Malcolm: Multi-agent Learning for Cooperative Load Management at Rack Scale

ALI HOSSEIN ABBASI ABYANEH, University of Waterloo, Canada
MAIZI LIAO, University of Waterloo, Canada
SEYED MAJID ZAHEDI, University of Waterloo, Canada

We consider the problem of balancing the load among servers in dense racks for microsecond-scale workloads. To balance the load in such settings tens of millions of scheduling decisions have to be made per second. Achieving this throughput while providing microsecond-scale latency and high availability is extremely challenging. To address this challenge, we design a fully decentralized load-balancing framework. In this framework, servers collectively balance the load in the system. We model the interactions among servers as a cooperative stochastic game. To find the game's parametric Nash equilibrium, we design and implement a decentralized algorithm based on multi-agent-learning theory. We empirically show that our proposed algorithm is adaptive and scalable while outperforming state-of-the art alternatives. In homogeneous settings, Malcolm performs as well as the best alternative among other baselines. In heterogeneous settings, compared to other baselines, for lower loads, Malcolm improves tail latency by up to a factor of four. And for the same tail latency, Malcolm achieves up to 60% more throughput compared to the best alternative among other baselines.

CCS Concepts: • **Computing methodologies** → **Machine learning algorithms**; **Multi-agent systems**; **Cooperation and coordination**; • **Computer systems organization** → Heterogeneous (hybrid) systems; *Distributed architectures.*

Additional Key Words and Phrases: Distributed load balancing, task scheduling, cooperative game theory, multi-agent reinforcement learning, heterogeneous systems

ACM Reference Format:
Ali Hossein Abbasi Abyaneh, Maizi Liao, and Seyed Majid Zahedi. 2022. Malcolm: Multi-agent Learning for Cooperative Load Management at Rack Scale. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 3, Article 59 (December 2022), 25 pages. https://doi.org/10.1145/3570611

1 INTRODUCTION

To process user requests, popular datacenter applications such as web search, e-commerce, and social networks rely on responses from thousands of services. In such applications, end-to-end response times are dictated by the slowest response [19]. To guarantee fast responses, datacenter services are governed by strict service-level objectives (SLOs). To meet these SLOs, it is imperative to provide high throughput at microsecond-scale latency [10]. This is particularly important for tasks with service times in the range of several to tens of microseconds. Examples include in-memory key-value stores [1, 6, 20, 50, 96], NoSQL databases [2, 5], transactional databases [7, 87, 92], microservices [13], web-search ranking and sorting [11], and graph stores [43, 88]. For such tasks,

Authors' addresses: Ali Hossein Abbasi Abyaneh, University of Waterloo, Waterloo, Canada, ali.hossein.abbasi.abyaneh@uwaterloo.ca; Maizi Liao, University of Waterloo, Waterloo, Canada, m7liao@uwaterloo.ca; Seyed Majid Zahedi, University of Waterloo, Waterloo, Canada, smzahedi@uwaterloo.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2476-1249/2022/12-ART59 $15.00
https://doi.org/10.1145/3570611

datacenters systems are expected to support tail-latency SLOs that are a small multiple of task service times.

To optimize for killer microseconds, efficient queue management and task scheduling have become paramount [10]. There has been significant work on microsecond-scale schedulers for multi-core servers [12, 17, 25, 37, 63, 68, 70]. For example, ZygOS uses work stealing to reduce tail latency [68], and Shinjuku leverages hardware support for virtualization to implement microsecond-scale preemptive scheduling [37]. While these solutions achieve microsecond-scale tail latencies for multi-core servers, they do not scale beyond a few tens of cores.

A typical high-density datacenter rack can comprise thousands of interconnected, heterogeneous computing units. In a traditional rack, dense blade servers are connected together via one or two top-of-rack (ToR) switches. In the emerging rack-scale architectures, a *disaggregated* rack hosts a dense pool of compute, memory, and storage blades, all interconnected by a high-bandwidth network fabric. In such architectures, servers are replaced by racks as the unit of deployment in datacenters. Examples of rack-scale architecture include proposals from industry (Intel [35], Google [86], Microsoft [71], and HP [40]) and academia [9, 16, 39, 47, 62, 69, 77].

