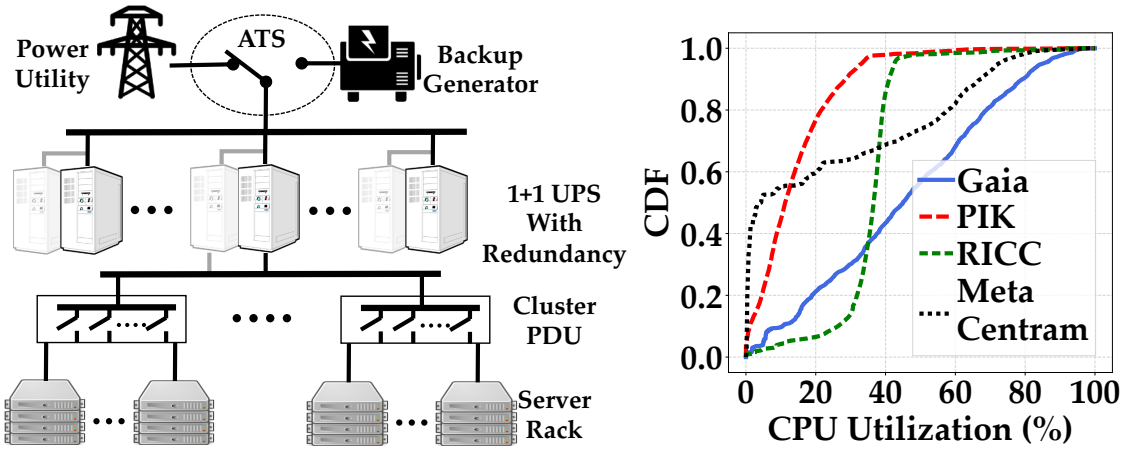
utilizing existing power management techniques such as dynamic voltage frequency scaling (DVFS) and hardware power capping (e.g., Intel's Running Average Power Limit (RAPL) [71]). Such power capping techniques are also available for modern heterogeneous computing architectures with accelerators, such as the `nvidia-smi` tool for Nvidia GPUs [72]. *Second,* we allow overloads by a relatively small margin as the maximum overload depends on the level of oversubscription (e.g., 20%). With such level of overloads, protective circuit breakers operate in the "long-delay" zone and take several tens of minutes before tripping [73–75]. Meanwhile, HPC data center cooling can also withstand these short-lived (HPC manager reacts to mitigate the overload) overloads due to thermal inertia [76]. In fact, reactively handling power overload is the norm in the cloud data centers [59, 77]. Therefore, we consider that *reactively handling power overloads is a safe approach for managing oversubscribed HPC systems.*

While reactively mitigating power overloads in HPC systems is a viable approach, the HPC manager still needs to decide how to best exercise the power reduction. The power reduction essentially translates into resource reduction (e.g., reduced CPU speed) for the active jobs executing in the system. Hence, overload handling adversely affects the active jobs' performance, and the HPC manager needs to judiciously apply power capping for a graceful power reduction with the minimum performance impact. This, however, brings us back to the challenges of job characteristics profiling of power-aware scheduling and requires the HPC manager to know the impact of resource reduction for every active job.

To avoid this burden on the HPC manager, we propose a market-based approach where the HPC users themselves determine the performance impact and actively participate in making the resource reduction decision during an overload. More specifically, we propose MPR (Market-based Power Reduction), where the users sup-

ply, in exchange for incentives/rewards (e.g., free HPC core-hours) from the HPC manager, the necessary resource reduction for handling the overloads. The users use a parameterized supply function to express how much resource they are willing to reduce at what price (i.e., incentive per unit reduction). The HPC manager acts as the market facilitator, and the market outcome determines which job will reduce how much resources and what would be the incentive for the resource reduction. We develop a static market and an interactive market for MPR. The static market offers rapid market decision, while the interactive market guarantees socially optimum power reduction with minimum performance degradation. We also develop strategies that the user can adopt to participate in these markets.

Merits of our approach. (❶) Our reactive approach frees the HPC scheduler from requiring job-wise power estimation and execution tracking. Instead, the HPC manager tracks the system's instantaneous total power consumption to detect overload and uses MPR to reduce power consumption. (❷) MPR brings the user in the loop in managing an oversubscribed HPC system. In MPR, users can integrate their own "perceived value" of performance (i.e., the same amount of performance loss can be valued differently by different users) in the resource reduction process - a user who values their performance more can ask for a greater incentive for resource reduction and vice versa. Such integration of user preference is not available in any existing work on managing an oversubscribed HPC system. (❸) MPR also offers a highly scalable HPC management solution as the HPC manager no longer needs to solve complicated scheduling problems with many variables (e.g., each job's resource allocation). Instead, she only decides the market-clearing price that ensures the active jobs supply the target amount of resource/power reduction. (❹) Finally, by empowering users to influence the HPC system's power consumption through the market mechanism, we believe MPR's user-in-the-loop approach can go beyond han-



(a) HPC power architecture

(b) CDF of HPC utilization

Figure 3.1: (a) HPC power architecture. Here, ATS = Automatic Transfer Switch, UPS = Uninterrupted Power Supply, PDU = Power Distribution Unit. (b) CDF of four real-world HPC cluster workloads [5].

dling power oversubscription. For instance, users can also assist in socially responsible HPC management, such as cutting carbon emissions by doing less work with "dirty" power with low/no renewable [78] and participating in demand response to improve the grid's stability [79].

Evaluation of MPR. We extensively evaluate MPR using several real-world trace-based simulations using the performance profiles of fourteen different HPC applications. We demonstrate that MPR can effectively handle power overloads while capturing the users' willingness for resource reduction.

We show that, by participating in MPR, a user always gets more incentive than its cost of performance loss, while the HPC manager enjoys orders of magnitude more resource gain than her incentive payoff to the users. Finally, to demonstrate MPR's effectiveness in real life, we run experiments on a prototype HPC system and show that MPR can effectively mitigate overloads due to oversubscription.

3.2 Background

HPC power system. As illustrated in Fig. 3.1(a), HPC data centers typically use a hierarchical power infrastructure [59, 60]. The utility power is delivered to the data center through an automatic transfer switch (ATS) that switches its power source to the backup generator if the utility power fails. The ATS feeds the UPS (uninterrupted power supply) which is responsible for supplying power while the generator warms up to takeover followed by a utility failure. The UPS typically needs to supply power for two to three minutes. There could be multiple UPSs working in parallel or active/redundant modes. For large HPC systems (e.g., 10-MW systems), the power infrastructure can be divided into multiple pieces, each with dedicated UPSs. The UPS powers the cluster PDUs (power distribution units) which supply power to the server racks.

Power oversubscription. In our context, oversubscription is permanently adding more servers than the HPC power infrastructure's capacity allows. Each layer of the HPC power infrastructure, from the server rack to the ATS/UPS, is subject to capacity limits and can be oversubscribed. However, we focus on UPS-level oversubscription while considering the cluster PDUs and racks have adequate capacity. We choose this as UPS is typically the dominant contributor in a data center's per kilowatt capital cost for its power system [80, 81]. Oversubscribing an existing HPC data center would mean that we add additional server racks connected to an existing cluster PDU with increased capacity or a new cluster PDU connected to the existing UPS. Cluster PDUs typically have a modular design, and we can increase their capacity by adding more circuit breakers [82].

Opportunities and challenges of power oversubscription. Oversubscription in the HPC data center is enabled by its low average utilization [63]. Fig. 3.1(b)

Table 3.1: Capacity oversubscription in Gaia [6].

Oversubscription	**10%**	**15%**	**20%**	**25%**
Extra Capacity (core-hours/month)	144K	216K	288K	360K
Probability of Overload	2.5%	5%	9%	14%
Overload Time (hours/month)	17.8	35.5	68.62	101.3
Overloaded Capacity (core-hour/month)	1.25K	3.9K	8.9K	17.5K
Estimated Maximum Overload Payoff	115×	55×	32×	20×

shows the CDFs of the utilization of four real-world HPC clusters where we see that ∼5% capacity of `Gaia` [6], ∼20% capacity of `Metacentrum` [83], ∼55% of `RICC` [84], and ∼65% of `PIK` [85] are rarely used. Even for the `Gaia` cluster with relatively high utilization, oversubscription can be beneficial.

Table 3.1 shows a quantitative analysis of the benefits of different levels of oversubscription on `Gaia` cluster considering the workload is scaled-up proportional to the extra capacity. Here, the unit of one core-hour indicates the availability of one HPC core for one hour. The extra capacity refers to the additional core-hours we can add to the 2004-core `Gaia` system. We have 144K extra core-hours every month at 10% oversubscription, going up to 360K core-hours at 25% oversubscription. The probability of overload tells us how often the total power consumption goes beyond the infrastructure capacity, and overload time gives us the total time the HPC system stays in an overloaded state each month. To understand the impact of these overload periods, we then calculate the total overloaded capacity, which indicates how many core-hours are spent over the HPC capacity. In other words, overloaded capacity tells us how many total core-hours we need to cut back every month to avoid these overloads. It also reveals the most intriguing observation from this analysis that *we add far more core-hours capacity each month (e.g., 144K added vs. 1.25K cut at 10% oversubscription) than we have to cut back to handle the overloads.* Finally, we show the maximum payoffs we can afford if we pay all the added core-hours as

payment to users for their core-hour cutbacks during overloads. For instance, at 10% oversubscription, we can pay up to 115× of a user's core-hour reduction.

There are compelling benefits of oversubscription in HPC data centers as the HPC manager can add a significant additional capacity to the system. However, as shown in Table 3.1, an oversubscribed HPC data center may occasionally get overloaded. While UPS circuit breakers can handle power overloads for tens of minutes, sustained overloaded operation will affect UPS's longevity [86,87]. More importantly, however, the HPC data center's cooling system cannot withstand overloads as long as UPSs [88]. Consequently, even when adopting a reactive approach, an HPC manager's goal is to mitigate the overloads as soon as possible. In the next section, we formalize the problem of handling these power overloads into an optimization problem, identify the HPC manager's challenges, and propose our market-based solution.

3.3 Handling Power Overloads in HPC

3.3.1 Problem Formulation

Let us consider that at any given time t, there are $M(t)$ jobs running in the HPC system resulting in a total power consumption of $P(t) = \sum_{m=1}^{M(t)} p_m(t, r_m)$, where $p_m(t, r_m)$ is the power consumption attributed to job m running with resource r_m. Note that, instead of traditional approaches of server-wise power modeling (e.g., [66]), here we do job-wise power modeling. The job-wise power model facilitates our market-based design where HPC users (who submit the jobs) play an integral role and are oblivious to how many servers are executing their jobs. Also, such a job-wise power model can be easily extracted by the HPC manager by attributing server power to jobs according to the job's resource share (i.e., number of cores) on that server. In addition, we are considering a unified aggregate power and capacity model for the HPC data

center. However, as described in Section 3.2, large HPC data centers can have multiple parallel power infrastructures, each connected to a dedicated UPS. Nonetheless, our model can be seamlessly extended to these data centers by considering individual infrastructure capacity C_i and aggregate power consumption $P_i(t)$ for the data center's i-th parallel power infrastructure.

With HPC data center power capacity of C, a power overload occurs when $P(t) > C$, and the HPC manager needs to intervene to handle this overload by reducing the power consumption by $P(t) - C$. The power reduction target, $P(t) - C$, needs to be met by reducing the power consumption of the running jobs. Reducing power consumption is accomplished through the reduction of resource allocation. For example, a CPU core slowed down to 90% of its regular frequency can be interpreted as allocation of "0.9 cores". Resource reduction to handle power overload leads to performance degradation of the affected jobs. Hence, an HPC manager's goal is to minimize the overall performance degradation while still achieving the target power reduction. We formalize this as the following optimization problem OPT (OPTimum power overload handling)

$$\mathsf{OPT}: \underset{\delta_m}{\text{minimize}} \sum_{m=1}^{M(t)} \mathcal{L}_m(\delta_m) \tag{3.1}$$

$$\text{subject to} \sum_{m=1}^{M(t)} \mathcal{P}(\delta_m) \geq P(t) - C, \tag{3.2}$$

where δ_m is the resource reduction for job m and $\mathcal{L}_m(\delta_m)$ and $\mathcal{P}(\delta_m)$ are the performance degradation and power reduction, respectively, due to resource reduction δ_m. Here, the optimization objective (3.1) is a scalar measure of overall performance impact due to the overload, and the constraint (3.2) specifies the power reduction requirement to mitigate the power overload.

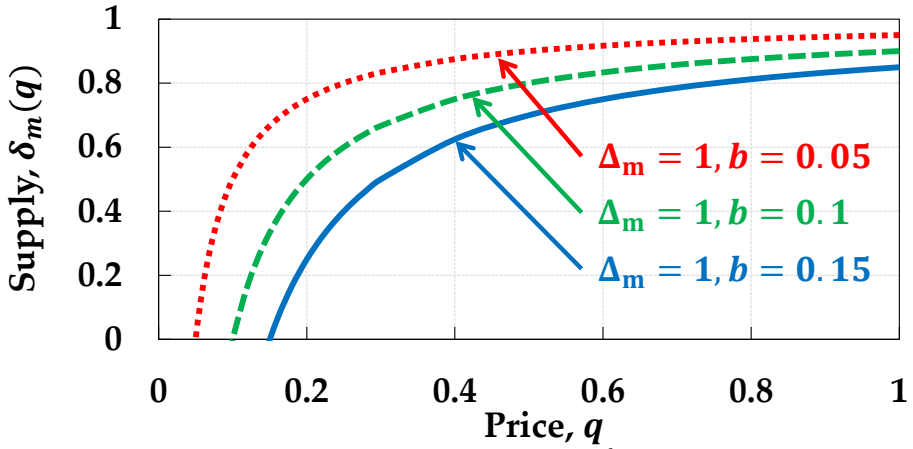


Figure 3.2: MPR's supply function, $\delta_m(q) = \Delta_m - \frac{b_m}{q}$. For a job m, $\delta_m(q)$ is the supply of resource reduction at price q, Δ_m is the maximum resource reduction, and b_m is the bid.

Challenges. A major challenge for an HPC manager in solving OPT is to accurately determine the performance impact $\mathcal{L}_m(\delta_m)$ for each running job when an overload occurs. However, the user who submits the job can best estimate the potential impact of the resource reduction, both in terms of performance degradation and its perceived impact. On the other hand, determining power reduction for resource reduction, $\mathcal{P}(\delta_m)$, is straightforward with established models for any adopted power capping technique [65].

In this work, *we decouple determining the jobs' performance impact due to resource reduction from the HPC manager and engage the HPC users in the power reduction decision.* We enable users to express their affinity for contributing towards meeting the goal of power reduction during overloads.

3.3.2 MPR: Market-Based Power Reduction

We propose a supply function bidding-based market mechanism, MPR, where the HPC users participate in the power overload handling by agreeing to "supply", in exchange for incentives, the required power reduction through resource reduction

of their jobs. At their own discretion, the users determine the level of participation in MPR.

Supply function. In our market design, HPC users use a predetermined form of supply function to indicate how much resource they can reduce at what level of incentive. For a job m, its user provides the parameters Δ_m and b_m to form the following parameterized supply function

$$\delta_m(q) = \left[\Delta_m - \frac{b_m}{q}\right]^+, \tag{3.3}$$

where Δ_m indicates the maximum resource reduction from job m, b_m is the bidding parameter that determines job m's affinity of resource reduction, and q is the incentive/reward per unit resource reduction (e.g., one core of CPU resource reduction for one hour). q can be interpreted as the "unit price" of the market's product which in our case is the resource reduction. $[\cdot]^+$ indicates that $\delta_m(b_m, r)$ is non-negative, meaning that in our market, no job is asked to increase its resource. A similar form of supply function has also been utilized in prior work on electricity markets [89, 90]. The supply function in Eqn. (3.3) indicates how much resource can be reduced for job m if the HPC manager offers an incentive of q for each unit of resource reduction. Fig. 3.2 illustrates MPR's supply function for different bids that results in different amounts of resource reduction for the same q.

Rationale for the choice of our supply function. While the form of our supply function in Eqn. (3.3) is widely used, there are other supply functions, for instance, a linear supply function [91], that can be used for the supply function bidding mechanism. However, our choice is motivated by the fact that Eqn. (3.3) captures the diminishing return on resource reduction, i.e., as we ask for more supply of resource reduction (δ_m), we need to pay more incentive per unit reduction (q). We see similar behavior in HPC applications (Fig. 3.7(b)), where as we increase the

resource reduction, the performance degradation increases super-linearly. In addition, our supply function in Eqn. (3.3) is also backed by theoretical performance guarantees under reasonable assumptions [89, 92, 93].

Power reduction during overload. When an overload occurs, the HPC manager invokes the market to reduce the power consumption of the running jobs. Acting as the facilitator of the market, the HPC manager needs to set the market "clearing price" $q'(t)$, which is used as the basis to determine how much resource reduction each running job needs to supply (i.e., $\delta_m(q'(t))$ for job m) towards meeting the total power reduction goal. Setting the market clearing price $q'(t)$ can be formalized as the following optimization problem MClr (Market Clearing)

$$\mathsf{MClr}: \underset{q}{\text{minimize}} \sum_{m=1}^{M} q \cdot \delta_m(q) \tag{3.4}$$

$$\text{subject to} \sum_{m=1}^{M} \mathcal{P}(\delta_m(q)) \geq P(t) - C. \tag{3.5}$$

MClr's objective in (3.4) is to minimize the cost of handling the overload by minimizing the total incentive payoff to the running jobs. The key distinction between OPT and MClr is that the *HPC manager in MClr no longer needs to determine the performance impact $\mathcal{L}_m(\delta_m)$ to set the resource reductions of the active jobs.*

Soliciting bids and exercising the market. As part of the implementation of MPR, we introduce two approaches towards how the bids (i.e., Δ_m and b_m) are collected from the users and how the market clearing price $q'(t)$ is set.

A static market: In the first approach, the bidding parameters Δ_m and b_m are supplied to the HPC manager during job submission. The HPC manager invokes a market instance when there is an overload and uses the already-received bids of all active jobs. The HPC manager sets the market clearing price by plugging the bids

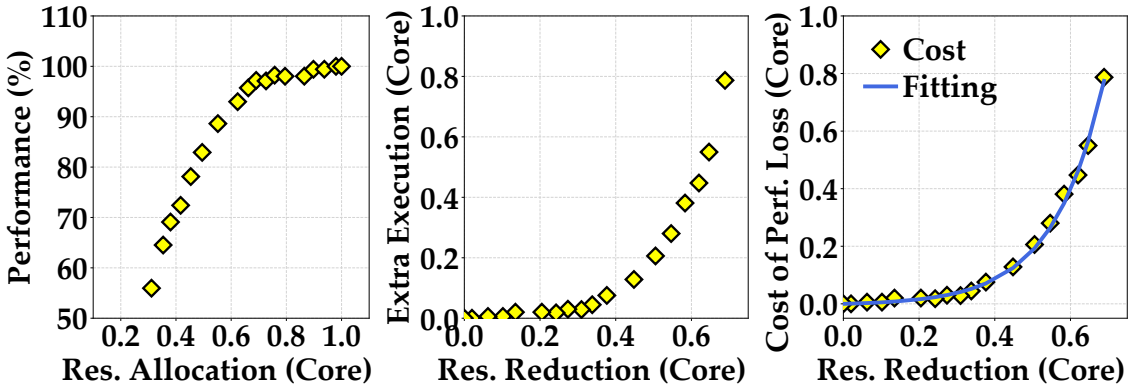
into MClr. Since the bids remain unchanged during the market execution, we call this approach MPR-STAT (MPR with static bidding).