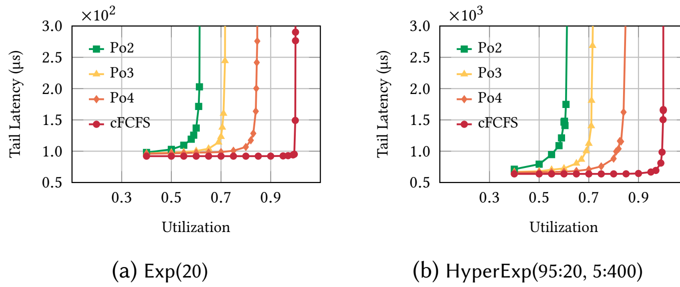
The increasing rack density poses new challenges for designing rack-scale schedulers. To address these challenges, in a recent work [102] the authors propose RackSched, a two-layer rack-scale scheduler. RackSched consists of a high-level inter-server scheduler and low-level intra-server schedulers. Each intra-server scheduler balances the load between cores in a server, and the inter-server scheduler balances the load between servers. To realize centralized rack-scale scheduling, RackSched implements the inter-server scheduler in programmable ToR switches. The key benefit of this approach is that the ToR switch can schedule tasks at the line rate as it already is on the path of all tasks sent to the rack.

Although RackSched achieves high scheduling throughput, the design has three main limitations. First, RackSched requires a programmable switch, which limits its deployment in datacenters without programmable switches. Second, RackSched imposes additional functionality to the packet switching fabric. Offloading computation to the switch data plane could ultimately lead to degraded network throughput [58]. Third, and more importantly, due to restricted computational and memory resources available in a programmable switch, RackSched uses power-of-2 to approximate cFCFS. While power-of-d-choices can balance the load in homogeneous systems, it can perform very poorly in the presence of heterogeneity [78, 100].

In this paper, we propose Malcolm, a dynamic, decentralized rack-scale load manager and task scheduler for microsecond-scale workloads. Malcolm is a heterogeneity-aware load-balancing framework that allows servers in the rack to collectively balance the load between themselves. We model the interactions among servers as a cooperative stochastic game, and use robust, game-theoretic analysis to study load-balancing strategies. Furthermore, to find the game's parametric Nash equilibrium, we design and implement a decentralized multi-agent learning algorithm. In our proposed solution, servers make scheduling decisions in tens of nanoseconds based on (possibly out-of-date) estimates of the load on other servers. Our implementation allows decentralized coordination among servers through infrequent network communications.

The key insights are: (a) load balancing at microsecond scale can be performed in a fully decentralized manner with infrequent communications using software-based solution; and (b) hand-crafted machine learning can be effectively exploited to find distributed load-balancing policies for microsecond-scale services. In summary, we make the following contributions.

- **Distributed load-balancing architecture §3.** We provide a distributed load-balancing architecture that enables independent servers to balance the load between themselves.



(a) Exp(20)

(b) HyperExp(95:20, 5:400)

Fig. 1. 99th-percentile latency under Po(D) vs. cFCFS in heterogeneous systems.

- **Distributed load-balancing game §4.** We model the interactions between independent servers as a potential Markov game and analyze Nash-equilibrium strategies in the game.
- **Distributed policy optimization §5–§6.** We design and implement a fully decentralized learning algorithm to find Nash-equilibrium policies.
- **Performance, scalability, and adaptivity §7.** We evaluate the performance, scalability, and adaptivity of Malcolm using rigorous experiments on our testbed. We show that Malcolm performs as well as the best alternative among other baselines for homogeneous racks. We further show that for heterogeneous racks, compared to other baselines, Malcolm improves tail latency under lower loads by up to a factor of four. And for the same tail latency, Malcolm achieves up to 60% more throughput compared to the best alternative among other baselines.

The code of Malcolm is open-source and available at https://github.com/uwaterloo-mast/malcolm.

2 BACKGROUND AND MOTIVATION

In recent years, there has been significant work on designing microsecond-scale scheduler for multi-core servers [12, 17, 25, 37, 63, 68, 70]. Today's servers often consist of tens to hundreds of core, and modern high-density racks deploy hundreds to thousands of (heterogeneous) cores. In practice, a single server-level task scheduler does not scale beyond eight to ten cores [37].

The increasing rack density in modern datacenters poses new challenges for designing rack-scale schedulers. In a rack with 1000 cores and an average service time of ten microseconds, the scheduler must handle, on average, 100 million tasks per second to fully utilize the rack. This means making one scheduling decision every ten nanoseconds. In addition to providing high scheduling throughput and low scheduling latency, a rack-scale scheduler has to guarantee high scheduling quality (i.e., supporting microsecond-scale tail latencies for each task). If tasks are simply scheduled to random servers, there will be temporal load imbalance between servers, which in turn causes long tail latencies for the entire system [102].

Centralized scheduling. Centralized first-come-first-serve (cFCFS) scheduling policy asymptotically minimizes tail latency for light-tailed service times [79]. A single core is capable of running a centralized scheduler for a server with eight to ten cores [37]. However, scheduling tasks for a rack with hundreds to thousands of cores far exceeds the capabilities of a general-purpose processor. To address this challenge, RackSched [102] proposes a two-layer hierarchical scheduler consisting of a high-level inter-server scheduler and low-level intra-server schedulers. For inter-server scheduling, RackSched implements power-of-d-choices policy in programmable ToR switches.