An interactive market: In the second approach, the bidding parameters Δ_m and b_m for job m are iteratively updated by the users after the HPC manager invokes the market following an overload. First, the HPC manager declares an initial clearing price $q'_0(t)$. Upon receiving the clearing price, users with jobs running in the system send their bids. The HPC manager plugs the bids into MClr and determines the new clearing price. The HPC manager sends the updated clearing price to the users, who in turn send back their updated bids. This back-and-forth communication continues until the clearing price converges to a stable value. The convergence of clearing price (i.e., Nash equilibrium) is guaranteed if the users take the price set by the HPC manager in each iteration and behave rationally by maximizing their net market gain (Eqn. (3.7)) [89, 92, 93]. Since the clearing price is determined based on interactions between the HPC manager and the users, we call this approach MPR-INT (interactive MPR). We defer the qualitative comparison of these two approaches to Section 3.3.4.

3.3.3 User Bidding in MPR

A key step for enabling MPR's performance-oblivious power reduction by the HPC manager is collecting bids from the users/jobs. Our market mechanism is designed to proxy the performance impact $\mathcal{L}_m(\delta_m)$ using the supply function in Eqn. (3.3). Hence, the bidding parameters need to be decided based on the performance impact from the job's resource (and hence power) reduction. Here, we describe how an HPC user devices its bids based on its performance impact.

Cost of performance loss. We define $\mathcal{C}_m(\delta_m)$ as the user-perceived cost of performance degradation from δ_m resource reduction. The notion of cost enables HPC users to integrate their own relative importance of different jobs in their bidding.



(a) Performance (b) Extra execution (c) Performance cost

Figure 3.3: (a) Performance at different levels of resource allocation. (b) Impact of resource reduction. $ExtraExecution = \frac{100 - Performance}{Performance}$. (c) Cost impact of resource reduction. $Cost = \alpha \cdot ExtraExecution$ with $\alpha = 1$.

While the HPC users can decide how they want to quantify the cost at their own discretion, in this work, we consider the additional work (i.e., increase in execution time or "extra execution") needed to finish the job as the cost of performance loss. Considering $\mathcal{L}(x)$ to be the job runtime with a core reduction of x, we can generalize the cost impact of resource reduction as

$$\mathcal{C}_m(\delta_m) = \alpha_m(\mathcal{L}_m(\delta_m) - \mathcal{L}_m(0)), \tag{3.6}$$

where $\alpha \geq 1$ is a coefficient that a user can tune to reflect its perceived cost of the additional execution. $\alpha = 1$ indicates that the HPC user does not add any surcharge on the actual performance impact. Alternative to this linear cost and popularly used in system research for performance cost is a "quadratic cost" function, i.e., $\mathcal{C}_m(\delta_m) = \alpha_m \cdot (\mathcal{L}_m(\delta_m) - \mathcal{L}_m(0))^2$, where the cost of performance grows quadratically with increasing performance loss.

As a concrete example, Fig. 3.3(a) shows the performance of `XSBench` application [94] with different levels of resource allocation. Here, a core allocation of "1" indicates the core is running at 100% speed. Next, in Fig. 3.3(b), we show the extra

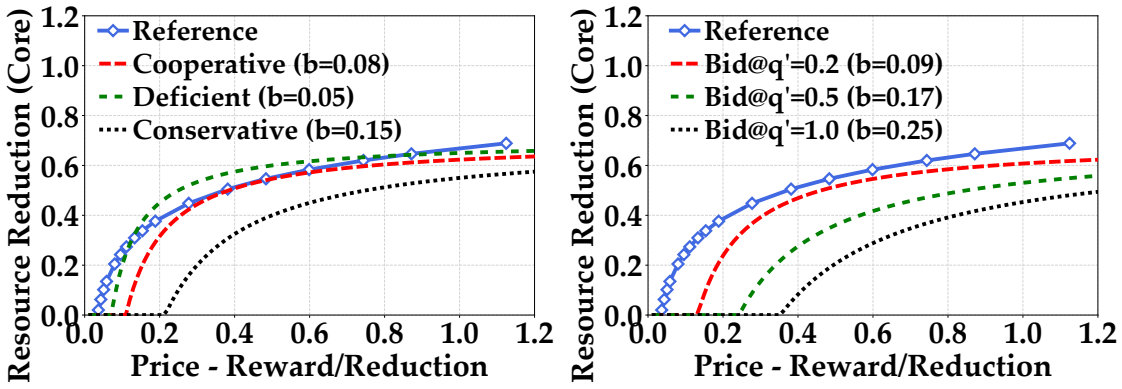
execution needed when the core allocation is reduced. In Fig. 3.3(c), we show the cost associated with different levels of resource reduction using Eqn. (3.6) and $\alpha = 1$.

Devising the reference cost for bidding. Our supply function is based on the unit price of supply, i.e., payment for per unit resource reduction, and hence to utilize the performance data for bidding we convert the *cost of resource reduction* into *cost per unit resource reduction* as $\mathcal{C}'_m(\delta_m) = \mathcal{C}_m(\delta_m)/\delta_m$. Using this equation, the reference lines in Figs. 3.4(a) and 3.4(b) are derived from the cost of performance shown in Fig. 3.3(c). For any amount of resource reduction (in the y-axis), from the reference lines in Fig. 3.4, we can find a user's actual cost of per unit resource reduction (in the x-axis). In our context of bidding for supply, we can interpret this cost reference curve as the upper limit on resource reduction without a loss (i.e., the cost is greater than the incentive).

Bidding strategy. An HPC user's net gain from market participation is the payment it gets for resource reduction minus the corresponding performance degradation cost it incurs. Hence, with the market clearing price q', we can write the m-th user's net gain as

$$\mathcal{G}_m = \overbrace{q' \cdot \delta_m(q')}^{\text{Market payoff}} - \overbrace{\mathcal{C}_m(\delta_m(q'))}^{\text{Cost of resource reduction}} \tag{3.7}$$

The bidding strategy for a user depends on what kind of market is implemented. For MPR-STAT market, the users need to decide their bidding parameters, b_m, without any knowledge of the market clearing price, q'. Note that the bidding parameter Δ_m depends on the HPC application's behavior, and the user does not tune this parameter during bidding. For instance, in `XSBench` we have $\Delta_m = 0.7$. In MPR-STAT, a user cannot maximize its net gain $\mathcal{G}_m$. Nonetheless, we propose a cooperative bidding strategy where the bids are devised to achieve a non-negative net gain over the entire price range. More specifically, in this bidding strategy, a user sets its



(a) Bidding in MPR-STAT (b) Bidding in MPR-INT

Figure 3.4: User bidding strategy for market participation.

bidding parameters to keep its bidding curve $\delta_m(q)$ always below its reference cost with the highest supply of resource reduction. We deem this a "cooperative" bidding strategy as the HPC users offer their best resource reduction for handling the power emergency. Fig. 3.4(a) illustrates the cooperative bid for `XSBench` application. To better understand this bidding strategy for MPR-STAT, we also show a "conservative" bid, where the HPC user is less willing to reduce power and bids for lower resource reduction than its reference. We also show a "deficient" bid, where the HPC user's bid may result in a negative gain for certain clearing prices (for $0.2 \leq q' \leq 0.8$ in Fig. 3.4(a)).

On the other hand, for MPR-INT market, the clearing price q' is iteratively updated. During each iteration, a user can plug in the clearing price into Eqn. 3.7, and can find the value of b_m that maximizes its net gain $\mathcal{G}_m$. In this bidding strategy, the user can maximize its market incentive. We illustrate this bidding strategy in Fig. 3.4(b) for three different clearing prices.

The bidding strategy presented here relies on the estimation of performance impact due to resource reduction. This performance estimation for HPC jobs, how-

ever, is non-trivial and introduces additional hurdles on the user's part. We offer a qualitative discussion of such challenges of MPR in Section 3.3.6.

3.3.4 Properties of Our Market Design

MPR-STAT vs MPR-INT. The fundamental difference between MPR-STAT and MPR-INT is the market agility, i.e., how quickly the HPC manager can determine the clearing price. While exercising the market in MPR-STAT, the HPC manager has all the information it needs (i.e., the bids and the reduction goal) to handle the overload and, therefore, can very quickly determine how much resource to cut back from each job. Meanwhile, in MPR-INT, multiple rounds of back-and-forth communication between the HPC manager and the users are needed to reach a consensus on the clearing price.

MPR-INT, however, offers theoretical guarantees on the optimality of the overall performance cost [89, 92], and performs as well as OPT. In MPR-STAT, on the other hand, the users devise their bids without any knowledge of the clearing price. Hence, unlike MPR-INT, they cannot guarantee that the power reduction is achieved with the minimum performance impact on the running jobs. MPR-STAT can still capture the relative performance impact of different users' jobs and consistently achieve better cost performance than performance-oblivious power overload handling strategies.

MPR-STAT, due to its agility, is suitable where fast reaction time to power overload is warranted. Meanwhile, MPR-INT can offer the best cost performance, where the HPC system can sustain the power overload long enough for the market to clear. Here, to ensure the safe handling of power emergencies, the HPC manager can set a fixed timeout (e.g., 30 seconds) for MPR-INT's iterations and take the last price as the clearing price. MPR-INT also requires autonomous software agents who send bids without manual user/human involvement. Such bidding agent implemen-

tation is relatively straightforward as they require lightweight computation to find the optimum bid for Eqn. (3.7). Also, to better accommodate MPR-INT, the HPC manager can invoke the market early by predicting power overloads and estimating the power/resource reduction goals.

Scalability. The HPC manager uses MPR's market when a power emergency is detected that needs to be mitigated by power reduction. To set the market clearing price and determine job-wise resource reduction in MPR-STAT, the HPC manager needs to solve MCIr only once using the bids of the active jobs. Moreover, since MCIr has only one optimization variable q and the objective function is monotonically increasing in q, it can be solved by finding the minimum, $q' = \min_q\{q|\sum_{m=1}^{M(t)}\mathcal{P}(\delta_m(q)) = P(t) - C\}$ using a bi-section search. MPR-STAT can scale very well with a growing number of active jobs. In our evaluation, we find that MPR-STAT can find the clearing price in less than a second for even 30,000 active jobs (Fig. 3.10(a)). In contrast, OPT has M optimization variables (i.e., number of jobs running in the HPC) with exponentially growing problem size (e.g., 40+ minutes to solve a problem of 30,000 jobs).

MPR-INT, on the other hand, is by nature slower as it needs iterative communication between the HPC manager and the users. However, the time required for MPR-INT mainly comes from the communication overhead as MPR-INT also solves the lightweight MCIr only once every communication round. Meanwhile, to determine their bids, the users need to solve even a simpler problem of maximizing $\mathcal{G}_m$ in Eqn. 3.7 without any constraints. More importantly, the users devise their bids by themselves in a distributed fashion. Hence, the growing number of users only adds more parallel work while the HPC manager's task (to solve MCIr) in every communication round grows similar to MPR-STAT. The main scalability concern for MPR-INT is how many communication rounds are needed for the clearing price to converge.

Because of our predefined form of the supply function, the convergence is guaranteed with the user's cost monotonically increasing with resource reduction [89, 92, 93]. In our evaluation, we find that the number of iterations needed for clearing the market for MPR-INT remains almost unchanged even when we increase the jobs from 10 to 30,000 (Fig. 3.10(b)).

3.3.5 Implementation of MPR

Detecting power emergency. MPR uses the HPC cluster's real-time power monitoring to identify when the power consumption exceeds the capacity and there is an overload. MPR then determines the amount of power that needs to be reduced to return the power at or below the capacity. To avoid declaring a power emergency for transient power spikes, the HPC manager may set a minimum duration of overload (e.g., 10 seconds).

Setting the market clearing price. The solicitation of bids and market clearing is done following the adopted version of MPR. The HPC manager plugs in the market clearing price q' and every user's bids (in case of MPR-INT, the bids in the final iteration) in the supply function (Eqn. 3.3) to determine the user's corresponding resource reduction.

Executing resource/power reduction. The HPC manager reduces each job's resource allocation utilizing existing techniques such as slowing down the processor using DVFS [95, 96]. During a power emergency, MPR also temporarily halts starting any new HPC job execution.

Resuming normal operation. MPR resumes normal HPC operation when it determines that lifting the power reduction will no longer violate the capacity. Hence, MPR lifts the power emergency when the power consumption falls below the capacity by at least the amount of power reduction. Here, the HPC manager can add

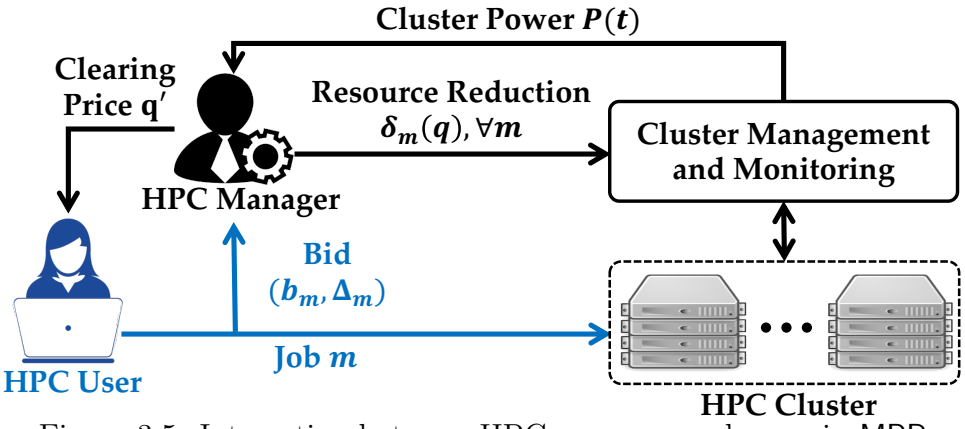


Figure 3.5: Interaction between HPC manager and users in MPR.

a cool-down timer (e.g., 60 seconds) to avoid lifting a power emergency followed by a momentary power dip only to declare an emergency again. The cool-down timer also ensures a minimum time frame for payout to the HPC users participating in the market.

Resource control mechanism. While MPR can be implemented with various resource reduction techniques, such as power capping and node/core scaling, we advocate using DVFS on CPU/GPU cores as it is ubiquitously available with rapid and scalable execution. Moreover, DVFS has a more predictable impact on job execution time as it only slows down the execution (Fig. 3.16(b)). Hence, confining MPR's resource control knobs to DVFS (or a similar technique) also alleviates the hurdles of performance modeling.

3.3.6 Challenges in MPR

Performance prediction for bidding. In MPR, the HPC users devise their bids based on the estimation of the performance impact of resource reduction. Hence, an integral part of MPR is performance prediction which remains challenging [65]. This paper mainly focuses on the user-in-the-loop handling of HPC oversubscription

and treats HPC performance modeling as an orthogonal task. Nevertheless, we would like to emphasize that *MPR is not dependent upon rigorous and extensive performance modeling.* In MPR, HPC users express their performance impact through a predefined form of supply function (Eqn. (3.3)), which is already an approximation of actual performance impact (Fig. 3.4), allowing margin-of-error in performance prediction. Moreover, HPC users, at their discretion, can intentionally raise their bids b_m (e.g., conservative bid in Fig. 3.4(a)) to add even more room for estimation error to account for uncertainties of the operating environment, such as interference between different jobs. We evaluate the impact of model error on MPR in Section 3.5.4.

Market participation. Naturally, MPR's user-in-the-loop design requires HPC users' willingness to actively participate in the market to supply the resource reductions necessary to handle the overloads. While the market participation requires additional user efforts (i.e., bidding), as opposed to prior studies on managing oversubscribed HPC systems, MPR compensates users for the inevitable performance impact of oversubscription. Hence, we believe there is a strong incentive for user participation in MPR. We study the impact of user participation on MPR in Section 3.5.4. Meanwhile, to encourage user participation in MPR and ease up their bidding process, the HPC manager can take a more active role by accommodating discounted job execution to assist performance modeling and hosting users' bidding agents.

Market collusion. In theory, MPR's design is susceptible to market collusion where multiple HPC users coordinate and artificially inflate the reward/price for their resource reduction. However, market collusion requires coordination among many users to have enough market power to influence the clearing price. Hence, we believe the efforts outweigh the incentives for market collusion in the HPC system.

Malicious users. Unlike self-serving market colluding users, malicious users want to steal private/secret data and harmfully affect the HPC system. MPR does

not add any new attack surface regarding data security. However, allowing users to take an active role in HPC overload handling, MPR creates a new vulnerability. By tracking when the market is invoked, a malicious user would know if the HPC system is experiencing a power overload. The attacker can utilize this information and launch "power attacks" [75, 97], where the attacker triggers power-intensive stage(s) of their active jobs to intensify the overload by creating power spikes. However, such power attacks cannot easily reach dangerous levels (e.g., HPC system/data center shutdown [75]) as the HPC manager actively manages jobs' resources (and hence power capping) and can quickly thwart unwanted power spikes by "directly" reducing the power of all users/jobs bypassing MPR.

Impact on the total cost of ownership (TCO). MPR affects the HPC's TCO in two ways - increase in HPC utilization and reward payoff to HPC users. The infrastructure utilization will increase due to oversubscription affecting the cost of electricity in TCO. Meanwhile, MPR rewards the HPC user for their market participation. The reward payoff will be a MPR specific addition to existing TCO calculations.

3.4 Evaluation Methodology

3.4.1 Simulation Settings for HPC

Workload traces. We use real-world workload traces for our evaluation. For our core results, we use the workload traces from the `Gaia` cluster at the University of Luxemburg [6]. The `Gaia` trace contains 51,987 jobs spanning a three-month period from May 2014 to August 2014. This trace has been widely used in the literature and referenced in a number of studies throughout the years to generate useful workloads

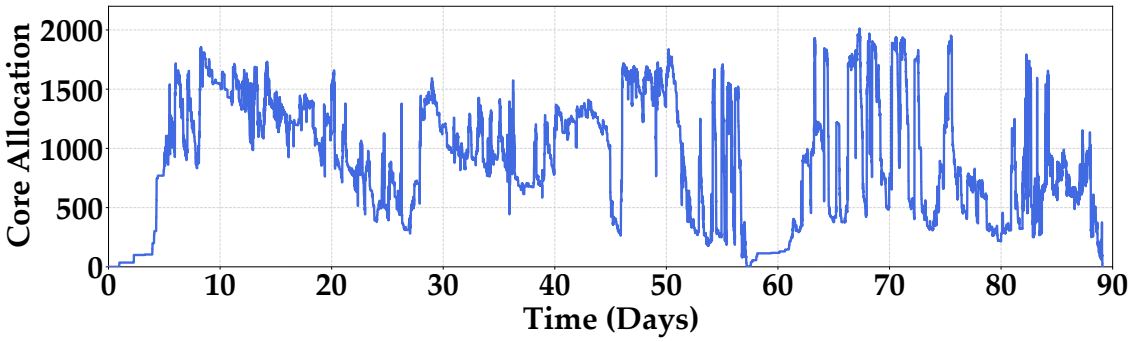


Figure 3.6: Core allocation of the `Gaia` cluster [6].

(e.g., [98–100]). Fig. 3.6 shows the core allocation of the `Gaia` cluster with a peak core allocation of 2012.

Power consumption. We convert the core allocations to power consumption using the widely used power model, $\text{Power} = \text{Power}_{static} + \text{Utilization} \cdot \text{Power}_{dynamic}$ [41], considering each core has a dynamic power of 125W and static power of 25W, resulting in peak power of 301.8KW for `Gaia`. Utilization is calculated as core allocation divided by total available cores. Here, we consider that the power consumption of different components, such as the uncore, DRAM, and storage power, are incorporated in Power_{static} and $\text{Power}_{dynamic}$. The per-core dynamic and static powers are only estimations. Our simulation and analysis hold for other power models as well.