Power-of-d-choices policy (PoD). Power-of-d-choices policy approximates cFCFS. In PoD, for each incoming tasks, an scheduler randomly selects d servers and probes the length of their task queues. The scheduler then sends the task to the server with the shortest queue among probed

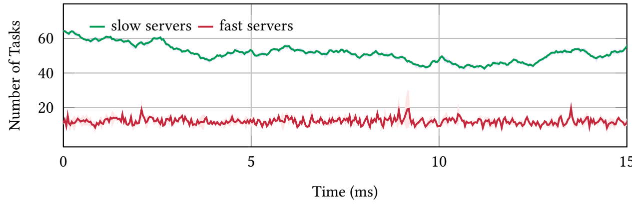


Fig. 2. Average server loads under Po4.

servers. While PoD can balance the load in homogeneous systems near-optimally, it can perform very poorly in the presence of heterogeneity [78, 100].

Heterogeneity in racks. Modern racks deploy increasingly heterogeneous hardware [45]. Heterogeneity has many manifestations. A rack could consist of servers with different processor types to improve efficiency by mixing off-the-shelf hardware [57]. Servers could distribute power unevenly across different processors to accelerate specific tasks [21, 99]. Processors could integrate heterogeneous cores to support diverse computational phases [36, 44, 55, 81].

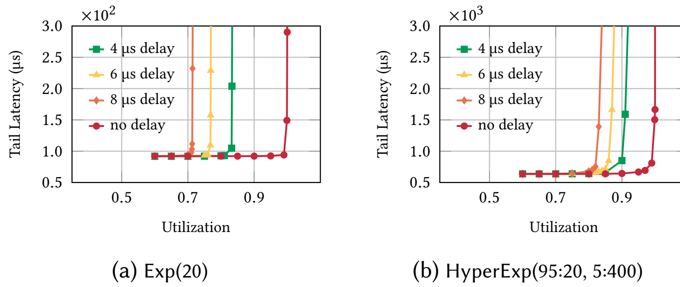
PoD in heterogeneous systems. To demonstrate suboptimality of PoD in heterogeneous systems, we use simulations on representative workloads. For the simulations, we assume two service-time distributions: (a) Exp(20) is an exponential distribution with mean 20 μs, representing low-dispersion workloads, and (b) HyperExp(95:20, 5:400) is a hyperexponential distribution with 95% of service times following Exp(20) and the other 5% following Exp(400), representing high-dispersion workloads. There are 16 servers, of which two are fast, and the rest are slow. Each fast server has 16 cores, while each slow server has only two. The intra-server scheduler for all servers is cFCFS.

Figure 1 compares ideal cFCFS against PoD for inter-server scheduling. The figure shows that power-of-2-choices policy fails to stabilize the 99th-percentile latency at loads as low as 65% for both workloads. The maximum sustainable load in terms of tail latency decreases as the number of queried servers decreases. This is mainly because PoD probes fast servers at the same rate as slow servers. As a result, the load is not balanced between fast and slow servers. This can be seen in Figure 2, which depicts total number of tasks (waiting and being served) in fast and slow servers over time at 85% load for Exp(20) workload under power-of-4-choices policy. The total number of tasks in fast servers is about 15, while on slow servers, it fluctuates between 40 to 60.

Centralized scheduling and network delay. Even if cFCFS could be implemented in a programmable switch, it is not clear whether cFCFS is optimal in terms of tail latency when network delays are a non-negligible fraction of service times. To realize cFCFS, tasks have to be queued at a centralized scheduler. Servers query the scheduler to fetch a new task every time they finish their current task. This takes a round-trip time (RTT) of at least a few microseconds[1], during which servers remain idle. This would be a noticeable overhead for workloads with average service times in the range of several to a few tens of microseconds. Figure 3 illustrates achievable 99th-percentile latencies by cFCFS under different RTTs for the two representative workloads. As network latency increases, the maximum sustainable utilization in terms of tail latency dramatically decreases.

Distributed scheduling. One alternative to centralized scheduling is distributed, client-based scheduling. In this approach, clients query servers and make scheduling decisions for each of their tasks. This approach has three major drawbacks. First, for every system reconfiguration, all clients have to be notified. This has a high system overhead when there are a large number of clients. Second, to minimize the overheads of probing, each client can only probe a fraction of servers. As

[1] Modern network stacks offer host-to-host RTT of about 4 μs [38].



(a) Exp(20)

(b) HyperExp(95:20, 5:400)

Fig. 3. 99th-percentile latency under cFCFS with network delay.

a result, clients may have to schedule tasks based on out-of-date load information, which could lead to low scheduling quality. Finally, client-based load balancing can lead to an undesirable race condition in which clients compete for service. When clients selfishly schedule their tasks to minimize their own tail latencies, the system becomes unstable at loads as low as 50% [26, 27].

Another alternative is distributed, server-based scheduling. In this approach, multiple dedicated dispatchers schedule tasks between servers in the rack. This approach addresses many drawbacks of the client-based solution. While there has been extensive work on distributed task scheduling [14, 64, 75], existing solutions do not meet the latency and throughput requirements of microsecond scale workloads. For instance, Sparrow [64] uses a combination of power-of-d-choices policy and late binding to schedule tasks. Although this approach works well for millisecond-scale workloads, as discussed earlier, delayed scheduling harms tail latency of microsecond-scale tasks.

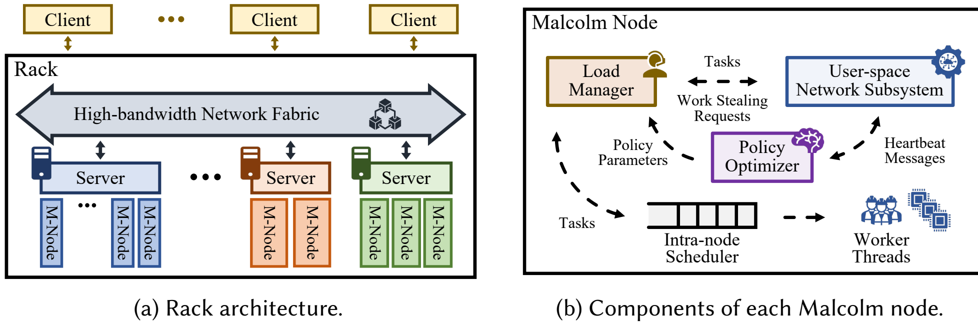
To bridge the gap between traditional task schedulers and modern microsecond-scale tasks, in this paper, we propose Malcolm, a distributed rack-scale scheduling framework. In Malcolm, each server runs one or more nodes. Malcolm nodes collaboratively balance the load among themselves. We model the interactions between Malcolm nodes as a team Markov game [90]. This allows for studying nodes' Nash equilibrium strategies. In an equilibrium, nodes cannot do better by unilaterally changing their strategy. We show that the game is a Markov potential game. This allows us to express the incentive of all nodes to change their load-balancing strategy using a single global function, called the potential function. Nash equilibria can be found in polynomial time by locating the local optima of the potential function

To find the game's (parametric) Nash equilibrium, we design and implement a decentralized algorithm based on multi-agent-learning theory. In the rest of this paper, we first present and analyze the game in §4. We then design (§5) and implement (§6) a decentralized algorithm to find Nash-equilibrium strategies. We finally present our empirical study on the performance, scalability, and adaptivity of Malcolm in §7.

3 MALCOLM ARCHITECTURE

We present the Malcolm framework for hierarchical, distributed task scheduling and load balancing at rack scale. Figure 4a illustrates an overview of the Malcolm architecture. Servers in the rack are interconnected by a high-bandwidth, low-latency network fabric. Servers can be heterogeneous with different computing capacities. Clients send their tasks to servers. Different servers can receive tasks at different rates.

Each server runs one or more Malcolm nodes. Like RackSched [102], Malcolm uses a multi-layer approach with intra-node and inter-node schedulers. Each Malcolm node consists of a centralized task scheduler for intra-node scheduling and a load manager for inter-node load balancing. Centralized intra-node schedulers schedule tasks between available worker threads within each Malcolm



(a) Rack architecture.

(b) Components of each Malcolm node.

Fig. 4. Overview of Malcolm design

node. Unlike RackSched, Malcolm adopts a distributed inter-node scheduling approach. Inter-node load managers cooperatively balance the load among Malcolm nodes at per-task granularity. The heartbeat manger is the core of each Malcolm node, which dynamically learns load-balancing strategies by interacting with other nodes. Nodes regularly send heartbeat messages to each other. Inter-node communications are done through a user-space network stack which bypasses the traditional TCP/IP stack. This enables nodes to communicate quickly with minimal delay. Figure 4b shows the main components of each Malcolm node.

3.1 Intra-node Scheduling

Each Malcolm node runs multiple worker threads to process tasks. Malcolm uses a centralized queue to buffer incoming tasks in each node. This has been shown to outperform per-worker queues for microsecond-scale workloads [37]. To schedule tasks between workers, there are two main policies: (a) first-come-first-served (FCFS) and (b) processor sharing (PS). Under FCFS, a worker finishes a task before starting a new one. Under PS, workers context switch between tasks to fairly divide processing capacity between all tasks.