Job simulation. We use Matlab to simulate the HPC job execution by dividing the entire simulation period into one-minute time slots. We get the start time, core allocation, and runtime for each job from the workload traces. We keep a list of active jobs with remaining runtimes. The list is updated at each time slot by adding new jobs (if the system is not overloaded) and discarding completed jobs. For every active job, we also track their core speeds. At the end of a time slot, the remaining runtimes of all active jobs are updated based on their corresponding core speeds. To determine how much work has been done for a given core speed, we use the performance models

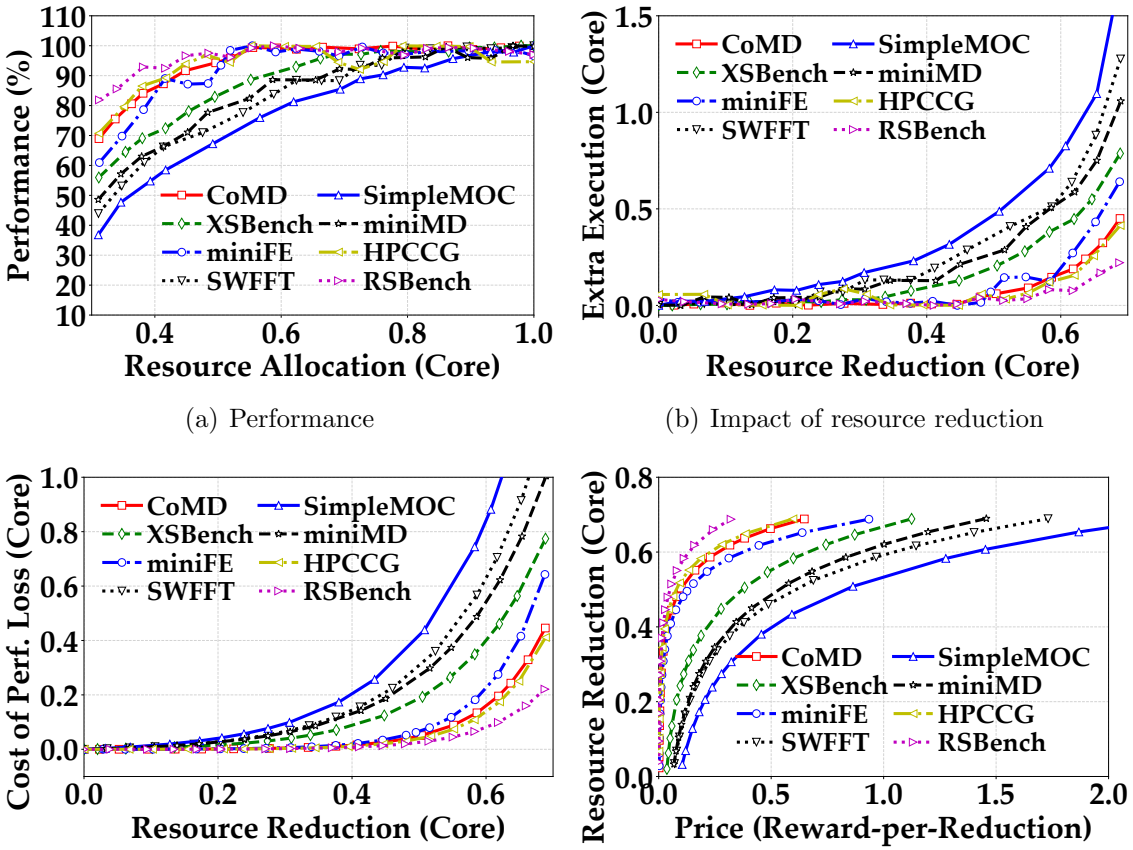
described in Section 3.4.2. We use the power model (Section 3.4.1) to convert the core allocation to HPC power consumption and determine if there is an overload.

Oversubscription levels and power emergencies. For our evaluation, we consider four levels of oversubscription thresholds at 5%, 10%, 15%, and 20%. With x% oversubscription, overloading occurs if the power demand exceeds $\frac{100}{100+x}$ of its peak power consumption (e.g., 301.8 kW for `Gaia`). To avoid immediate relapse to another overload, we set the power reduction target using an additional 1% buffer as $\Delta P = P(t) - 0.99 \cdot C$. We use a 10-minute cool-down period before we consider resuming normal operation by giving back the capped resources to the active jobs. After the 10-minute cool-down, we resume normal operation if $0.99 \cdot C - P(t) \geq \Delta P$.

Benchmark algorithms. We evaluate MPR against two benchmark algorithms - OPT and EQL. OPT finds the optimum resource allocation by solving the non-linear optimization problem that minimizes the total cost of performance loss of all active jobs. OPT acts as the performance upper limit for handling the overloads. EQL, on the other hand, is oblivious to the performance impact of resource change and equally slows down all cores in the system to reduce power.

3.4.2 Simulation Settings for Users

Performance models. We utilize existing literature to model the performance impact of power capping [65]. We collect the power vs. performance measurements for eight applications that include `CoMD`- a molecular dynamics simulation application that studies dynamic properties of various materials, `XSBench`- an application that stresses system through memory capacity, `miniFE`- a proxy application for unstructured finite element solver, `SWFFT`- an application for cosmology and astrophysics, `SimpleMOC`- a three-dimensional reactor simulation application), `miniMD`- a parallel molecular dynamics code from Mantevo mini-application suite, `HPCCG`- a conjugate



(a) Performance

(b) Impact of resource reduction

(c) Cost of resource reduction

(d) Cost references for bidding

Figure 3.7: Performance models, cost models, and bidding references for benchmark applications.

gradient proxy application, and `RSBench`- a transport application for Monte Carlo neutron transport.

We convert the power capping values from [65] to core allocations by normalizing no power capping (290W) to the core allocation of "1". Fig. 3.7(a) shows the performance changes for resource allocation changes of our benchmark applications. We see that different applications have different impacts on their performances when their resource allocation is altered. We see that some applications, such as `SimpleMOC`, `SWFFT`, `miniMD`, and `XSBench`, are more sensitive to changes in resource allocation than

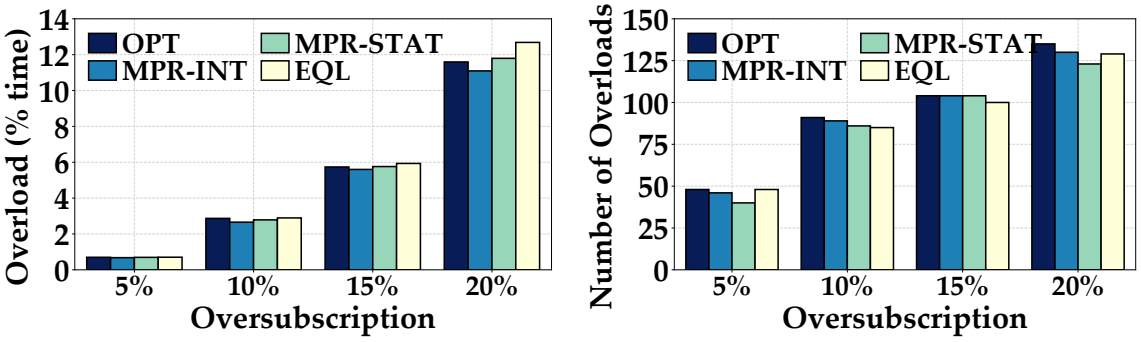
others. Here, we do not consider the impact of inter-core/node communication, which is typically much less compared to the impact of power capping [66, 67].

Next, in Fig. 3.7(b), we show the impact of the performance change in terms of "extra execution" that these applications need to finish the same job due to a change in their performance. In this figure, both the resource reduction and extra execution have the same unit of time - if we consider resource reduction for one hour, then we need the corresponding amount of extra execution cores for one hour as well.

Cost models. We consider the extra execution as the added "cost" for the application when their resource is reduced. Then we use the cost model (Section 3.3.3) to derive the cost. To model the cost of performance loss due to resource reduction, we use a logarithmic curve fitting on our costs calculated based on results from Fig. 3.7(b). Our logarithmic fitting is $cost = a \log(b \cdot x) - a$, where x is the resource reduction, and a and b are model parameters. Fig. 3.7(c) shows the cost of resource reduction based on our logarithmic model.

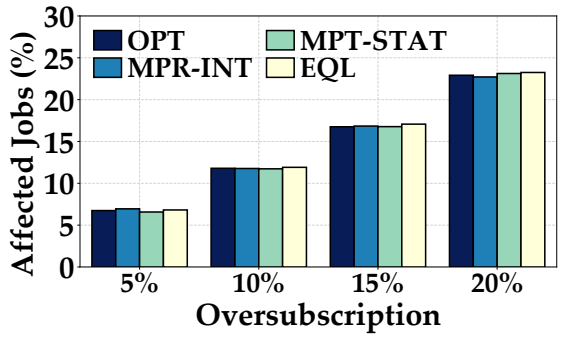
Bidding references. Using the cost calculated in Fig. 3.7(c), we derive the bidding references for our applications and show them in Fig. 3.7(d). Here, the price of the bidding references is the cost of unit resource reduction. Since we use cores as both the unit of cost and the unit of resource reduction, our price becomes unitless.

Application profiles. We devise eight application profiles using our performance models, cost models, and bidding references. We uniformly randomly assign an application profile to each HPC job we simulate. The application profile of a job determines its performance impact due to core reduction and its bids in market participation. We also scale up our per-core model with the core allocations of the respective HPC job.

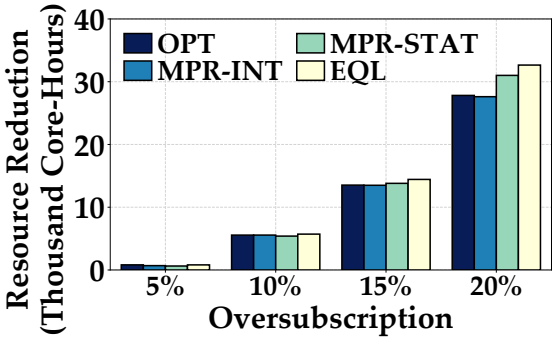


(a) Capacity overloads

(b) Overload events



(c) Affected jobs



(d) Resource reduction

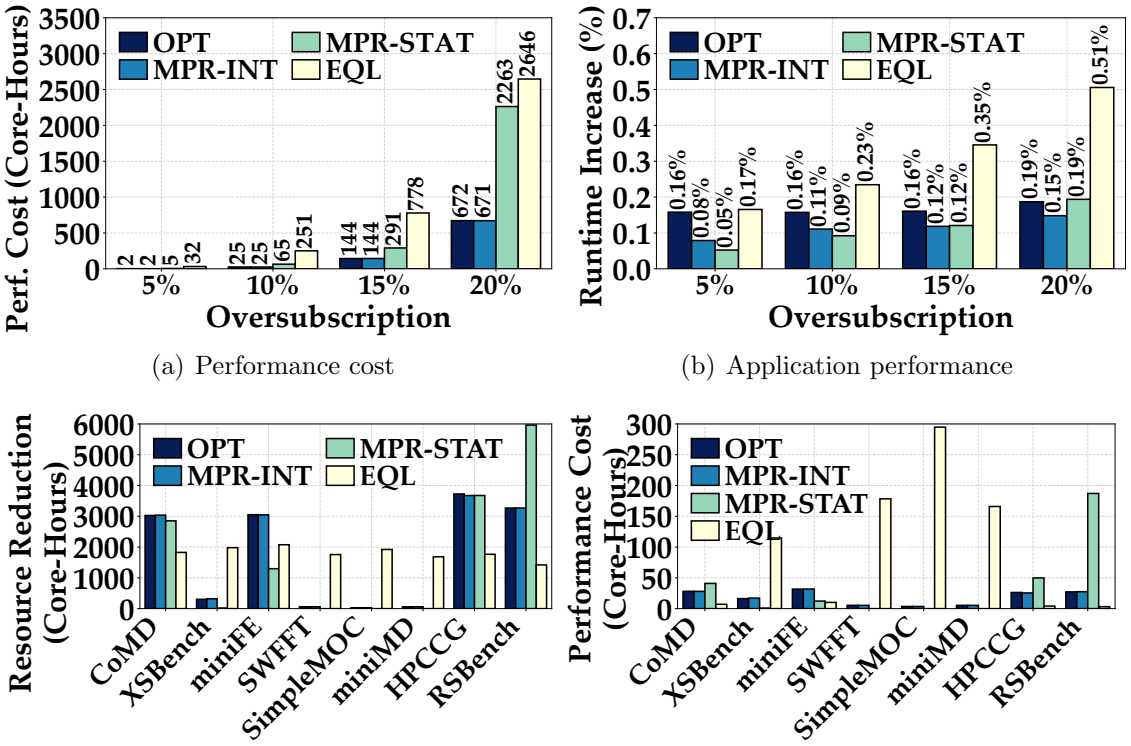
Figure 3.8: Impact of oversubscription on the `Gaia` system and the HPC jobs. `Gaia` has a capacity of ∼4.3million core hours over our simulation period.

3.5 Evaluation Results

3.5.1 Impact of Oversubscription

Capacity overloads. Figs. 3.8(a) and 3.8(b) show how often the system stays in the overloaded state as we increase the oversubscription level. We see that at 5% oversubscription, the system stays in the overloaded state less than 1% of the time. However, as we increase the oversubscription level, the overload percentage grows super linearly, indicating a diminishing return of oversubscription. All the algorithms perform comparably to each other in terms of causing overloads.

Impact on jobs. In Fig. 3.8(c), we show the percentage of jobs affected by the overloads. We consider a job has been affected by overload if an overload event occurs



(a) Performance cost

(b) Application performance

(c) Profile wise resource reduction at 15% oversubscription

(d) Profile wise cost at 15% oversubscription

Figure 3.9: Comparison of benchmarks over 90-days simulation using `Gaia` trace.

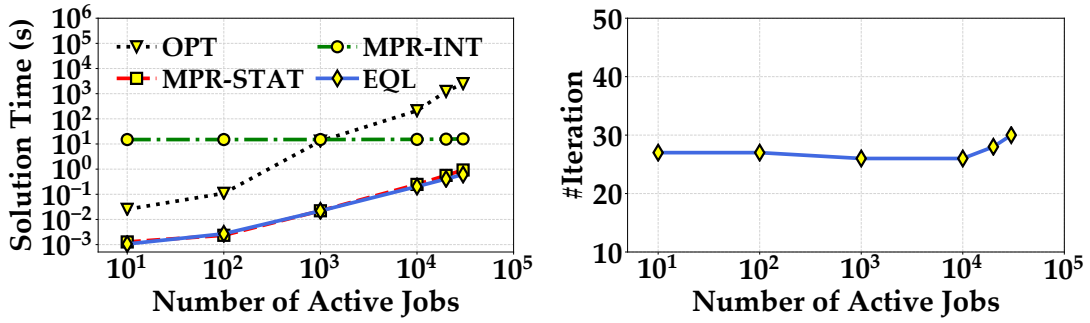
when the job was in the active state, regardless of whether the job's resource was cut back or not to handle the overload. We observe that increasing oversubscription affects more jobs. Note that, despite a greater percentage of jobs being affected during the oversubscription, the performance impact on the jobs is not significant (Fig. 3.9(b)).

Resource reduction. Fig. 3.8(d) shows the total resource reduction for different algorithms. Since the required resource reduction is dictated by the overloads, all algorithms result in similar amounts of resource reduction.

3.5.2 Benchmark Comparison

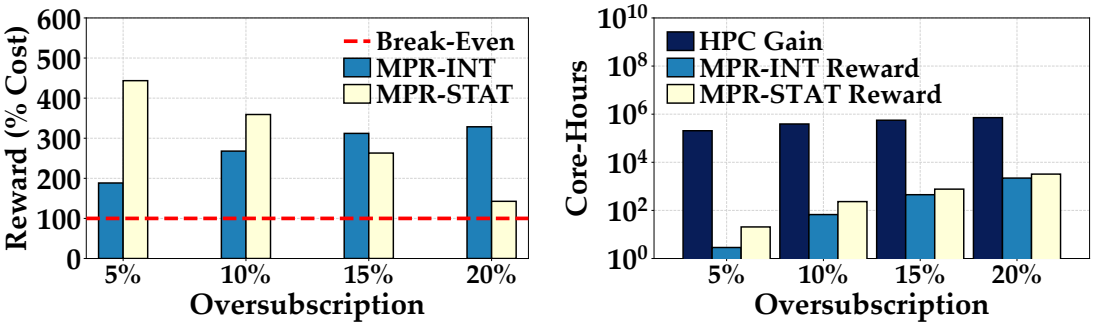
Performance cost. Fig. 3.9(a) shows the total cost of performance loss due to resource reduction to handle overloads at different oversubscription levels. The unit of cost is "core-hours", indicating how much extra computing is needed to handle the slowdown caused by the overloads. Naturally, the cost increases as we increase oversubscription. Here, we see significant differences in algorithm performances, where EQL suffers from significantly higher performance cost, while MPR-INT achieves cost performance at nearly the same level as OPT. MPR-STAT, however, incurs notably more cost than OPT. EQL's higher cost is due to its performance-oblivious nature. As shown in Fig. 3.9(c), EQL reduces as much resource from sensitive applications (e.g., `SimpleMOC`) as other less-sensitive applications (e.g., `RSBench`) and incurs high performance cost for the sensitive applications (Fig. 3.9(d)). Both OPT and MPR-INT achieve a good balance in spreading the resource reduction among the applications, reducing more resources from less-sensitive applications, and vice versa. MPR-STAT, on the other hand, reduces much more resources from the less-sensitive applications (e.g., `RSBench`) and does not reduce any resources from sensitive applications (e.g., `SWFFT`). This is because MPR-STAT uses static bidding from users where users have to bid considering a wide range of prices and end up soliciting unnecessary conservative bids for lower price ranges.

Application performance. Fig. 3.9(b) shows the average increase in runtime (compared to the no oversubscription case) of only the jobs affected by the overload. We see that there is less than 1% increase in average runtime for any algorithm. The performance impact is very small as the overload periods are a small fraction of typical job's total execution time. Moreover, most overloads require less than 20% resource



(a) Solution time (b) Iteration number for MPR-INT

Figure 3.10: Observed solution time of benchmarks.



(a) Reward and cost (b) Reward

Figure 3.11: User rewards and HPC system's gain from MPR.

reduction, for which, only sensitive jobs suffer a discernible performance degradation (Fig. 3.7(b)).

We see that EQL performs worse than other algorithms, and causes a greater increase in the execution time. We also see that MPR-STAT performs better than OPT and MPR-INT on many occasions. This is because MPR-STAT heavily relies on the less-sensitive jobs for reaching the target resource reduction. Nonetheless, the takeaway is that overload handling does not significantly affect performance.

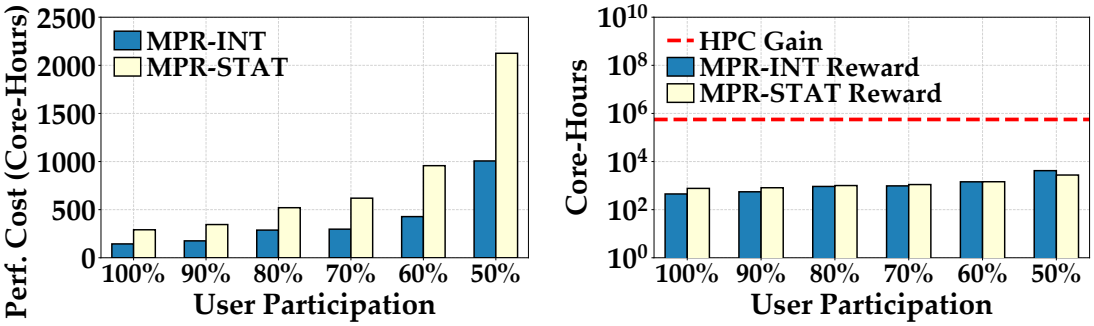
Scalability. Fig. 3.10 presents the solution times (i.e., time to determine the resource reductions) of OPT, MPR-INT, MPR-STAT, and EQL for varying numbers of active jobs on an iMac computer with Intel Core i9 processor and 128GB of memory. The solution time increases for all the benchmarks as the number of active jobs

increases (Fig. 3.10(a)). Note that, MPR-STAT demonstrates very good performance compared to OPT for varying numbers of active jobs. The EQL benchmark achieves the same level of performance as that of MPR-STAT, however, incurs a significantly higher cost of performance loss compared to MPR-STAT (Fig. 3.9(a)). Note here that, EQL's increasing solution time is due to the "bookkeeping" (e.g., log each job's new CPU allocation) associated with an increasing number of active jobs. MPR-INT, on the other hand, needs additional communication time for each round of iterative bidding. We present MPR-INT's solution time considering that each communication round adds 500 milliseconds. Nevertheless, we see that MPR-INT can find the resource allocation in less than 30 seconds even with 30,000 active jobs. Fig. 3.10(b) shows the number of iterations needed for clearing the market for MPR-INT algorithm. The iteration number remains almost unchanged with the number of active jobs.