Tail latency and service-time distribution. FCFS minimizes tail latency for light-tailed service-time distributions [79]. And PS minimize tail latency for heavy-tailed workloads [93]. Unfortunately, there is no static, work-conserving policy that minimizes tail latency for both workloads. There is an interesting dichotomy: policies that perform well under light-tailed workloads perform poorly under heavy-tailed workloads, and vice versa [93]. Among existing solutions, ZygOS [68] uses FCFS, while Shinjuku [37] implements PS. Malcolm is orthogonal to these works. The Malcolm design allows both solutions to be deployed. However, the default scheduler in Malcolm is FCFS.

3.2 Inter-node Load Balancing

Modern servers often consist of tens to hundreds of core. And modern high-density racks consist of hundreds to thousands of cores. However, a single intra-node scheduler does not scale beyond eight to ten cores [37]. To scale up, Malcolm adopts a hierarchical and distributed approach. Each server runs one or more Malcolm nodes. Malcolm nodes could be heterogeneous in terms of the number of worker threads. Nodes collaboratively balance the load among themselves.

Load balancing. Upon receiving a new task, the load manager decides whether to accept the task or migrate it to another node §4. This decision is made based on nodes' (possibly out-of-date) loads. The load manager can migrate incoming tasks to less loaded nodes when the local load is higher than the load on other nodes. Accepted tasks will be scheduled between worker threads by the intra-node scheduler. After processing each task by a worker thread, the load manager can decide if

it needs to steal tasks from other nodes. Work-stealing decisions on task completions complement the migration decisions on task arrivals.

Policy optimization. The load manager uses an adaptive policy to make migration and work-stealing decisions. This policy is periodically updated by the heartbeat manger based on the past decisions and current load differences (§5). To make these updates, heartbeat mangers communicate by sending heartbeat messages to one another. This allows heartbeat mangers to reach consensus on the rack-scale policy. The goal of the heartbeat manger is to minimize load imbalance among nodes with the minimum number of required migrations and work stealing requests.

Load estimation. In a homogeneous system, the load on a node can be captured by the length of the node's queue. The queue length, however, is not a useful metric in heterogeneous systems. Equal number of waiting tasks on node A and B does not mean that the two nodes have equal load if A is twice as fast as B. To account for heterogeneity, a more reliable metric is the queue length weighted by the inverse of service rate [76, 100]. This metric closely approximates the expected wait time of the last task in the queue [76]. To estimate service rate, each Malcolm node maintains a moving average of the inverse of task service times.

Instantaneous vs. average load. The load imbalance between two nodes is capture by the absolute difference between their loads. Temporal load imbalance among nodes results in higher tail latency for microsecond-scale workloads. Therefore, the main objective of the inter-node scheduler is to balance instantaneous loads over time. This is different from balancing long-term average loads. The former leads to the latter but not vice versa. Two nodes could have equal long-term average loads, while their instantaneous loads are different at any given time. While prior work has focused on balancing long-term average loads [30, 31, 67, 80, 85], Malcolm focuses on balancing instantaneous loads to minimize tail latency for microsecond-scale workloads.

4 THE DISTRIBUTED LOAD-BALANCING GAME

We present the distributed load-balancing game (DLB) that is used in Malcolm as a framework to balance the load between nodes at rack scale. Clients send tasks to nodes. Different nodes can receive and process tasks at different rates. Upon receiving a task, the load manager in each node decides whether to keep the task or migrate it to another node. Upon completing a task, the load manager decides if it needs to steal a task from another node. The state of the game evolves over time as scheduling decisions collectively shape the load on different nodes. The goal of the load manager is to minimize the load imbalance between nodes in the rack while migrating and stealing the minimum number of tasks.

4.1 Game Formulation

We model the DLB game as a team Markov game. Markov games generalize both Markov decision processes (MDPs) and repeated games. An MDP is a Markov game that is played by a single agent, and a repeated game is a Markov game with a single game state. The game consists of N heterogeneous nodes, represented by N agents. Time is divided into rounds[2]. For microsecond-scale workloads, the duration of each round could be tens to hundreds of microseconds. We assume that service times and inter-arrival times follow a fixed geometric distribution[3]. At each round, node i receives a new task with probability p_i and completes a task with probability q_i[4]. As we show in §7, Malcolm performs well for different service-time and inter-arrival-time distributions.

[2]For the ease of explanation, we present the game as a discrete-time game. Our analysis extends easily to continuous-time setting.

[3]In our experiments, we study performance of Malcolm for workloads with variety of other service-time distributions (see §7.1). We further study how Malcolm adapts to changes in service-time and inter-arrival-time distributions (see §7.4).

[4]In continuous-time setting, geometric distribution is replaced by exponential distribution.

States and actions. The load on each node represents the state of the node. The load on node i at round r is denoted by $x_{i,r}$. The state of the game at round r is $x_r = (x_{1,r}, \dots, x_{N,r})$. The state of the game evolves over time as agents take scheduling actions. We denote the set of scheduling actions taken by agent i at round r by $a_{i,r}$. For example, if node i accepts an incoming task at round r and steals another one from node j, then $a_{i,r} = \{accept, steal\ from\ j\}$. We use $a_r = (a_{1,r}, \dots, a_{N,r})$ to aggregate all actions taken by all nodes at round r.
Strategies. A strategy, π, provides a complete description of how an agent plays the game. Let $h_r = (x_0, a_0, x_1, a_1, \dots, x_r)$ denote the history of the game at round r. A deterministic strategy prescribes an action for all possible histories. To allow randomization, a *behavioral* strategy specifies a probability distribution over actions for any given history. In the DLB game, a behavioral strategy returns two probability distributions, one over migration and one over work-stealing actions.

The domain of deterministic and behavioral strategies is exponentially large as there are exponentially many different histories. To narrow the domain, we focus on a specific class of behavioral strategies called *stationary* strategies. A stationary strategy depends only on the final state of each history. This enables nodes to take scheduling actions based on the current system load and not the history of states and actions. Stationary strategies form a rich class of scheduling policies, which includes well-known policies such as power-of-d-choices, join-idle-queue, and join-below-threshold.
Utility. The utility function of each agent at each round is:

$$u(x_r, a_r) = -\sum_{i,j} (x_{i,r} - x_{j,r})^2 - C(a). \tag{1}$$

The utility function captures two costs: the cost of instantaneous load imbalance between all nodes and the cost migrations and work stealing requests. We use a linear function, C, to penalize each migration and work stealing request with a constant average cost. Let π_i denote the strategy of agent i, and let $\pi_{-i} = (\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_N)$ represent the strategy of all agents other than i[5]. The value function represents the long-term value of a state x for each agent i under strategy $\pi = (\pi_i, \pi_{-i})$, and it is defined as:

$$V_x^i(\pi) = \mathbb{E}\left[\sum_{r=0}^{\infty} \delta^r u(x_r, a_r) \mid a_r \sim \pi, x_0 = x\right].$$

This function captures the expected payoff in state x plus the expected discounted sum of future payoffs. Payoffs in the future are discounted because, all being equal, agents prefer performance sooner rather than later.

4.2 Nash Equilibrium

Agents optimize their scheduling strategies to maximize their expected long-term payoff. Agents would play their *best responses* if they knew exactly how other agents will play the game. Formally, agent i's best response to the strategy of others, π_{-i}, is a strategy π_i^* that satisfies $\pi_i^* = \text{argmax}_{\pi_i} V_x^i(\pi_i, \pi_{-i})$ for all states x. A *Nash equilibrium* (NE) is a strategy profile in which all agents simultaneously play best responses against each others strategies. Formally, an NE of a Markov game is a strategy profile π^* that satisfies $\pi_i^* = \text{argmax}_{\pi_i} V_x^i(\pi_i, \pi_{-i}^*)$ for all agents and states.
Equilibrium analysis. In general, the problem of finding an NE is computationally expensive. Theoretically, the complexity of computing a sample NE of a general-sum finite game with two or more agents is known to be *PPAD-complete* [18]. In practice, it is common belief that in the worst case, computing a sample NE takes time that is exponential in the size of the game. Fortunately, an

[5]Subscript $-i$ is used to refer to all agents other than agent i.

NE could be computed in polynomial time for the DLB game. This is because, as we show in the rest of this section, the DLB game is a *Markov potential game (MPG)*. And for an MPG, an NE can be obtained in polynomial time by solving a corresponding MDP [48, 56].

Definition 1. *A Markov game is said to be an MPG if there is a potential function, Φ, which satisfies the following condition for all agents i, states x, policies π_i, π_i', and π_{-i}.*

$$V_x^i(\pi_i, \pi_{-i}) - V_x^i(\pi_i', \pi_{-i}) = \Phi_x(\pi_i, \pi_{-i}) - \Phi_x(\pi_i', \pi_{-i}).$$

Informally, in an MPG, the incentives of all agents to change their strategies can be expressed in a single global function, called the potential function. For MPGs, an NE always exists, but it is not necessarily unique [48]. Team Markov games constitute a particular case of MPGs, with potential function being the common utility function (Corollary 1 in [56]). As a result, the problem of finding an NE in the DLB game is reducible to the optimal-control problem of finding a π that maximizes $\Phi_x(\pi)$ for all states x [48, 56, 97]. Although this optimal control problem can be solved in polynomial time, the order of the polynomial might be too large for any algorithms to be practical [52, 65].