3.5.3 Market Performance

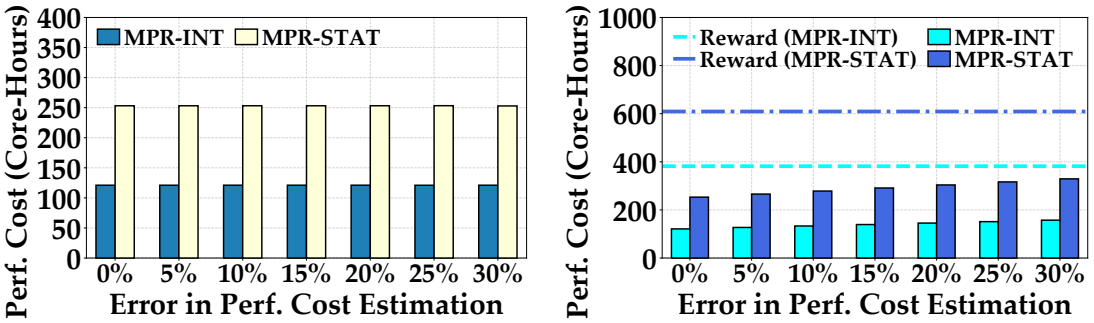
User's reward. Fig. 3.11(a) shows the reward users receive for their participation. The reward is calculated as a percentage of the cost incurred due to performance degradation from resource reduction. As evident from the figure, users always receive more rewards (>100%) than their cost of performance loss. Therefore, users will always enjoy a net benefit for participating in MPR's market for overload handling.

HPC system's benefit. Fig. 3.11(b) shows the gain of the HPC manager due to oversubscription and the reward earned by the HPC users. We see that the HPC manager gains orders of magnitude more core-hours than she had to pay to the users as the incentive/reward. Also, note that while the HPC gain increases with oversubscription, the user incentive grows at a higher rate. It indicates that it is not beneficial for the HPC manager to oversubscribe the system beyond a certain



(a) Performance cost (b) Reward

Figure 3.12: Impact of user participation on MPR. (a) Impact on performance cost. (b) Impact on reward payoff.



(a) Random estimation error (b) Performance cost underestimation

Figure 3.13: (a) Random estimation errors do not affect MPR. (b) Even when users underestimate cost, they retain a net gain - more reward than cost.

level. Nevertheless, *these results highlight the economic motivation for both the HPC manager and HPC users to adopt MPR.*

3.5.4 Sensitivity Study

User participation. If fewer users participate in MPR, for a given power reduction target, each job needs to supply more resource reduction incurring more performance costs, while the HPC manager will need to pay more rewards. Fig. 3.12 shows the impact of user participation on the overall performance cost and reward for 15% oversubscription. We see increasing performance cost with decreasing user

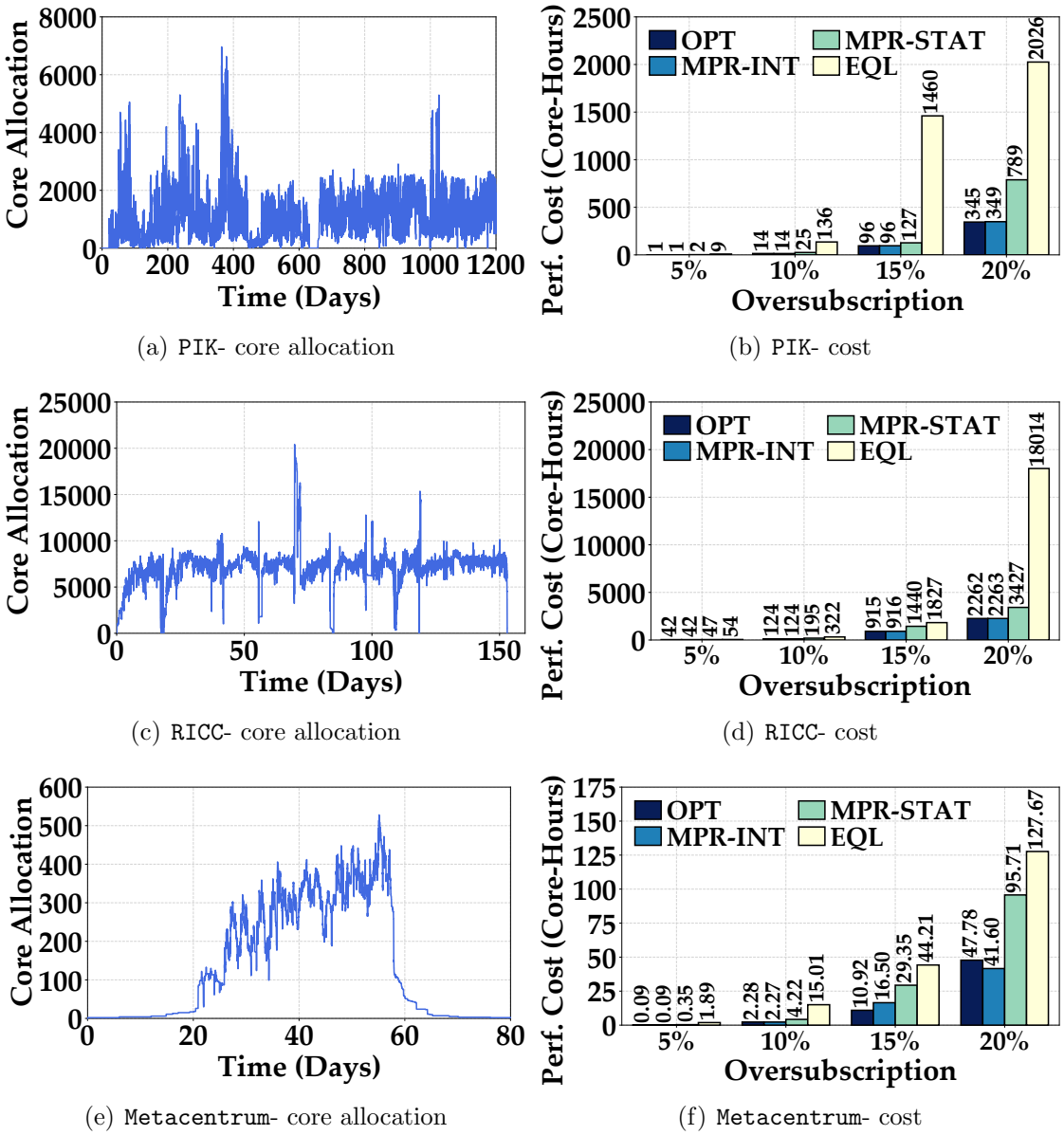
participation in Fig. 3.12(a). As shown in Fig. 3.12(b), the increase in users' performance cost is offset by an increase in reward payment from the HPC manager. However, note that, even at 50% user participation, HPC gain remains two orders of magnitude higher than the reward payment.

Error in the performance cost model. To understand the impact of errors in the performance cost model, we study the actual performance cost when there are random estimation errors of up to 30% in Fig. 3.13(a). We see that random estimation errors do not affect the overall performance cost. We then study with a pessimistic setting where users underestimate their true performance cost and show the results in Fig. 3.13(b). We see that, while the cost increases with underestimation, even at 30% underestimation, the reward is two times the cost for MPR-INT and MPR-STAT.

3.5.5 Evaluation Under Different Settings

Different workload traces. We collect three other workloads (`PIK`, `RICC`, and `Metacentrum`) for this study from [5]. The traces are representative of different workload characteristics. The `PIK` trace is from a medium-scale HPC cluster over a longer time duration (Fig. 3.14(a), `RICC` trace is from a large-scale HPC cluster (Fig. 3.14(c)), and `Metacentrum` trace is from a small-scale HPC cluster (Fig. 3.14(e)). The `PIK` trace contains 742,964 jobs spanning a three-year period from April 2009 to July 2012. The `RICC` trace contains 447,794 jobs over a 5-month period from May 2010 to September 2010. The `Metacentrum` trace contains 103,656 jobs, which are collected from January 2009 to May 2009. `PIK` trace, `RICC` trace, and `Metacentrum` traces have peak CPU allocation of 6,963 cores, 20,4156 cores, and 528 cores, respectively.

Figs. 3.14(b), 3.14(d), and 3.14(f) show the cost of performance loss for the workload `PIK`, `RICC`, and `Metacentrum`, respectively. The performance cost increases with the increase in oversubscription. MPR-INT achieves cost performance almost the



(a) PIK- core allocation

(b) PIK- cost

(c) RICC- core allocation

(d) RICC- cost

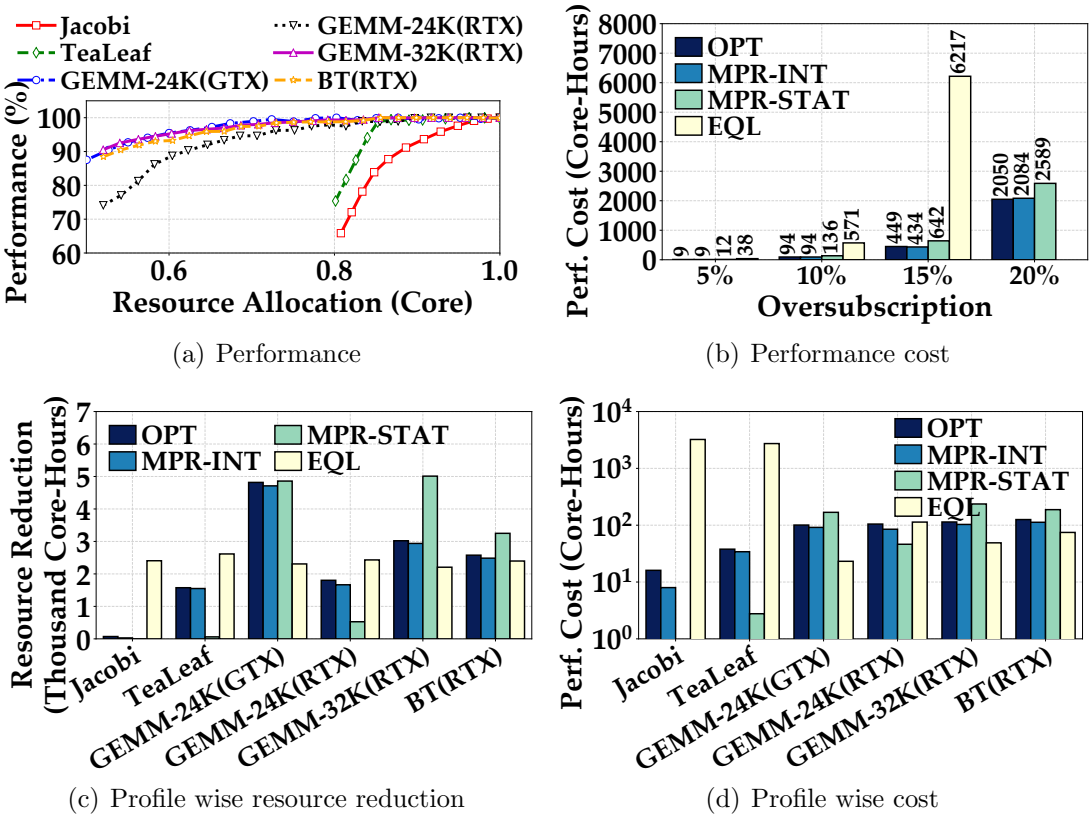
(e) Metacentrum- core allocation

(f) Metacentrum- cost

Figure 3.14: Performance comparison of MPR under different workload traces demonstrating its effectiveness in various scenarios.

same as OPT for all traces. EQL suffers from much higher performance cost compared to MPR-INT and OPT while MPR-STAT also incurs higher costs than OPT.

Heterogeneous system with GPU. To evaluate MPR using a heterogeneous HPC system, we collect six different HPC applications' power and performance data



(a) Performance

(b) Performance cost

(c) Profile wise resource reduction

(d) Profile wise cost

Figure 3.15: MPR under a heterogeneous system with GPUs.

on GPU nodes from [101, 102]. The resource-performance relation of the six applications is shown in Fig. 3.15(a). `Jacobi` and `TeaLeaf` are from [101] and runs on NVIDIA P40 GPUs. Meanwhile, the `GEMM` and `BT` are from [102] running on NVIDIA GTX 1070 and RTX 2080 GPUs. We use `Gaia` trace for this evaluation. We normalize each application's maximum power to "one core" allocation to maintain generality. For instance, for `Jacobi` and `GEMM`, "one core" is when the power consumption is 225W and 200W, respectively.

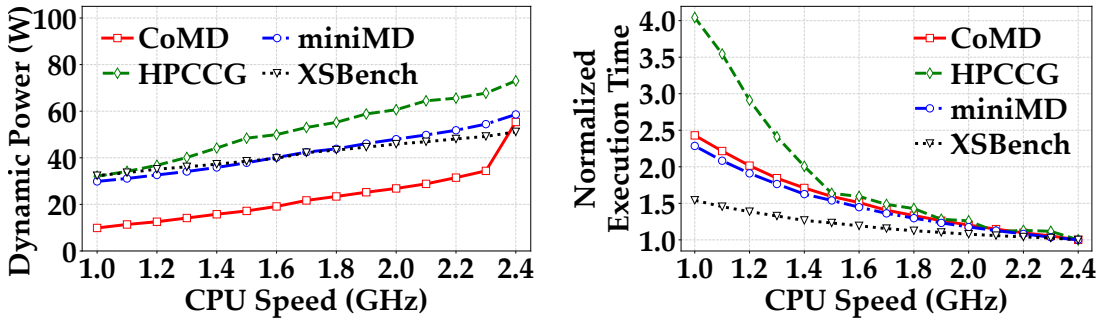
Fig. 3.15(b) shows the overall performance cost under different levels of oversubscription. The results are similar to our prior evaluation using a homogeneous CPU-based system. MPR-INT performs at the same level as OPT, while MPR-STAT incurs additional costs. EQL, in this case, performs much worse and cannot even pro-

vide a feasible resource allocation at 20% oversubscription. As shown in Figs. 3.15(c) and 3.15(d), this is because `Jacobi` and `TeaLeaf` suffer significantly more performance loss under EQL due to resource reduction, while the other algorithms do not ask for much resource reduction from these two performance-sensitive applications. These results highlight that *performance oblivious approaches will suffer significantly when managing HPC applications with a diverse resource-performance relation.*

3.5.6 Prototype Experiment

We run MPR on a prototype HPC cluster consisting of two Dell PowerEdge servers with a total of 40 Intel Xeon CPU cores and 256GB of memory. We implement four applications from our simulation study - `CoMD`, `HPCCG`, `miniMD`, and `XSBench`. We run these applications each with 10 CPU cores. We use `acpi-cpufreq` driver for Linux to control the CPU frequency for resource/power reduction [103]. Fig. 3.16(a) shows the dynamic power of the applications as we change the CPU speed from 1GHz to 2.4GHz. Fig. 3.16(b) shows the corresponding execution times normalized to each application's execution time at 2.4GHz. In both figures, we see that the impact of CPU speed change is different for different applications. This supports the need for MPR's application/user-level control approach.

Next, we run two 30-minute experiments - one without MPR and one with MPR, where we create overload conditions by setting the power capacity at 400W. As shown in Fig. 3.17(a), MPR handles the overload by reducing the power by nearly 50W by slowing down the CPU speeds (i.e., reducing resource allocation) of the applications. In Fig. 3.17(b), we see that different applications reduce different amounts of resources based on their performance impact and bids. We devise the bids for these applications based on their performance impact (Fig. 3.16(b)) and follow the steps outlined in



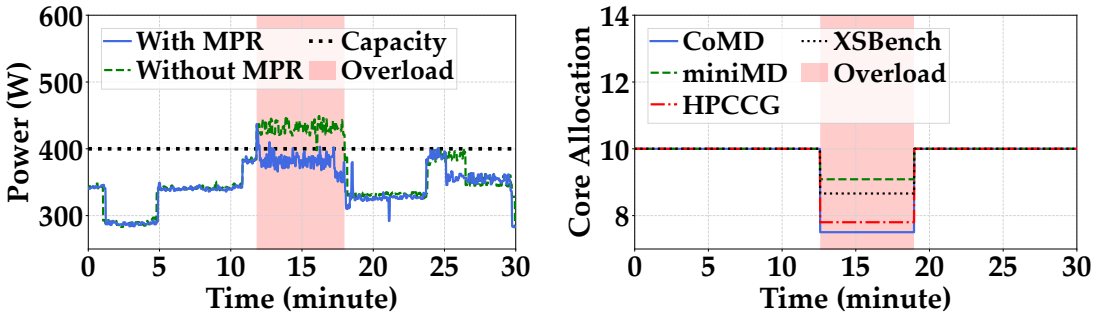
(a) Power (b) Performance

Figure 3.16: Impact of CPU speed change on dynamic power and execution time.

Section 3.3.3. Our prototype experiment, albeit on a small scale, demonstrates the effectiveness of using MPR in handling power overloads due to HPC oversubscription.

3.6 Related Work

Power overprovisioning in the cloud and HPC systems. Power oversubscription in hyper-scale/cloud data centers has been actively studied to overcome infrastructure underutilization and save capital investment (e.g., [57, 104, 105]). However, power overprovisioning in HPC systems has been relatively less explored but gaining traction in recent years. Works on HPC overprovisioning focus on tackling the job scheduling to satisfy the ensuing operational constraints [63, 68, 106, 107]. Khemka et al. [108, 109] develops dynamic resource management techniques to safely oversubscribe heterogeneous distributed systems. Xiong et al. [64] discuss the interference problem that can be introduced when colocating applications on oversubscribed nodes. The authors then propose an application framework to colocate HPC applications by combining offline profiling, machine learning, and scheduling. Sakamoto et al. [66] explores power-aware resource management techniques at scale in overprovisioned HPC systems. In [67], authors develop a hardware overprovisioning system that allocates extra nodes to the system. The proposed system includes various



(a) Power (b) Resource allocation

Figure 3.17: Demonstration of MPR on our prototype HPC cluster.

strategies for dynamic allocation of power capping, various node on-off techniques, and job scheduling techniques. Patel et al. [65] presents a power management framework to improve the system throughput of a hardware-overprovisioned HPC system while ensuring fairness among concurrently running jobs.

Mechanism design applications. Mechanism design is widely-used in many real-life applications. Vickery Clarke Grove (VCG) auction is a sealed-bid auction mechanism, where bidders submit their bids of items with unknown information about other bidders. The mechanism rewards users for their true valuations of the items. VCG auction has been widely used in different fields, including network communication [110, 111], crowdsourcing [112, 113], smart grid [114], among others. Although VCG auction mechanism is efficient and incentive compatible, the mechanism requires the users to reveal their cost functions, which are private function.

Supply function bidding is a cost-efficient mechanism that ensures optimality at a Nash equilibrium. Compared to other mechanism models (e.g., VCG auction) supply function is simpler and does not reveal the private cost function of the users. Supply function has been applied in various applications, such as demand response [90, 91] and power emergencies [59].