Parametric strategies. To allow practical solutions, we focus on parametric strategies. Given a parameter vector w, a parametric strategy $\pi_w(x)$ maps states x to distributions over actions. With parametric policies, we could focus on *parametric Nash equilibrium* (PNE) strategies. Informally, a PNE is a projection of some NE onto a parametric class. The performance of a PNE approximates that of the projected NE. If parametric strategies are expressive enough, we can expect to achieve arbitrarily close performance to that of non-parametric solutions.

To find the in the DLB game, one option is to solve a centralized control problem:

$$\underset{w}{\text{maximize}} \;\; \Phi_x(\pi_w). \tag{2}$$

This, however, is not practical as it requires an accurate analytic model of system dynamics. Formulating system dynamics under different strategies for a rack with tens of nodes and thousands of cores is not tractable. Moreover, parameters of the system could change over time. The centralized optimal control problem should be resolved every time system dynamics change.

To address these challenges, in the next section, we propose a multi-agent-learning algorithm. The algorithm is model free and does not require any prior knowledge of system dynamics. The proposed decentralized algorithm solves the centralized control problem in a distributed manner and can be implemented and deployed in practice.

5 DECENTRALIZED POLICY OPTIMIZATION

In this section, we present a distributed policy optimization algorithm that is the core of Malcolm. The algorithm is based on multi-agent-learning theory, and it is guaranteed to find PNE strategies of the DLB game in a distributed manner.

5.1 Multi-agent Learning

Single-agent reinforcement learning. In single-agent reinforcement learning (RL), an agent learns through interactions with an environment. Model-free RL allows an agent to find the optimal policy without any prior knowledge of the system dynamics and payoff functions. At every round, r, the agent observes a state, x_r, takes an action, a_r, and receives a payoff, u_r. The state then transitions to x_{r+1}. The goal of the agent is to learn a policy that maximizes the expected long-term payoff, $J = \mathbb{E}[\sum_r \gamma^r u_r]$, where γ is the discount factor.

Q-Learning. Q-learning [91] is one of the most popular methods in RL. In Q-learning, agents learn the optimal policy indirectly by learning the optimal action-value function, $Q(x, a)$. The optimal Q

values can be learned recursively using the following Bellman equation:

$$Q(x_r, a_r) = (1 - \alpha)Q(x_r, a_r) + \alpha(u_r + \gamma \max_a Q(x_{r+1}, a)),$$

where α is the learning rate. The optimal action at a given state is the action that maximizes the Q value at that state.

Policy gradient. Policy gradient methods are another popular option in RL. The main idea is to directly learn the optimal parameters of the policy, w, to maximize the expected long-term payoff, $J(w)$, by taking steps in the direction of the gradient of the objective, $\nabla_w J(w)$, which can be derived using policy gradient theorem [83]:

$$\nabla_w J(w) = \mathbb{E}[Q^\pi(x, a)\nabla_w \log \pi_w(a, x)].$$

Policy gradient methods often differ with each other on how they compute Q^π. For example, REINFORCE [94], uses a simple unbiased Monte Carlo sampling to estimate the action-value function. An alternative approach is to use function approximation to directly learn a parametric approximation of the action-value function, Q^π_θ. This approximation is called the *critic*, and it leads to a variety of actor-critic methods [82].

From single-agent RL to multi-agent RL. The simplest way to apply RL to multi-agent settings is to let each agent learn independently using Q-learning methods or policy gradient algorithms [22, 84]. However, since agents adjust their policies independently, the environment could become non-stationary from each agent's point of view [22, 54]. Hence, single-agent RL is not guaranteed to find an optimal policy in a general multi-agent setting.

Centralized training, decentralized execution. To address the challenges of multi-agent learning, a popular approach is *centralized training with decentralized execution* (CTDE) [22, 23, 54]. In this approach, agents are trained in a centralized manner, but they execute their learned policies in a decentralized manner based on their local observations. A primary motivation behind this approach is that, during centralized learning, actions taken by all agents are known. This makes the environment stationary even as the policies change.

One way to implement CTDE is to train a centralized critic offline using a simulator. This method is not practically appealing, because a simulator might not be available, or the system might have time-varying dynamics. Another option is to implement a centralized controller that communicates with all agents to train a centralized critic. This method is also not desirable for two main reasons. First, it is not robust as the centralized controller becomes a single point of failure. Second, it is not scalable as it requires all agents to communicate with a single controller, making the controller a hotspot and causing long network delays. Moreover, the computational overhead of optimizing policy for all agents in a single controller puts policy optimization on the critical path.