3.7 Conclusion

In this paper, we presented MPR, a market-based approach to managing oversubscribed HPC systems. MPR enables HPC users' participation in resource reduction in exchange for rewards. Using extensive real-world trace-based simulation, we showed that both HPC users and HPC managers are highly incentivized for their market participation. To the best of our knowledge, the solution outlined in this paper is the first market-based approach to handle power oversubscription in an HPC system via active user participation.

3.8 Acknowledgments

This work is supported in parts by the US National Science Foundation under grants ECCS-2152357 and CNS-2300124.

CHAPTER 4

Enabling Workload-Driven Elasticity in MPI-based Ensembles

4.1 Introduction

The end of Dennard scaling together with the progressive waning of Moore's Law has caused an explosion of parallelism and a rapid increase of resource heterogeneity and complexity, making it much harder to run and reproduce multidisciplinary scientific workflows [115]. To cope with increased complexity, resource management needs to adapt to schedule diverse workflow components, including multi-scale simulations, in-situ data analysis, and Artificial Intelligence and Machine Learning (AI/ML) techniques [116–119], accounting for changing events, resource dynamism, and costs [115]. To add complexity, each component may be mapped to a specific on-node resource (e.g., GPU or dedicated accelerator) or even to a separate cluster with the desired hardware capabilities [120].

The increase in system parallelism has encouraged a transition to ensemble techniques for simulation and uncertainty quantification (UQ) [121]. Ensembles are a core component of HPC workflows for drug discovery [122, 123], molecular screening [120], cancer research [124], inertial confinement fusion [125, 126], and weather and climate models [127–129] among many others. At Lawrence Livermore National Laboratory, one of the largest scientific computing centers in the world, nearly 50% of jobs on large production clusters are ensemble-based [130]. HPC ensemble-based workflows can adapt dynamically based on Machine Learning (ML) models and incorporate in-situ feedback while using Message Passing Interface (MPI) for intra-job communication [124, 131], and be composed of tens [128, 132] to over 100 million

jobs [126]. Ensemble-based composite HPC workflows present significant challenges in deployment, portability, reproducibility, and automated management. Running complex scientific workflows on systems of increasing size and heterogeneity with resource dynamism compounds the challenges.

New economic forces are shaping the future of computing, leading to innovations to manage system and workflow complexity. A new fragmentary economic cycle creates specialized computing islands that bring fewer benefits to each other; communities that cannot use innovations financed and produced by market leaders will be left behind [133]. Cloud computing is on pace to overtake all other computing sectors by 2025, attaining nearly $1T in total revenue [7].

As cloud computing becomes a dominant market force, it drives innovation in both hardware [133] and the software needed to manage the rapidly increasing resource diversity, scale, and complexity of current and future systems.

Converged computing, or an environment that provides the performance and efficiency of HPC together with the automation, portability, and reproducibility of the cloud, is a promising emerging area of computing that seeks to address challenges faced by complex scientific workflows. K8s [30], the de-facto standard container orchestration framework, provides native support for automation, reproducibility, and elasticity. K8s' vast (over 90,000) contributor base and widespread adoption in the industry have made it the second largest open-source software project after Linux [134, 135]. The scale of adoption and reliance on K8s makes it an excellent choice for ensuring workflow portability.

From the HPC side of converged computing, Flux, a hierarchical, graph-based resource management and scheduling framework, was created to solve portability, throughput, and scheduling challenges faced by complex scientific workflows running at exascale [130]. The Flux Framework features rich application programming in-

terfaces (APIs) and a modular design that facilitates integration into cloud technologies [130,136,137]. Flux and K8s are well-matched technologies that, when integrated, can bring the benefits of converged computing to workloads.

Creating a seamless HPC+K8s converged environment that supports HPC- and cloud-oriented components of workloads requires K8s to adopt characteristics of HPC, and HPC those of the cloud. Large, dynamic MPI ensemble-based complex workloads running in static HPC allocations can experience long startup times, leading to resource under-utilization [131]. The ability to dynamically increase the size of an HPC resource allocation and improve efficiency through autoscaling remains a highly desirable [131] but an unrealized goal. There have been recent advancements in the ability to run MPI-based workloads at scale using the automation of cloud-native orchestration in K8s [138, 139], but unlocking efficiencies of elasticity also requires autoscaling capability in HPC.

In this study, we develop and implement a workload-driven autoscaling strategy that adjusts the number of nodes running an ensemble based on the length of a job queue and the application median runtime. We perform a series of experiments to measure the performance, overheads, and costs of running ensembles of four well-known MPI-based proxy applications under static and autoscaled resources via the Flux Operator running on Elastic Kubernetes Service (EKS) in Amazon Web Services (AWS).

Specifically, we make the following contributions:

- Develop a workload-driven autoscaling strategy to change the number of nodes running an MPI-based ensemble workload
- Analyze end-to-end runtime and dollar cost of ensembles in three configurations—static allocation, full autoscaling, and workload-driven autoscaling

- Demonstrate that workload-driven autoscaling costs less and reduces ensemble runtime in comparison to fully automatic autoscaling and enables a trade-off option that balances cost and runtime
- Measure and analyze overheads of operations such as cluster creation and resource scale-out operations

4.2 Background

4.2.1 Flux Framework

Emerging scientific workflows present throughput, portability, co-scheduling, and job coordination challenges that cannot be addressed by current HPC resource managers and schedulers [130].

The Flux Framework [140] supports hierarchical resource management, which allows it to instantiate external resource manager instances (or itself) for portability [140] and high throughput [130]. Its modular architecture and APIs facilitate integration with the cloud [137, 138]. Additionally, the Flux Framework's directed graph-based resource model, sophisticated scheduling policies, and resource management algorithms provide flexible representation to enable workloads to run efficiently.

Flux follows a leader-follower architecture. A Flux broker is a distributed message broker that runs in each cluster node. A single dedicated leader is the root of a tree-based overlay network to which other follower brokers connect. The brokers can self-identify their roles via a shared system configuration file. This setup works equivalently on HPC nodes as it does on "nodes" in the cloud, which are typically virtual machine instances. Thus, in this paper, we use the terms "node" and "instance" interchangeably.

4.2.2 Kubernetes

There are several compelling reasons behind K8s' widespread adoption compared to other orchestration solutions. Its declarative management approach allows users to define the desired state, and K8s automatically maintains that state and recovers from failures. Although K8s typically runs as an infrastructure as a service in the cloud, it can be deployed on premises. We describe several K8s components relevant to the study in the following sections.

4.2.2.1 Pods

A Pod [141] is a fundamental component of application deployment, grouping one or more containers. These containers share storage and network resources. To ensure isolation between Pods, Kubernetes utilizes Linux namespaces and cgroups.

4.2.2.2 Horizontal Pod Autoscaler

The K8s Horizontal Pod Autoscaler (HPA) [40] is a dynamic control mechanism that adjusts the number of Pod replicas in an application deployment or replica set based on specified metrics or resource utilization. It periodically evaluates resource metrics such as CPU and memory utilization or custom metrics. It then automatically adjusts the number of Pod replicas within a deployment or replica set to maintain a specified resource utilization target.

4.2.2.3 Cluster Autoscaler

The Cluster Autoscaler (CA) [142] is a tool that automatically adjusts the size of a K8s cluster based on resource demands. It increases the cluster size when resource shortages are causing Pod scheduling failures and decreases it when nodes

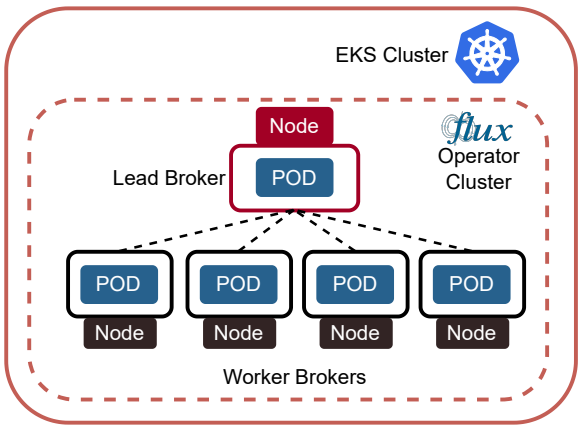


Figure 4.1: Example of HPC with cloud computing convergence used in our study. The Flux Operator is deployed in a Kubernetes cluster, which deploys an HPC cluster in K8s. The K8s cluster is managed by, e.g., AWS.

are consistently underutilized, ensuring resource utilization and improving efficiency. Cloud providers provide cluster autoscalers for their respective resources, including AWS. The CA operates by first identifying the autoscaling group designated for scaling instances. It constantly monitors the K8s cluster for pending Pods. When it identifies pending Pods, it assesses whether the autoscaling group's configuration can accommodate them, and if so, requests the cloud provider to increase the number of instances.

4.2.3 Convergence of Flux and K8s

A K8s operator is a controller that uses domain-specific knowledge to automate the entire lifecycle of complex applications within the K8s ecosystem. The Flux Operator [143] was developed to deploy an entire Flux cluster on demand. When deployed in Kubernetes, this Custom Resource is called a MiniCluster. Each K8s Pod running a Flux broker is mapped to a node, and the network to connect the nodes is provided by a K8s Headless Service. Pods are mapped one-to-one to K8s nodes to manage hardware resources efficiently. Several features distinguish the Flux Operator

from other MPI operators, such as interactive modes, queue monitoring, bursting to external resources, and, most importantly, elasticity (ability to expand/retract resources dynamically). The Flux MiniCluster can scale due to the Scale Subresource [144], which increases the size of the underlying indexed Job that comprises the cluster nodes. When new follower brokers appear (running pods) and register with the lead broker, the Flux workload manager registers them as up nodes from a down state. This ability to grow in size is essential for both autoscaling strategies described in this work. Due to its unique design and utilization of K8s resources efficiently, the Flux Operator consistently outperforms the MPI operator [145] by completing the execution of an application at least 5% faster [143]. Figure 4.1 shows our adaptation of the converged computing utilizing the Flux Operator and K8s.

4.2.4 Public Cloud

Public cloud providers, such as AWS, Google Cloud Platform (GCP) [146], Microsoft Azure [147], and IBM Cloud [148] offer scalable and elastic infrastructure as a service (IaaS), enabling access to compute resources, storage, and services on-demand. The scale and elasticity of the cloud model poses a challenge for the connectivity needs for HPC applications.

Unlike HPC, cloud clusters are often geographically or spatially distributed. We run our experiments on AWS, which uses the Elastic Fabric Adapter (EFA) [149–151]. EFA provides lower latency communications for HPC applications running across instances. EFA uses a specialized operating system bypass mechanism to enable HPC applications to scale on AWS [152]. To improve performance, AWS suggests that EFA be used with a placement group. Placement groups [153] ensure low-latency communication by launching instances within the same subnet, binding them to the same Availability Zone [154], reducing data transfer time [155].

4.2.5 MPI-based Ensembles

Ensembles are workloads composed of similar runs of an application, differing by initial conditions or parameters and often based on MPI. Ensembles can number tens [128, 132] to hundreds of millions [126] of elements or members. Ensembles can have a fixed number of members or be dynamic [120,124,131], depending on the input or parameter space to explore. They are frequently part of a larger complex workflow, which may have AI/ML components or data analysis stages that create new ensemble members. Ensemble-based workflows running near exascale can suffer underutilization due to startup delays caused by dependence on input generation [124,131]. Ensembles, AI/ML, and data analysis components of emerging workflows can benefit from improving utilization via cloud techniques like autoscaling.

4.3 Methodology

Enabling elasticity in MPI-based ensemble workflows poses unique challenges. For example, cloud autoscaling solutions are designed for microservices with variable resource utilization and dynamic traffic. However, autoscaling solutions designed for microservices are impractical for HPC applications, which often exhibit 100% CPU utilization. Transitioning HPC to the cloud requires understanding various cloud features such as instance acquisition time or the time to pull large container images. Additionally, setting up an HPC cluster in the cloud is complex, necessitating automation. In the next section we address these challenges and discuss our contributions to the autoscaling of ensemble workloads and the performance analysis on running HPC in the cloud.

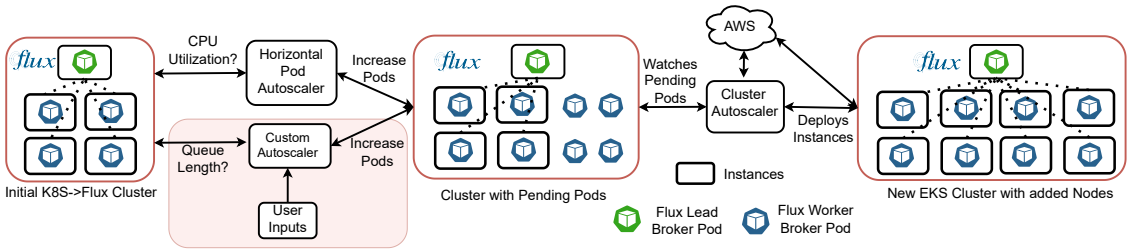


Figure 4.2: Flow of fully automatic autoscaling and workload-driven autoscaling mechanism. Given jobs that each require 4 nodes, a fully automatic strategy (top) scales based on CPU utilization, often resulting in extra pods that cannot be utilized. A workload-driven mechanism (bottom and marked area) adds nodes that fit job requirements exactly.

4.3.1 Autoscaling

We studied the benefits of elasticity on MPI-based ensemble workloads using two approaches. First, we examined the default autoscaling behavior of K8s with the Flux Operator and MPI-based ensemble workloads. Then, we developed a workload-driven autoscaling approach where the workload manager can launch instances based on parameters such as the job queue size, time to get instances from the cloud, and the median job completion time. We compared these mechanisms' time and dollar costs with the baseline approaches of no autoscaling.

Figure 4.2 shows the overall system flow diagram of the two autoscaling approaches. A K8s cluster with a fixed number nodes is deployed. Then the Flux Operator deploys the application container to a pod on each node, adding Flux to create a MiniCluster. The Flux MiniCluster is responsible for maintaining the Flux Operator Pods cluster's size and deploying one Pod per physical instance to maintain the one-to-one mapping. Then, the MPI-based ensemble application is launched. Each ensemble task (also referred to as a job) is submitted to the Flux *lead* broker. The jobs are scheduled by the Flux scheduler based on resource availability. If resources are unavailable, jobs have to wait in the queue.

4.3.1.1 Fully Automatic

This setup is fully automatic because no intervention or external input is required for the autoscaling decision. Fully automatic autoscaling involves the integration of two crucial components within the K8s orchestration system: the Horizontal Pod Autoscaler (HPA) [40] and the Cluster Autoscaler (CA) [142].

HPA tracks the CPU utilization of all Pods managed by the Flux Operator and increases the number of Pods to maintain a CPU utilization target. As Flux Operator Pods occupy the entire resources of one node, the newly deployed Pods stay pending due to the unavailability of the instances [143].

The CA considers the requirements of the pending Pods and finds matching instances from AWS that can host these Pods. If it finds matching instances satisfying user-provided constraints, it launches the instances from AWS by communicating via the AWS cloud API.

Once these new nodes are successfully integrated into the K8s cluster and become visible to the K8s control plane, the pending Pods are scheduled onto these nodes. Finally, the Flux follower brokers are initiated on the newly added nodes, allowing Flux to recognize and seamlessly incorporate these nodes into its cluster. The Flux lead broker can then launch pending jobs from its queue.

4.3.1.2 Workload Driven

For our workload-driven mechanism, we devised a strategy that determines required resources rather than relying solely on automated response to CPU utilization. Figure 4.2 shows the steps involved in the full and workload-driven autoscaling setup. In the case of workload-driven autoscaling, instead of relying on HPA, we utilize custom metrics and Algorithm 2.

Algorithm 2 Workload-Driven Autoscaling

Input: Median job runtime X, job resource requirement R, instance acquisition time I, max num. of instances N_{max}

Output: Scaling Operation, num. nodes N at job completion event t (SCALEUP/SCALEDOWN/NOACTION, N_t)

1: **for** each job completion event t **do**
2: Get num. jobs in the queue J_t, current num. nodes N_t
3: $R_t = R(J_t - \lceil \frac{I}{X} \rceil - 1)$ Num. nodes required for queued jobs
4: **if** $R_t < N_t$ **then**
5: return SCALEDOWN by R_t
6: **else if** $R_t == N_t$ **then**
7: return NOACTION, N_t
8: **else**
9: return SCALE UP $\min(R_t, N_{max})$
10: **end if**
11: **end for**

First, the cluster is created with an initial number of instances, which in our study was set equal to the number required by each job (8). The choice of an initial cluster size of 8 instances allows one job to run immediately. Our approach described in Algorithm 2 starts after jobs are submitted to the queue and executes at the completion of each job. While the strategy can scale the number of cluster nodes up or down, the parameters we selected in our experimental work caused the cluster to scale up to the maximum number of instances (Nmax = 48) upon completion of the first job.

Recognizing the existence of pending Pods, the cluster autoscaler responds by provisioning new EC2 instances, aligning with the demands signaled by the pending Pods. Once the new instances are added to the cluster, the pending Pods are scheduled, and the Flux follower brokers in those Pods start executing and connecting to the lead broker. Once the scheduler detects newly available resources satisfying job requirements, the resource manager starts queued jobs on the resources.

4.3.1.3 Study of Performance Overhead

Along with developing an autoscaling strategy for ensemble workloads, we also investigated the unique issues that can arise when running HPC applications in a converged computing setting. For example, with elasticity, when we request instances from the cloud there is a waiting time to provision the instances. Moreover, we also need to understand how container pulling impacts application launch time and incurs additional costs. Moreover, the container images built for HPC applications can be larger than traditional cloud service-oriented applications. Requesting hundreds of them from a central registry at the same time may create a bottleneck.

To automate the deployment of cloud infrastructure and measure these times, we used a tool called "kubescaler" that uses the Kubernetes API and automates the deployment of clusters, controls resource elasticity, and records timings of various events that are important when running MPI-based ensemble in the cloud [156]. The tool automates the deployment of an EKS cluster to aid in performing the scalability study of EKS.

4.3.2 Selection of MPI-based ensemble

For our evaluation, we used four benchmarks from the CORAL2 [157] suite: LAMMPS [158], AMG [159, 160], Kripke [161], and Laghos [162, 163]. CORAL-2

benchmarks have been designed specifically for performance analysis of upcoming supercomputers. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [158] is a classical molecular dynamics (MD) code that models a large set of particles. The application can be run in serial mode, in distributed parallel mode with MPI, in multithreaded mode with OpenMP, and in hybrid MPI and OpenMP mode. AMG [159, 160] is a parallel multigrid solver that solves linear systems on unstructured grids. Kripke [161] is a scalable 3D deterministic particle transport code designed to study the impact of data layout, programming paradigms, and architectures on implementation and performance. Laghos (LAGrangian High-Order Solver) [162, 163] is a mini-application designed to solve the time-dependent Euler equations for compressible gas dynamics.

We created an ensemble workload by running 20 identical jobs of each application. These jobs are identical in their resource requests and in the initial conditions or parameter space they explore. We also ran a larger ensemble with 100 jobs and an ensemble with varying application problem sizes to study the effectiveness of workload-driven autoscaling on ensembles with a greater number of jobs and variable job runtimes, respectively.

4.3.3 Determining Utilization Threshold

To determine the appropriate target CPU utilization for HPA in a fully automatic setup, we conducted a simulation considering that HPC applications typically maintain a consistent CPU utilization close to 100%. Figure 4.3 shows the corresponding new instance counts for desired utilization thresholds. By varying the target CPU utilization, it was observed that starting above 50% was necessary to prevent unbounded behavior. When the desired metric value was set below 50%, the HPA continuously doubled instance counts due to the rapid job launches on newly

Table 4.1: Node and Cluster Setup

Module	**Attributes**
Kubernetes Version	V1.27
eksctl version	base - v0.168 with customization for ARM
EFA	enabled
Placement Group	enabled
Instance CPU	64 ARM cores
Threads Per Core	1
Instance Memory	128 GB
Instance Network	200 Gigabit
Initial Cluster Size	8 nodes
Total Ranks	512
Ranks in each node	64

added nodes, resulting in a repetitive cycle. Conversely, setting the metric to 100% prevented HPA from adding instances. As a balanced approach, a target metric value of 80% was chosen for the experiments. Moreover, our experiments show that the target metric value is also independent of application end-to-end completion time.

4.4 Experimental Setup

4.4.1 Autoscaling Study

This section describes the experimental design for testing a novel autoscaling strategy in the context of deployment overhead such as time to completion and cost. For each of a suite of MPI applications, we create a cluster that is either static (no autoscaling) or dynamic (autoscaling). For each cluster we submit a set of jobs and allow the jobs to run given static resources or with an autoscaling strategy. We compare our workload-driven autoscaling approach against CPU utilization-based autoscaling and static cluster sizes.

4.4.1.1 Cluster Creation

Both static and dynamic EKS clusters are created using the `eksctl` [164] command line tool. We measured the cluster creation times for all autoscaling study experiments. Once the cluster and associated resources are available, we set up the Flux Operator using the kubectl command line tool. This deploys a "MiniCluster," an entire HPC cluster (Flux Framework) in K8s with lead and follower brokers connecting pods [143]. The job submission of each experiment is performed from the lead broker pod. For a static cluster the size (8, 16, 32, or 64 nodes) is constant. A dynamic or autoscaling cluster starts with a preliminary size of 8 nodes that is then scaled up or down depending on application needs.

4.4.1.2 Ensemble Job Design

An ensemble is composed of a group of jobs, where each job occupies a specific number of nodes. For all of our experiments, we chose a job size of eight nodes as a baseline configuration. For these experiments, we used `hpc7g.16xlarge` nodes with 64 CPU cores. This choice was driven by cost and temporal feasibility – obtaining hundreds of nodes in the cloud would have been time-consuming and costly. Thus, this choice is practical, and strikes a balance between representing an HPC environment and managing costs effectively. For the dynamic, autoscaling clusters, we chose to match the initial cluster size to the job size to ensure greater resource utilization at the onset of creation of a cluster. The workload-driven and fully automatic setups add more instances to the eight nodes. Additional details of the node and cluster setup are available in Table 4.1.

Figure 4.3: HPA calculates new instances based on desired utilization value and current utilization

4.4.1.3 Application Setup

Applications, autoscaling strategies, iterations, problem sizes, and parameters are detailed in Table 4.2. Problem sizes were chosen explicitly to achieve a desired runtime of 2-3 minutes for each job, and exact median runtimes were recorded for each application to inform the strategy. For AMG, the largest possible problem size that would run on the `hpc7g.16xlarge` instance without running out of memory resulted in a runtime comparable to LAMMPS. Parameters were chosen for Laghos that ensured a longer runtime to introduce runtime heterogeneity in the experiments.

For this set of experiments, each application was run at least 20 times. A time span of a few minutes is short enough to run substantial repetitions to measure variability, but not too long to add cost without additional benefit to our analysis. The workload-driven autoscaling strategy (Algorithm 2) considers the median job runtime and the time it takes to acquire new instances from the cloud, and longer median runtimes do not impact behavior. Further, longer runtimes are more susceptible to cloud maintenance and failures. Each job is launched with 64 MPI ranks for each of 8 nodes, resulting in 512 ranks in total.

To test the applicability of the workload-driven algorithm to larger ensembles and varying ensemble member runtimes we ran AMG experiments with an increased number of iterations and varying problem sizes. These experiments are "Larger Ensemble" and "Runtime Variability," respectively (see Table 4.2). For the varying problem sizes, we generated parameters by sampling from a normal distribution, imposing an upper bound on the distribution to ensure successful job execution given instance memory constraints. As an example, we set the mean to 150 for a particular job parameter with a standard deviation of 5 and then constrained the values to fall within the range of 140 to 160 by setting the lower and upper bounds. This design is important to the study because ensemble jobs can exhibit variability in problem sizes or parameters, leading to varying runtimes.

Table 4.2: Autoscaling Experiments *S (static), F (fully automatic), W (workload-driven)*

Experiment Name	Application	Autoscaling-Strategy	Ensemble Size	Parameters
Autoscaling Study	LAMMPS	S8 S16 S64 F W	20	(problem size) `64x16x16`
	AMG	S8 S16 S64 F W	20	(problem size) `160x145x70`
	Kripke	S8 S16 S64 F W	20	`--groups 500 --zones 64,64,64--procs 16,8,4`
	Laghos	S8 S16 S64 F W	20	(mesh) `-m cube_211_hex.mesh` (solver and mesh refinements) `--ode-solver 7 -rs 4 -rp 1` (steps) `--max-steps 160`
Larger Ensemble	AMG	S8 S16 S64 F W	100	(problem size) `160x145x70`
Runtime Variability	AMG	S8 S16 S64 F W	20	(problem size) `varying (Section 4.4.1.3)`

4.4.1.4 Pods and Container Timings

The time to pull an application container from a registry until the time it can run an application is an important consideration when assessing experimental costs. Further, in the context of autoscaling when containers must wait for a node allocation to be scheduled, pending time is important to assess. We used the Kubernetes API to record pod events (creation, scheduling, and ready) during all repetitions of the automatic and workload-driven experiments. With the timing event data we can

determine the time it takes for a Pod to be scheduled, then be pulled from the registry and to enable it to run. Once all these steps are completed, the new Flux broker Pods join the lead broker of the MiniCluster.

4.4.1.5 Autoscaling Protocol

For each experiment described in Table 4.2, the cluster is created and the Flux Operator deployed (4.4.1.1), and the set of jobs (4.4.1.2) are submitted to run on the Flux MiniCluster. For a static cluster, the jobs are allowed to run to completion with no additional intervention. When autoscaling with the CPU utilization-based (fully automatic) strategy, the cluster autoscaler adds nodes in response to pending Pods based on CPU-utilization. The workload-driven autoscaler uses a metrics from the Flux queue to change the number of nodes based on Algorithm 2. Within the fully automatic setup, we opt for a target CPU utilization of 80% averaged across all pods (Section 4.3.3). We run five repetitions for each of the static and fully automatic experiments, and three repetitions for each workload-driven experiment. Each experiment progresses until all ensemble jobs are completed, and then the cluster is deleted.

To measure the impact on cost of downscaling with the workload-driven strategy we ran two ensembles of 20 AMG jobs with and without downsizing. Workload-driven autoscaling with downsizing enabled allows reduction of the number of nodes when those nodes are no longer running jobs. Without downscaling, the nodes persist and continue to incur costs until the cluster is terminated when all jobs are complete.

4.4.2 Scaling Study

We conducted a scaling study to assess the time and cost overhead of cluster creation and deletion.

4.4.2.1 Cluster Creation

The kubescaler tool [156] provides a means to create and scale Kubernetes clusters on AWS (EKS) and Google Cloud. It provides a Python SDK that leverages the AWS boto3 API for automating the process [165], which is similar to `eksctl` in that it also uses AWS CloudFormation, an infrastructure-as-code service, to model, provision, and manage AWS and third-party resources.

Kubescaler initially creates a CloudFormation stack to establish a Virtual Private Cluster (VPC), which provides a logically isolated virtual network for launching resources [166]. Subsequently, another CloudFormation stack is created to deploy a K8s cluster. Finally, a third CloudFormation stack (a worker stack) is created to set up a managed nodegroup responsible for managing EC2 instances for K8s. When creating the worker stack, it generates an AWS Auto Scaling Group (ASG) in the background and maintains the desired number of instances. To scale the instances of the K8s cluster, the user can adjust the desired size of the CloudFormation stack. The CloudFormation stack then updates the size of the underlying AWS ASG, which in turn updates the EC2 instances. As the managed nodegroup is a part of the EKS cluster, the nodes are launched, and the K8s controller can locate them. During this process, Kubescaler keeps timings in seconds for each operation. These timings include the creation of instances in AWS during CloudFormation stack updates, the time required for these instances to become ready to accept Pod requests from Kubernetes, and the completion of CloudFormation stack updates.

4.4.2.2 Scaling Operations

In conducting the scale-out operations, we used the `hpc6a.48xlarge` instance type with 96 cores and 384GB memory to match our autoscaling experiments. Af-

Table 4.3: EKS cluster creation times (seconds) by size

Cluster Size	Median	Min	Max	Repetitions
8	1087.57	957.08	1230.02	14
16	1151.66	1009.05	1261.20	5
32	1052.12	1033.091	1180.450	4
64	1106.82	971.56	1212.55	4
128	1050.46	1010.97	1119.64	3
256	1280.22	1257.55	1300.012	3

Timings in seconds to create clusters with initial instance sizes of 8-256. Timings of size 8-64 were measured as part of the autoscaling study. Size 128 and 256 tests were run to measure larger cluster creation time and did not include autoscaling or application runs.

ter cluster deployment (Section 4.4.2.1) we systematically increased the number of instances in increments of 4 nodes until a maximize size of 64 nodes, measuring the time of each incremental scaleout. We repeated this procedure with increments of 8 and 16 nodes (each up to 64 nodes), providing us with cluster creation, deletion, and scaling times for all increments. Understanding how long these processes take is an important factor when designing an autoscaling experiment.

4.5 Results

In this section we present results of our performance studies of running MPI-based ensembles in the cloud. We assessed the temporal and monetary cost of autoscaling ensembles (Section 4.5.1), along with cluster creation and deletion times (Section 4.5.2). You can find all of our data and scripts to reproduce the experiments here: https://doi.org/10.5281/zenodo.13247408 [167].

4.5.1 Autoscaling Study

4.5.1.1 Comparison of Autoscaling Strategies

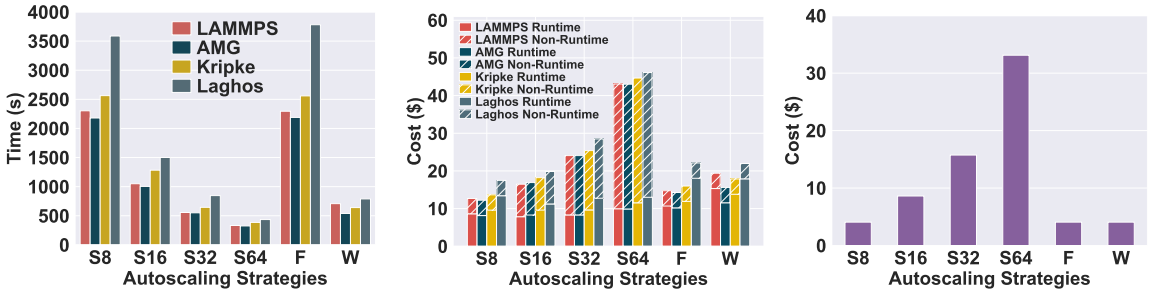
Figure 4.4(a) shows the end-to-end runtime of all ensemble members for each of the five strategies. Median runtimes based on application configurations and resources

were determined to be 116, 110, 128, and 179 seconds each for each of LAMMPS, AMG, Kripke, and Laghos, respectively. The workload-driven autoscaling ensemble completion time is 4× less than the fully automatic time for AMG, and 4.7× less for Laghos. The fully automatic configuration adds node counts that are incompatible with the job requirements. For example, adding seven nodes when eight are needed causes underutilization because seven nodes cannot be used for jobs. The workload-driven strategy adds an integer multiple of the per-job node count requirement, ensuring that enqueued jobs will be started on the resources. Adjusting the strategy to account for downscaling produces similar end-to-end median runtimes (382.37 seconds) as without downscaling (383.97 seconds).

Table 4.3 shows the median, minimum, and maximum times required to create EKS clusters ranging from sizes 8 to 256 nodes. The median time required to launch a cluster does not depend on the number of instances up to 128 instances, and increases by 22% from 128 to 256 instances. Figure 4.8(a) illustrates the time required for creating an EKS cluster and its associated resources, while Figure 4.8(b) shows the timings for deletion and associated resources.

4.5.1.2 Cost

Figure 4.4(b) illustrates the median costs of total end-to-end runtime and non-runtime for various autoscaling strategy. For LAMMPS, the median runtime cost of the static cluster with 8 nodes was $8.62; the workload-driven strategy was $15.32, and the fully automatic scaling $10.68. The static cluster with 64 nodes produces the lowest total ensemble completion time but incurs the highest cost. For our experiments, the largest contributor to the costs are non-runtime costs, as shown in Figure 4.4(c).



(a) Ensemble end-to-end runtime.

(b) Total ensemble costs including non-runtime and runtime

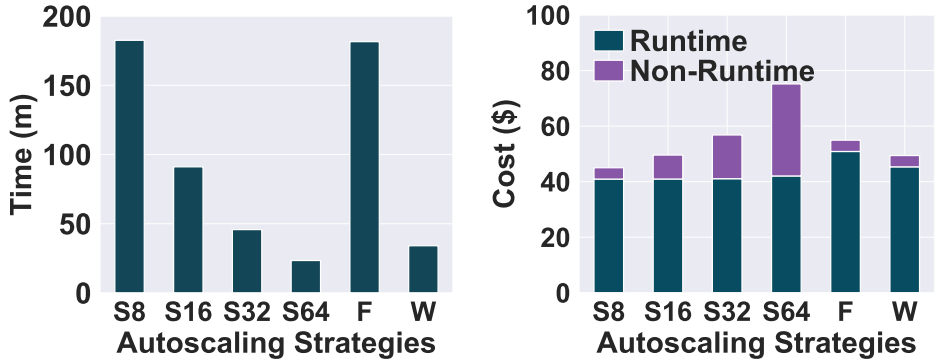
(c) Non-runtime costs

Figure 4.4: Comparison of MPI ensembles end-to-end runtime and costs in various autoscaling strategies for each of a specific size of static (***S***), fully automatic (***F***), and workload-driven (***W***) setups with 3 repetitions. End-to-end runtime is fastest across applications for the static size 64 and workload-driven setups (a), however costs are highest for the same static size 64 setup (b), primarily resulting from the non-runtime costs for the setup (c).

The larger the initial cluster size, the greater non-runtime cost incurred. The workload-driven approach incurred only a 1.43× more cost for a 3.24× lower total completion time than the full autoscaling strategy in LAMMPS. With downsizing enabled, we can save as much as 20% cost over workload-driven autoscaling without downsizing.

4.5.1.3 Larger Ensemble

We assessed the performance of our workload-driven autoscaling strategy for larger MPI ensembles. Figure 4.5(a) shows the end-to-end runtime of ensembles with 100 members, whereas figure 4.5(b) shows the corresponding cost of running the larger ensemble and non-runtime cost of the cluster, respectively. Figure 4.5(a) demonstrates that workload-driven autoscaling outperforms static (except for static-64) and fully automatic setup in terms of runtime and incurs only a small percentage more cost than the static-8 setup.



(a) End to end runtime

(b) Total costs with runtime and non-runtime

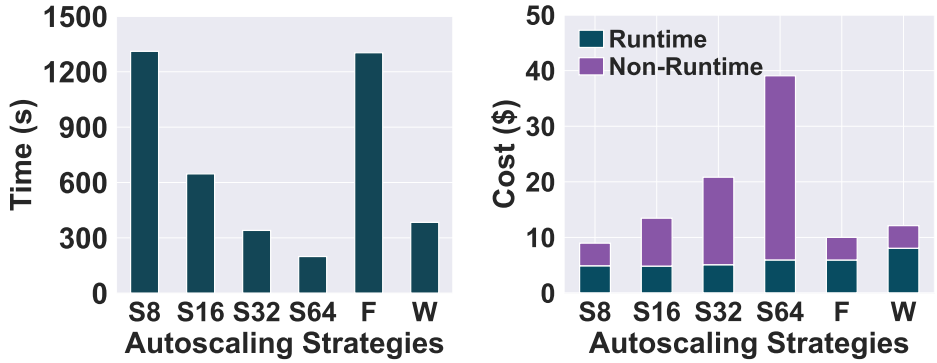
Figure 4.5: Total cost and end-to-end runtime of 100-member ensemble of AMG jobs with fixed resource requirements. The majority of cost across strategies is incurred during setup, and the temporal pattern (a) is consistent with smaller ensemble runs in Figure 4.4(a).

4.5.1.4 Runtime Variability

Figure 4.6(a) illustrates the overall costs of running an ensemble of jobs with greater runtime variability, which is controlled by varying the problem size (Section 4.4.1.2). While there is no strategy that is both fastest and lowest cost, workload-driven autoscaling offers a balance between end-to-end completion time and cost, a trade-off that we will discuss in Section 4.6.

4.5.1.5 Pods and Container Timings

We observed no significant differences in median registry pull times between workload-driven autoscaling (50.1 seconds) and fully automatic strategies (50.02 seconds). However, the median time for pending Pods was lower for workload-driven autoscaling (155.11 seconds) compared to fully automatic (158.27 seconds), with greater variability (standard deviation - 11.85 and 3.28 seconds, respectively). The workload-driven setup results in greater variability because the autoscaler system requests a



(a) End to end runtime

(b) Total costs with runtime and non-runtime

Figure 4.6: End-to-end runtime (a) and total cost (b) of 20 runs of AMG with variable job sizes and fixed resource requirements. Parameters are generated from a normal distribution.

larger number of instances simultaneously, leading to longer wait times for some instances to become available.

4.5.2 Scaling Study

4.5.2.1 Cluster Operations

The operations in this experiment start with creating the VPC and configuring subnets, followed by cluster creation, and conclude with worker stack creation. Figure 4.8(a) depicts the timing for creating a VPC and worker stack or managed node group. The worker stack creation time is measured to be constant. As seen in Figure 4.8(a), the median time for cluster creation is approximately 550 seconds.

We also measure the timings of deletion of the same components upon EKS cluster termination. Figure 4.8(b) displays the timing for deleting the VPC stack, worker stack, and overall cluster. Similar to resource deployment, resource deletion occurs sequentially. Cluster deletion typically takes less time than creation; however, there are cases when cluster deletion may require more time.

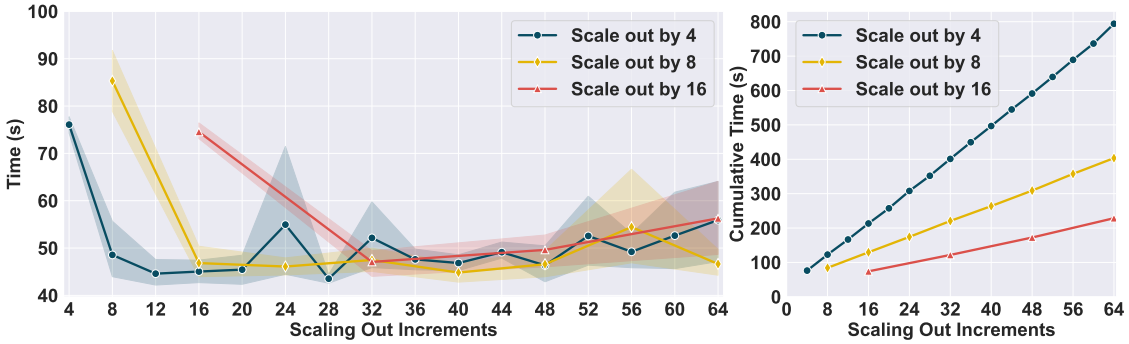
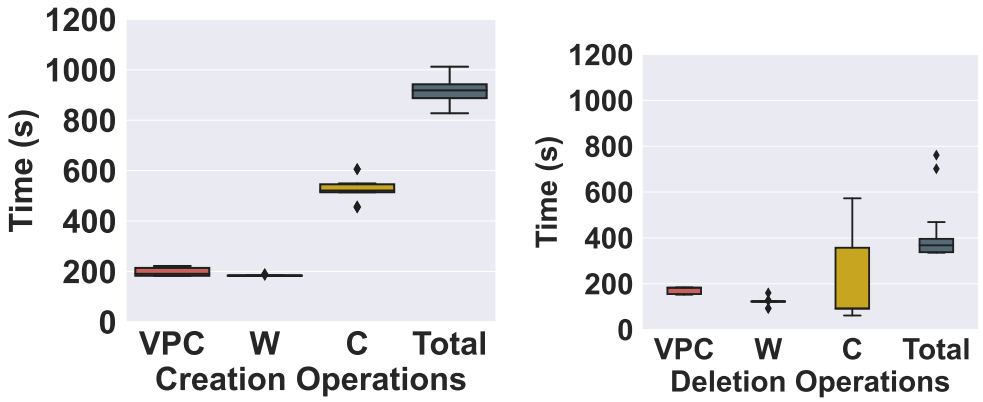


Figure 4.7: Time in seconds to increase the cluster to total nodes (x-axis) by increments of 4 (blue), 8 (yellow), and 16 (red).



(a) Cluster creation Timings (b) Cluster deletion Timings

Figure 4.8: Timings of creating (a) and deleting (b) EKS clusters and supporting stacks. "**W**"-worker stack, "**C**"-cluster creation

4.5.2.2 Scaling Out Operations

Figure 4.7 illustrates the timings of scale-out operations under different instance increment sizes. Scaling out a cluster can be done in large increments less frequently or in small increments more frequently. Since each scaling out operation accrues additional time, selecting a scale out increment that matches the job requirements and minimizes the number of scale out operations is suggested.

4.6 Discussion

Ensembles composed of MPI-based applications are increasingly common in HPC [120–129] and form a large percentage of HPC workloads at large centers like Lawrence Livermore National Laboratory [130]. Ensembles running on static resource allocations can suffer from underutilization due to staged workloads and long startup times. The cloud provides native technologies such as autoscaling and resource dynamism to dynamically manage and provision resources, which can be used to improve the utilization and efficiency of MPI-based ensemble workloads. Autoscaling in the cloud has traditionally used simple metrics like CPU utilization that work well for scaling stateless services. However, it is not clear if CPU utilization accurately accounts for the resource needs of ensembles.

When the needs of a workload, including the number of ensembles, ensemble members, and application parameters are known before runtime, deploying a fixed number of resources that match the exact requirements is sufficient. However, a different strategy is needed for dynamic ensembles, where the amount of work depends on simulation values, statistical properties of output data, or the workload is steered by AI/ML. We describe our key findings in the following subsections.

4.6.1 Novel workload-driven autoscaling strategy

We demonstrate that our workload-driven strategy outperforms both traditional CPU-based autoscaling strategies, along with different choices of static cluster sizes with four HPC proxy applications. Our results showed that the workload-driven autoscaling outperforms both static and fully automatic autoscaling setups by up to 4.7×, while incurring only 1.43× higher costs compared to the static setup. We reproduced this result with a larger ensemble size and with varying problem sizes,

showing the extensibility of the strategy. While our study focused on running MPI-based ensembles, this general pattern can be extended to any kind of application that requires a group of nodes to perform coordinated work concurrently, which also includes AI/ML and cloud-native workloads.

4.6.2 Analysis of end-to-end runtime and costs

Another purpose of our study was to understand the overhead of necessary operations for running MPI-based ensemble applications in the cloud with elasticity. We measured the time required to launch an EKS cluster with varying numbers of instances *and found that the creation time does not depend on the requested number up to 128 instances, and increases by 22% from 128 to 256 nodes.* Additionally, we observed that there is a waiting period to obtain instances from AWS, and to pull and execute containers. Consequently, it is essential to design a scale-out policy strategically to maximize utilization and minimize instance costs. Due to the unknown contributions to non-runtime costs, it is essential for the community to measure, analyze, and understand the component costs of deploying workloads on cloud infrastructure.

4.6.3 Determining the impact of scale-out strategies

In a workload-driven autoscaling setup, we have the option of selecting scale-out strategies, such as scaling out with fewer instances more frequently or scaling out with more instances less frequently. We observe that scaling out by 16 instances proves to be more effective because the less frequent scaling incurs lower waiting times and thus translates to lower cost.

However, scaleup may result in under-utilization of allocation when some of the additional instances are not needed. To address this, we introduced downscaling

(scaling in) to our workload-driven autoscaling strategy. This addition saved us more than 20% in costs compared to the prior version. Based on our findings, we conclude that requesting more instances less frequently is more cost-effective.

4.7 Related Work

4.7.1 Autoscaling HPC Applications

Dynamic autoscaling of high-performance applications is a complex task that involves challenges on multiple fronts. Typically, HPC applications are designed to use a fixed number of resources that can not utilize elasticity provided by the cloud [168, 169]. Elasticity enabling API extensions with MPI V2.0 are proposed and applied to an application with a prototype solution in [169]. Other approaches involve checkpoint-restart mechanisms that lead to application termination and redeployment, which require data redistribution and impact performance [168]. Reinit [170] and Reinit++ [171] support global-restart recovery and ULFM [172] provides MPI fault-tolerance, but these techniques do not support full elasticity.

Substantial effort has been put into autoscaling workflows and dynamic resource management of workflow stages. Liu et al. [173] propose an autoscaling framework for ML workflows. Their work demonstrates that application performance can be gained for ML workflows through auto-tuning the horizontal Pod autoscaler threshold by monitoring and utilizing application run-time metrics like fluctuations in CPU utilization [173]. However, as we demonstrated, such CPU utilization-based auto-tuning mechanisms are ineffective in the HPC domain since these applications are designed to maintain CPU utilization close to 100%. The paper [174] presents a novel autoscaling strategy designed for scientific workflows in cloud computing environments. The

strategy leverages spot instances for cost efficiency and employs a heuristic scheduling method to optimize makespan while mitigating the impact of out-of-bid failures.

Finally, the KubeFlow project [175] provides the MPI Operator [145] and other necessary tools to facilitate MPI-based applications. The MPI Operator bootstraps MPI but does not provide queuing, resource management, or scheduling and struggles to scale [139].

4.7.2 Autoscaling in the Cloud

There is much work on addressing resource management and autoscaling of cloud applications. Some of the work focuses on improving the horizontal autoscaling of web applications [42, 176–179], and some of it focuses on resource allocation and quality of services [9, 12, 17, 25, 180–186]. K8s provides a horizontal autoscaling mechanism based on the utilization threshold of various metrics such as CPU or Memory [40]. The authors of [176] also propose a rule-based autoscaling mechanism based on CPU and memory utilization. The author of SHOWAR [177] uses the variance in historical usage for vertical scaling and a proportional-integral-derivative (PID) controller for horizontal scaling. Nonetheless, SHOWAR still requires extensive tracing from the CPU scheduler for its scaling decision. However, threshold-based approaches work best if there is a variable amount of resource utilization of the applications and some relationship between utilization and performance. HPC applications do not show such behaviors.

These works [9, 12, 25, 27] proposed various techniques and frameworks to find the efficient resource allocation and quality of services for cloud applications. They used machine learning and iterative techniques to identify the root cause of performance bottlenecks and find the optimum resources for iterative applications. The

above approaches are explicitly designed for microservices and can not be readily applicable to MPI-based ensemble workflows.

Recently, AWS developed a cluster autoscaler module called Karpenter [187] to assist autoscaling nodes for the EKS cluster. Karpenter provides several advantages over the cluster autoscaler, such as supporting a full range of instance types across availability zones and providing instances directly from AWS without a managed node group. While Karpenter could have been used to perform autoscaling, the ability to extend our work to other cloud providers was important. We choose not to use Karpenter because it did not have support for additional providers at the onset of this work.

To the best of our knowledge, we are the first to enable autoscaling for MPI-based ensemble workflows.

4.8 Conclusion & Future Work

The work performed in this study provides valuable insights for running MPI ensembles in the cloud. We started with a series of comprehensive tests to measure the costs of deploying, managing, and running ensembles of MPI-based applications within AWS EKS clusters. To improve upon traditional approaches toward CPU utilization-based auto-scaling, we developed and implemented a workload-driven scaling strategy that reduced end-to-end runtime by 3-5×. To inform the sizing strategy for scaling out a cluster, we studied the costs and timings of resource scale-out based on different node increments. While cloud infrastructures and cost models change rapidly, our workload-driven scaling strategy, test methodology, and data presented here provide valuable insights to inform future improvements in cloud workload deployment. Since cloud technologies and their cost models evolve rapidly, it is essential

to understand the interaction between HPC workload deployments and cloud infrastructure costs.

Our findings offer guidance to engineers, developers, and scientists running HPC workloads in Kubernetes environments, but also emphasize the need for resource and allocation elasticity in traditional HPC to reap the same benefits. Efficient resource provisioning and workload-driven resource scaling are essential for improving performance and cost-effectiveness.

In future work, we will focus on enhancing the workload-driven autoscaling strategy by introducing machine learning techniques to adapt and update these strategy parameters dynamically. We will improve upon the deployment and autoscaling strategies performed here by testing advanced metrics such as inter-job dependency representations, dynamic job lengths, variable resource requirements, and variable instance acquisition patterns by AI/ML. Advanced analysis techniques can expose further efficiencies and increase the performance of complex, dynamic workflows composed of traditional HPC applications as well as cloud services and integrated AI/ML.

4.9 Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-864304) and was supported by the LLNL-LDRD Program under Project No. 22-ERD-041 and US National Science Foundation under grants ECCS-2152357 and CCF-2324915. We wish to thank Evan Bollig and Heidi Poxon of AWS for their help deploying MPI workloads on EKS with EFA. We also wish to thank AWS for providing credits that enabled our experimental work. We thank Jae-Seung Yeom for providing constructive feedback to improve the manuscript.

CHAPTER 5

Conclusion

In summary, this dissertation contribute to advancing resource management strategies across different computing paradigms, addressing the growing challenges in cloud computing, microservices, and high-performance computing (HPC) systems.

First, the development of PEMA addresses the critical need for efficient resource management in microservice-based cloud environments. PEMA 's feedback-based approach offers a lightweight, scalable solution that ensures Service Level Objectives (SLOs) are met while optimizing resource usage. This innovation is particularly relevant in the context of cloud-native applications, where the demand for elasticity and cost efficiency continues to rise.

Second, in the domain of HPC systems, the exploration of power management strategies highlights the importance of energy efficiency as we move towards exascale and zettascale computing. The proposed market-based power reduction mechanism introduces a novel way to involve users in managing power consumption, thereby enhancing system utilization without compromising performance. This approach not only addresses the challenges of oversubscription but also provides a scalable, user-centric solution for managing power in large-scale HPC environments.

Finally, the integration of cloud technologies with HPC systems, as explored in the third article, demonstrates the potential for converged computing environments. By enabling workload-driven elasticity in MPI-based ensembles, this research bridges the gap between the automation and flexibility of cloud computing and the performance demands of HPC workloads. The proposed autoscaling algorithm and its

implementation within Kubernetes and Flux frameworks exemplify how cloud-native principles can be applied to enhance the efficiency and scalability of complex scientific workflows.

Together, these articles contribute to the broader goal of developing resource management techniques that are not only effective and scalable but also adaptable to the evolving needs of diverse computing environments. The innovations presented here offer practical solutions for optimizing resource utilization, reducing costs, and improving the overall efficiency of cloud and HPC systems, paving the way for more sustainable and performant computing infrastructures in the future.

REFERENCES

[1] "Sock shop microservice demo," https://microservices-demo.github.io/, accessed: 08/31/2021.

[2] Z. et al., "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[3] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," pp. 323–324, 2018. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[4] Y. G. et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *ASPLOS*, 2019.

[5] D. Feitelson *et al.*, "Parallel workloads archive," http://www.cs.huji.ac.il/labs/parallel/workload/, 2007.

[6] J. Emeras, "Parallel workloads archive: The university of luxemburg gaia cluster log," https://www.cs.huji.ac.il/labs/parallel/workload/l_unilu_gaia/index.html, 2007.

[7] Gartner, Inc., "Gartner Says More Than Half of Enterprise IT Spending in Key Market Segments Will Shift to the Cloud by 2025," https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending, 2022, accessed: Sept 22, 2023.

[8] "Microservice vs monolithic architectures," https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/, accessed: 01/05/2022.

[9] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical Efficient Microservice Autoscaling with QoS Assurance," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 240–252. [Online]. Available: https://doi.org/10.1145/3502181.3531460

[10] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, 2018.

[11] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 223–226.

[12] X. H. et al., "Alphar: Learning-powered resource management for irregular, dynamic microservice graph," in *2021 IEEE IPDPS*, 2021, pp. 797–806.

[13] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[14] "The definition of microservice," https://martinfowler.com/microservices/, accessed: 01/05/2022.

[15] "Introduction to microservices," https://www.nginx.com/blog/introduction-to-microservices, accessed: 01/05/2022.

[16] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 149–161.

[17] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, 2012.

[18] "Google cloud autoscale," https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, last Accessed: 10/05/2021.

[19] "Azure autoscale," https://azure.microsoft.com/en-us/features/autoscale/, last Accessed: 10/05/2021.

[20] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on OSDI*, Nov. 2020, pp. 805–825.

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

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

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

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

[25] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *International Conference on ASPLOS*, 2021, pp. 167–181.

[26] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *ASPLOS*, 2021.

[27] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ASPLOS*, 2019.

[28] "Kubernetes autoscaler," https://github.com/kubernetes/autoscaler, last Accessed: 01/27/2022.

[29] "Docker: Empowering app development for developers," https://www.docker.com/, accessed: 01/20/2022.

[30] "Kubernetes: Production grade container orchestration," https://kubernetes.io/, accessed: 01/20/2022.

[31] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with qos assurance," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 240–252. [Online]. Available: https://doi.org/10.1145/3502181.3531460

[32] "grpc: A high performance, open source universal rpc framework," https://grpc.io/, accessed: 01/20/2022.

[33] "Prometheus - from metrics to insights," https://prometheus.io/, last Accessed: 10/08/2021.

[34] "Linkerd: A different kind of service mesh," https://linkerd.io/, accessed: 08/31/2021.

[35] "Jaeger - end to end tracing distributed tracing," https://www.jaegertracing.io/, last Accessed: 10/08/2021.

[36] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *ACM/SPEC ICPE*, 2019, p. 25–32.

[37] "Amazon aws autoscale," https://docs.aws.amazon.com/autoscaling/index.html, last Accessed: 10/05/2021.

[38] "Kubernetes cpu throttling," https://vmblog.com/archive/2021/10/07/kubernetes-cpu-throttling-the-silent-killer-of-response-time-and-what-to-do-about-it.aspx, accessed: 10/26/2021.

[39] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128609000541

[40] "Kubernetes horizontal pod autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, last Accessed: 10/05/2021.

[41] M. A. Islam, S. Ren, A. H. Mahmud, and G. Quan, "Online energy budgeting for cost minimization in virtualized data center," *IEEE Transactions on Services Computing*, vol. 9, no. 3, pp. 421–432, 2015.

[42] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," in *ACM Comput. Surv.*, vol. 51, no. 4, July 2018, pp. 1–33.

[43] A. F. Baarzi, T. Zhu, and B. Urgaonkar, "Burscale: Using burstable instances for cost-effective autoscaling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 126–138.

[44] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 783–798.

[45] M. Wajahat, A. A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning based application-agnostic autoscaler," *Sustain. Comput. Informatics Syst.*, vol. 22, pp. 287–299, 2019.

[46] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1994–2004.

[47] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 28–37.

[48] K. R. et al., "Autopilot: Workload autoscaling at google scale," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3342195.3387524

[49] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 80–90.

[50] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 427–441.

[51] P. Sharma, A. Ali-Eldin, and P. Shenoy, "Resource deflation: A new approach for transient resource reclamation," in *EuroSys*, 2019.

[52] D. Monroe, "Fugaku takes the lead," *Communications of the ACM*, vol. 64, no. 1, pp. 16–18, 2020.

[53] TOP500.org, "Green 500 list - november 2021," https://www.top500.org/lists/green500/list/2021/11/, 2022, accessed 12$^{\text{th}}$ March 2022.

[54] A. Mann, "Core concept: nascent exascale supercomputers offer promise, present challenges," *PNAS*, 2020.

[55] X.-k. Liao, K. Lu, C.-q. Yang, J.-w. Li, Y. Yuan, M.-c. Lai, L.-b. Huang, P.-j. Lu, J.-b. Fang, J. Ren, *et al.*, "Moving from exascale to zettascale computing: challenges and techniques," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 10, pp. 1236–1244, 2018.

[56] S. Malla and K. Christensen, "The effect of server energy proportionality on data center power oversubscription," *Future Generation Computer Systems*, vol. 104, pp. 119–130, 2020.

[57] V. Sakalkar, V. Kontorinis, D. Landhuis, S. Li, D. De Ronde, T. Blooming, A. Ramesh, J. Kennedy, C. Malone, J. Clidaras, *et al.*, "Data center power oversubscription with a medium voltage power plane and priority-aware capping," in *ASPLOS*, 2020.

[58] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura, *et al.*, "{Prediction-Based} power oversubscription in cloud platforms," in *USENIX ATC*, 2021.

[59] M. A. Islam, X. Ren, S. Ren, A. Wierman, and X. Wang, "A market approach for handling power emergencies in multi-tenant data center," in *HPCA*, 2016.

[60] Y. Li, C. R. Lefurgy, K. Rajamani, M. S. Allen-Ware, G. J. Silva, D. D. Heimsoth, S. Ghose, and O. Mutlu, "A scalable priority-aware approach to managing data center server power," in *HPCA*, 2019.

[61] G. Wang, S. Wang, B. Luo, W. Shi, Y. Zhu, W. Yang, D. Hu, L. Huang, X. Jin, and W. Xu, "Increasing large-scale data center capacity by statistical power control," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–15.

[62] T. Patel, A. Wagenhäuser, C. Eibel, T. Hönig, T. Zeiser, and D. Tiwari, "What does power consumption behavior of hpc jobs reveal?: Demystifying, quantifying, and predicting power consumption characteristics," in *IPDPS*, 2020.

[63] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. De Supinski, "Practical resource management in power-constrained, high performance computing," in *HPDC*, 2015.

[64] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun, "Tangram: Colocating hpc applications with oversubscription," in *HPEC*, 2018.

[65] T. Patel and D. Tiwari, "Perq: Fair and efficient power management of power-constrained large-scale computing systems," in *HPDC*, 2019.

[66] R. Sakamoto, T. Cao, M. Kondo, K. Inoue, M. Ueda, T. Patki, D. Ellsworth, B. Rountree, and M. Schulz, "Production hardware overprovisioning: Real-world performance optimization using an extensible power-aware resource management framework," in *IPDPS*, 2017.

[67] R. Sakamoto, T. Patki, T. Cao, M. Kondo, K. Inoue, M. Ueda, D. Ellsworth, B. Rountree, and M. Schulz, "Analyzing resource trade-offs in hardware overprovisioned supercomputers," in *IPDPS*, 2018.

[68] O. Sarood, A. Langer, A. Gupta, and L. Kale, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *SC*, 2014.

[69] A. Langer, H. Dokania, L. V. Kalé, and U. S. Palekar, "Analyzing energy-time tradeoff in power overprovisioned hpc data centers," in *IPDPSW*, 2015.

[70] E. G. Coffman, G. Galambos, S. Martello, and D. Vigo, "Bin packing approximation algorithms: Combinatorial analysis," in *Handbook of combinatorial optimization*. Springer, 1999, pp. 151–207.

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

[72] T. Patki, Z. Frye, H. Bhatia, F. Di Natale, J. Glosli, H. Ingolfsson, and B. Rountree, "Comparing gpu power and frequency capping: A case study with the mummi workflow," in *WORKS*, 2019.

[73] X. Fu, X. Wang, and C. Lefurgy, "How much power oversubscription is safe and allowed in data centers," in *ICAC*, 2011.

[74] Raritan, "Data center power overload protection," *White Paper*, 2016.

[75] M. A. Islam, S. Ren, and A. Wierman, "Exploiting a thermal side channel for power attacks in multi-tenant data centers," in *ACM CCS*, 2017.

[76] C. Nadjahi, H. Louahlia, and S. Lemasson, "A review of thermal management and innovative cooling strategies for data center," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 14–28, 2018.

[77] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's data center-wide power management system," in *ISCA*, 2016, pp. 469–480.

[78] F. Yang and A. A. Chien, "Zccloud: Exploring wasted green power for high-performance computing," in *IPDPS*, 2016.

[79] K. Ahmed, J. Liu, and X. Wu, "An energy efficient demand-response model for high performance computing systems," in *MASCOTS*, 2017.

[80] "Data center cost: Total cost of ownership (tco) pricing breakdown," https://ongoingoperations.com/data-center-pricing-credit-unions/#closemodal, 2022, accessed 8$^{\text{th}}$ Oct 2022.

[81] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," pp. 68–73, 2008.

[82] APC.com, "Apc 100kva modular power distribution unit," https://www.apc.com/shop/us/en/products/APC-100kVA-Modular-Power-Distribution-Unit-208V-72-Poles-MBP-1-Subfeed/P-PDPM100F-M, accessed 4$^{\text{th}}$ Feb 2022.

[83] D. Klusáček and V. Chlumskỳ, "Evaluating the impact of soft walltimes on job scheduling performance," in *JSSPP*, 2018.

[84] M. Kurokawa, "Parallel workloads archive: The ricc log," https://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html, 2010.

[85] C. Linstead, "Parallel workloads archive: The pik ibm idataplex cluster log," https://www.cs.huji.ac.il/labs/parallel/workload/l_pik_iplex/index.html, 2012.

[86] Predictive Technology Inc, "Four factors that affect data center battery life," https://www.ptisolutions.com/four-factors-data-center-battery-life/, 2020, accessed 18th May 2022.

[87] W. Zheng, K. Ma, and X. Wang, "Hybrid energy storage with supercapacitor for cost-efficient data center power shaving and capping," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1105–1118, 2016.

[88] W. Zheng and X. Wang, "Data center sprinting: Enabling computational sprinting at the data center level," in *ICDCS*, 2015.

[89] R. Johari and J. N. Tsitsiklis, "Parameterized supply function bidding: Equilibrium and efficiency," *Operations research*, vol. 59, no. 5, pp. 1079–1089, 2011.

[90] Y. Xu, N. Li, and S. H. Low, "Demand response with capacity constrained supply function bidding," *IEEE Transactions on Power Systems*, vol. 31, no. 2, pp. 1377–1394, 2015.

[91] N. Li, L. Chen, and M. A. Dahleh, "Demand response using linear supply function bidding," *IEEE Transactions on Smart Grid*, vol. 6, no. 4, pp. 1827–1838, 2015.

[92] N. Chen, X. Ren, S. Ren, and A. Wierman, "Greening multi-tenant data center demand response," *Performance Evaluation*, vol. 91, pp. 229–254, 2015.

[93] F. Kamyab, M. Amini, S. Sheykhha, M. Hasanpour, and M. M. Jalali, "Demand response program in smart grid using supply function bidding mechanism," *IEEE Transactions on Smart Grid*, vol. 7, no. 3, pp. 1277–1284, 2015.

[94] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *PHYSOR*, 2014.

[95] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *HPCA*, 2008.

[96] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: understanding the runtime effects of frequency scaling," in *ICS*, 2002.

[97] C. Li, Z. Wang, X. Hou, H. Chen, X. Liang, and M. Guo, "Power attack defense: Securing battery-backed data centers," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 493–505, 2016.

[98] J. Emeras, S. Varrette, M. Guzek, and P. Bouvry, "Evalix: classification and prediction of job resource consumption on hpc platforms," in *JSSPP*, 2015.

[99] W. Ahmed, S. Fomenkov, and S. Gaevoy, "Reducing approximation time of cluster workload by using simplified hypergamma distribution," in *ICIEAM*, 2018.

[100] A. A. El-Moursy, A. Abdelsamea, R. Kamran, and M. Saad, "Multi-dimensional regression host utilization algorithm (mdrhu) for host overload detection in cloud computing," *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–17, 2019.

[101] R. Azimi, C. Jing, and S. Reda, "Powercoord: A coordinated power capping controller for multi-cpu/gpu servers," in *IGSC*, 2018.

[102] A. Krzywaniak and P. Czarnul, "Performance/energy aware optimization of parallel applications on gpus under power capping," in *PPAM*, 2019.

[103] H. Ye, "Using cpufreq on linux servers to manage power consumption," *Lenovo Press*, 2018.

[104] S. Reda, R. Cochran, and A. K. Coskun, "Adaptive power capping for servers with multithreaded workloads," *IEEE Micro*, vol. 32, no. 5, pp. 64–75, 2012.

[105] C. Zhang, A. G. Kumbhare, I. Manousakis, D. Zhang, P. A. Misra, R. Assis, K. Woolcock, N. Mahalingam, B. Warrier, D. Gauthier, *et al.*, "Flex: High-availability datacenters with zero reserved power," in *ISCA*, 2021.

[106] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. De Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *SC*, 2013.

[107] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz, "Dynamic power sharing for higher job throughput," in *SC*, 2015.

[108] B. Khemka, R. Friese, L. D. Briceno, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, *et al.*, "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2394–2407, 2014.

[109] D. Machovec, B. Khemka, N. Kumbhare, S. Pasricha, A. A. Maciejewski, H. J. Siegel, A. Akoglu, G. A. Koenig, S. Hariri, C. Tunc, *et al.*, "Utility-based resource management in an oversubscribed energy-constrained heterogeneous environment executing parallel applications," *Parallel Computing*, vol. 83, pp. 48–72, 2019.

[110] D. Yang, X. Fang, and G. Xue, "Truthful auction for cooperative communications with revenue maximization," in *ICC*, 2012.

[111] S. Hua, X. Zhuo, and S. S. Panwar, "A truthful auction based incentive framework for femtocell access," in *WCNC*, 2013.

[112] W. Dong, S. Rallapalli, R. Jana, L. Qiu, K. Ramakrishnan, L. Razoumov, Y. Zhang, and T. W. Cho, "ideal: Incentivized dynamic cellular offloading via auctions," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1271–1284, 2013.

[113] D. Yang, G. Xue, X. Fang, and J. Tang, "Crowdsourcing to smartphones: Incentive mechanism design for mobile phone sensing," in *MobiCom*, 2012.

[114] M. Zeng, S. Leng, S. Maharjan, Y. Zhang, and S. Gjessing, "Group bidding for guaranteed quality of energy in v2g smart grid networks," in *ICC*, 2015.

[115] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," 12 2018. [Online]. Available: https://www.osti.gov/biblio/1473756

[116] S. H. Langer, B. Spears, J. L. Peterson, J. E. Field, R. Nora, and S. Brandon, "A HYDRA UQ Workflow for NIF Ignition Experiments," in *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2016, pp. 1–6.

[117] D. Wang, X. Luo, F. Yuan, and N. Podhorszki, "A Data Analysis Framework for earth system simulation within an In-Situ Infrastructure," *Journal of Computer and Communications*, vol. 05, no. 14, p. 76–85, 2017.

[118] U. U. Turuncoglu, B. Önol, and M. Ilicak, "A New Approach for in Situ Analysis in Fully Coupled Earth System Models," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 6–11. [Online]. Available: https://doi.org/10.1145/3364228.3364231

[119] M. Dorier, J. M. Wozniak, and R. Ross, "Supporting Task-Level Fault-Tolerance in HPC Workflows by Launching MPI Jobs inside MPI Jobs," in *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale*

Science, ser. WORKS '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3150994.3151001

[120] D. H. Ahn, X. Zhang, J. Mast, S. Herbein, F. Di Natale, D. Kirshner, S. A. Jacobs, I. Karlin, D. J. Milroy, B. De Supinski, B. Van Essen, J. Allen, and F. C. Lightstone, "Scalable Composition and Analysis Techniques for Massive Scientific Workflows," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 32–43.

[121] D. M. Higdon, M. C. Anderson, S. Habib, R. Klein, M. Berliner, C. Covey, O. Ghattas, C. Graziani, M. Seager, J. Sefcik, *et al.*, "Uncertainty quantification and error analysis," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2010.

[122] L. Casalino, A. Dommer, Z. Gaieb, E. P. Barros, T. Sztain, S.-H. Ahn, A. Trifan, A. Brace, A. Bogetti, H. Ma, H. Lee, M. Turilli, S. Khalid, L. Chong, C. Simmerling, D. J. Hardy, J. D. C. Maia, J. C. Phillips, T. Kurth, A. Stern, L. Huang, J. McCalpin, M. Tatineni, T. Gibbs, J. E. Stone, S. Jha, A. Ramanathan, and R. E. Amaro, "AI-Driven Multiscale Simulations Illuminate Mechanisms of SARS-CoV-2 Spike Dynamics," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/11/20/2020.11.19.390187

[123] S. A. Jacobs, T. Moon, K. McLoughlin, D. Jones, D. Hysom, D. H. Ahn, J. Gyllenhaal, P. Watson, F. C. Lightstone, J. E. Allen, I. Karlin, and B. V. Essen, "Enabling rapid COVID-19 small molecule drug design through scalable deep learning of generative models," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 469–482, 2021. [Online]. Available: https://doi.org/10.1177/10943420211010930

[124] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. Kokkila-Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P.-T. Bremer, J. N. Glosli, and H. I. Ingólfsson, "A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356197

[125] J. L. Peterson, K. D. Humbird, J. E. Field, S. T. Brandon, S. H. Langer, R. C. Nora, B. K. Spears, and P. T. Springer, "Zonal flow generation in inertial confinement fusion implosions," *Physics of Plasmas*, vol. 24, no. 3, p. 032702, 03 2017. [Online]. Available: https://doi.org/10.1063/1.4977912

[126] J. L. Peterson, B. Bay, J. Koning, P. Robinson, J. Semler, J. White, R. Anirudh, K. Athey, P.-T. Bremer, F. Di Natale, D. Fox, J. A. Gaffney, S. A. Jacobs, B. Kailkhura, B. Kustowski, S. Langer, B. Spears, J. Thiagarajan, B. Van Essen, and J.-S. Yeom, "Enabling machine learning-ready HPC ensembles with Merlin," *Future Generation Computer Systems*, vol. 131, pp. 255–268, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22000322

[127] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping Science on Keel When Software Moves," *Commun. ACM*, vol. 64, no. 2, p. 66–74, jan 2021. [Online]. Available: https://doi.org/10.1145/3382037

[128] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. C. Bates, G. Danabasoglu, J. Edwards, M. Holland, P. Kushner, J.-F. Lamarque, D. Lawrence, K. Lindsay, A. Middleton, E. Munoz, R. Neale, K. Oleson, L. Polvani, and M. Vertenstein, "The Community Earth System Model (CESM) Large Ensemble Project: A Community Resource for Studying Climate Change in the Presence of Internal Climate Variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333 – 1349, 2015. [Online]. Available: https://journals.ametsoc.org/view/journals/bams/96/8/bams-d-13-00255.1.xml

[129] C. Tebaldi, K. Dorheim, M. Wehner, and R. Leung, "Extreme metrics from large ensembles: investigating the effects of ensemble size on their estimates," *Earth System Dynamics*, vol. 12, no. 4, pp. 1427–1501, 2021. [Online]. Available: https://esd.copernicus.org/articles/12/1427/2021/

[130] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19317169

[131] H. Bhatia, F. Di Natale, J. Y. Moon, X. Zhang, J. R. Chavez, F. Aydin, C. Stanley, T. Oppelstrup, C. Neale, S. K. Schumacher, D. H. Ahn, S. Herbein, T. S. Carpenter, S. Gnanakaran, P.-T. Bremer, J. N. Glosli, F. C. Lightstone, and H. I. Ingolfsson, "Generalizable Coordination of Large Multiscale Workflows: Challenges and Learnings at Scale," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.

[132] R. C. J. Wills, Y. Dong, C. Proistosecu, K. C. Armour, and D. S. Battisti, "Systematic Climate Model Biases in the Large-Scale Patterns of

Recent Sea-Surface Temperature and Sea-Level Pressure Change," *Geophysical Research Letters*, vol. 49, no. 17, p. e2022GL100011, 2022, e2022GL100011 2022GL100011. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2022GL100011

[133] N. C. Thompson and S. Spanuth, "The Decline of Computers as a General Purpose Technology," *Commun. ACM*, vol. 64, no. 3, p. 64–72, Feb 2021. [Online]. Available: https://doi.org/10.1145/3430936

[134] T. K. Authors, "Kubernetes project dashboard," https://k8s.devstats.cncf.io/d/24/overall-project-statistics?orgId=1, accessed: 2024-8-5.

[135] "Project journey report," https://www.cncf.io/reports/kubernetes-project-journey-report/, June 2023, accessed: 2023-9-26.

[136] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17.

[137] C. Misale, D. J. Milroy, C. E. A. Gutierrez, M. Drocco, S. Herbein, D. H. Ahn, Z. Kaiser, and Y. Park, "Towards Standard Kubernetes Scheduling Interfaces for Converged Computing," in *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*, J. Nichols, A. B. Maccabe, J. Nutaro, S. Pophale, P. Devineni, T. Ahearn, and B. Verastegui, Eds. Cham: Springer International Publishing, 2022, pp. 310–326.

[138] C. Misale, M. Drocco, D. J. Milroy, C. E. A. Gutierrez, S. Herbein, D. H. Ahn, and Y. Park, "It's a Scheduling Affair: GROMACS in the Cloud with the KubeFlux Scheduler," in *2021 3rd International Workshop on Containers and*

New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), 2021, pp. 10–16.

[139] D. J. Milroy, C. Misale, G. Georgakoudis, T. Elengikal, A. Sarkar, M. Drocco, T. Patki, J.-S. Yeom, C. E. A. Gutierrez, D. H. Ahn, and Y. Park, "One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC," in *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2022, pp. 57–70.

[140] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17.

[141] The Kubernetes Authors, "Pods in kubernetes," https://kubernetes.io/docs/concepts/workloads/pods/, accessed: April 22, 2024.

[142] "Kubernetes: Cluster Autoscaler," https://github.com/kubernetes/autoscaler, 2023, online; accessed 29-August-2023.

[143] V. Sochat, A. Culquicondor, A. Ojea, and D. Milroy, "The Flux Operator [version 1; peer review: 2 approved] ," *F1000Research*, vol. 13, no. 203, 2024.

[144] https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/#scale-subresource, accessed: 05/09/2024.

[145] "Kube Flow-MPI Operator," https://github.com/kubeflow/mpi-operator, online: accessed 19-September-2023.

[146] "Google Cloud-The new way to cloud starts here," https://cloud.google.com/, accessed: 07/25/2023.

[147] "Microsoft Azure-Endless possibilities—from the edge to the cloud ," https://azure.microsoft.com/en-us, accessed: 07/25/2023.

[148] "IBM Cloud. Hybrid. Open. Resilient." https://www.ibm.com/cloud, accessed: 07/25/2023.

[149] None, "Lower the Time-to-Results for Tightly Coupled HPC Applications on the AWS Cloud with the Elastic Fabric Adapter," AWS, Intel, Tech. Rep., 2020. [Online]. Available: https://d1.awsstatic.com/HPC2019/Lower-Time-To-Results-wtih-EFA-Jan2020.pdf

[150] "Elastic Fabric Adapter in AWS," https://aws.amazon.com/hpc/efa/, last Accessed: 09/17/2023.

[151] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag, "A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC," *IEEE Micro*, vol. 40, no. 6, pp. 67–73, 2020.

[152] "Scale HPC Workloads with Elastic Fabric Adapter and AWS ParallelCluster," https://aws.amazon.com/blogs/opensource/scale-hpc-workloads-elastic-fabric-adapter-and-aws-parallelcluster/, accessed: 05/07/2024.

[153] "Placement Groups in AWS," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html, last Accessed: 09/17/2023.

[154] "AWS Global Infrastructure and Availability Zones," https://aws.amazon.com/about-aws/global-infrastructure/, accessed: 10/03/2023.

[155] D. Perez, L.-H. Hung, S. Xu, K. Y. Yeung, and W. Lloyd, "An Investigation on Public Cloud Performance Variation for an RNA Sequencing Workflow," in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388440.3414859

[156] Sochat and Hossen, "converged-computing/kubescaler: v0.0.15.1," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8306780

[157] "CORAL-2 Benchmarks, Advanced Simulation and Computing," https://asc.llnl.gov/coral-2-benchmarks, last Accessed: 09/18/2023.

[158] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[159] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002, developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168927401001155

[160] "Amg - algebraic multigrid solver codebase," https://github.com/LLNL/AMG2023, accesses: 05/02/2024.

[161] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - A MASSIVELY PARALLEL TRANSPORT MINI-APP," *OSTI*, 6 2015. [Online]. Available: https://www.osti.gov/biblio/1229802

[162] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012. [Online]. Available: https://doi.org/10.1137/120864672

[163] "Laghos," https://github.com/CEED/Laghos, accessed: 07/17/2024.

[164] "eksctl-The official CLI for Amazon EKS," https://eksctl.io/, accessed: 10/02/2023.

[165] "Boto3: AWS SDK for Python," https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, last Accessed: 9/18/2023.

[166] "AWS CloudFormation," https://aws.amazon.com/cloudformation/, last Accessed: 07/30/2024.

[167] M. R. Hossen, V. Sochat, A. Sarkar, M. Islam, and D. Milroy, "Repository for workload-driven autoscaling Cluster 2024," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13247408

[168] A. Raveendran, T. Bicer, and G. Agrawal, "A Framework for Elastic Execution of Existing MPI Programs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 940–947.

[169] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H.-J. Bungartz, "Infrastructure and API Extensions for Elastic Execution of MPI Applications," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 82–97. [Online]. Available: https://doi.org/10.1145/2966884.2966917

[170] I. Laguna, T. Gamblin, K. Mohror, M. Schulz, H. Pritchard, and N. Davis, "A Global Exception Fault Tolerance Model for MPI," 9 2014. [Online]. Available: https://www.osti.gov/biblio/1186781

[171] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 536–554.

[172] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013. [Online]. Available: https://doi.org/10.1177/1094342013488238

[173] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow-K8s: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 376–385.

[174] D. A. Monge and C. García Garino, "Adaptive Spot-Instances Aware Autoscaling for Scientific Workflows on the Cloud," in *High Performance Computing*, G. Hernández, C. J. Barrios Hernández, G. Díaz, C. García Garino, S. Nesmachnow, T. Pérez-Acle, M. Storti, and M. Vázquez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 13–27.

[175] "Kubeflow: The Machine Learning Toolkit for Kubernetes," https://www.kubeflow.org/, accessed: 10/02/2023.

[176] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 80–90.

[177] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 427–441. [Online]. Available: https://doi.org/10.1145/3472883.3486999

[178] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2019-July, pp. 329–338, 2019.

[179] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "Hansel: Adaptive horizontal scaling of microservices using bi-lstm," *Applied Soft Computing*, vol. 105, p.

107216, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494621001393

[180] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[181] M. R. Hossen, K. Ahmed, and M. A. Islam, "Market mechanism-based user-in-the-loop scalable power oversubscription for hpc systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 485–498.

[182] M. R. Hossen, "Pema+: A comprehensive resource manager for microservices," *SIGMETRICS Perform. Eval. Rev.*, vol. 51, no. 3, p. 10–12, jan 2024. [Online]. Available: https://doi.org/10.1145/3639830.3639836

[183] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "MLscale: A machine learning based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 287–299, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537917300835

[184] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," *Proceedings - International Conference on Distributed Computing Systems*, vol. 2019-July, pp. 122–132, 2019.

[185] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, 2022.

[186] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 46–49, 2021.

[187] AWS, Open Source Community, "Karpenter: Just-in-time Nodes for Any Kubernetes Cluster," https://karpenter.sh/, 2023, version-v0.29.2, Accessed-07/25/2023.