Decentralized training, decentralized execution. To meet the requirements of an adaptive microsecond-scale scheduler, Malcolm adapts a decentralized approach inspired by recent advances in decentralized machine learning techniques [46, 98]. In particular, the load-balancing policies are trained separately in a distributed manner. First, learning an action-value function could be expensive in terms of computational, communication, and storage costs. To avoid these costs, policy optimizers directly learn a parametric value function, V^π_θ. Second, to train these parametric value functions, policy optimizers use the common utility function as the potential function.

To allow each policy optimizer to locally calculate the utility function, they regularly broadcast heartbeat messages. As we show in §7.4, these broadcasts could happen very infrequently as load-balancing strategies are tolerant to stale load information. In each heartbeat message, nodes include their load and the number of tasks they migrated or stole. Network delays or lost packets could cause individually learned value functions to drift away from one another. To address this

challenge, policy optimizers occasionally perform a consensus update: $\theta_i = (1/N) \sum_{j \in N} \theta_j$. After each consensus update, nodes reach an agreement on the global parametric value function.

The pseudocode of our proposed distributed policy-optimization algorithm is shown in Algorithm (1). Note that the steps for updating actor and critic parameters are based on temporal-difference learning [82]. Note further that policy gradient is guaranteed to converge to an NE for MPGs [48]. In the next section, we discuss the implementation of our algorithm to make it feasible for microsecond-scale deployment.

Algorithm 1: Distributed Policy Optimization

Input: α critic learning rate, β policies learning rate.
Randomly initialize $\theta_i, w_i;\ \forall i \in N$.
repeat ▹ Forever loop
  for *all* $i \in N$ **do** ▹ Decentralized execution
    take action according to $\pi_{w_i}(x_r)$
  observe actions, a_r, and new state, x_{r+1}
  for *all* $i \in N$ **do** ▹ Decentralized training
    compute global utility, u_r ▹ Equation (1)
    $\delta_r \leftarrow u_r + \gamma V_{\theta_i}(x_{r+1}) - V_{\theta_i}(x_r)$
    $\theta_i \leftarrow \theta_i + \alpha \cdot \delta_r \cdot \nabla_{\theta_i} V_{\theta_i}$ ▹ Critic
    $w_i \leftarrow w_i + \beta \cdot \delta_r \cdot \nabla_{w_i} \log \pi_{w_i}(a_r, x_r)$ ▹ Actor
  for *all* $i \in N$ **do** ▹ Decentralized consensus
    if $r \equiv 0 \pmod P$ **then**
      $\theta_i \leftarrow (1/N) \sum_{j \in N} \theta_j$

6 IMPLEMENTATION

In this section, we describe the details of Malcolm implementation. We first discuss each component of the Malcolm node (see Figure 4b). We then focus on the implementation details of updating policy parameters and maintaining migration and work-stealing probabilities.

6.1 Malcolm Node

User-space networking. To provide low-latency node-to-node communication, Malcolm uses eRPC [38], a general-purpose yet high-performance remote-procedure-call library. eRPC provides exceptional networking performance on lossy networks, implements congestion control, and handles packet losses. eRPC takes advantage of user-space networking stacks, such as DPDK [24] and RDMA [4].

Intra-node task scheduler. Malcolm's design allows recent dataplane operating systems and server-level schedulers such as ZygOS [68] and Shinjuku [37] to be deployed for scheduling tasks between worker threads. Malcolm is orthogonal to these works. However, for completeness, we implement a default task scheduler based on FCFS policy. Our implementation uses a centralized lock-free task queue to queue incoming tasks. Worker threads run in parallel on each core. They dequeue tasks from the centralized queue, execute them, and prepare responses. Responses are then sent back to clients by the node's *load manager* thread.

Load manager. Each Malcolm node has a dedicated load manager thread for inter-node load management and node-to-client communications. The load manager threads is responsible for handling incoming tasks (from clients and other nodes) and sending responses back to clients

Communication with local Malcolm nodes (nodes running on the same physical server) happens through shared memory with parallel, lock-free inbox queues. To communicate with remote nodes and clients, the load manager thread runs eRPC event loop.

On the arrival of each task, the load manager consults the migration policy to decide whether to accept the task or migrate it to another node. For a rack with n nodes, the migration policy is represented by an array of n elements with values that sum up to 1. For a given node i the value on the ith element of the array denotes the probability that node i accepts the task, while the value of each of the remaining $n - 1$ elements corresponds to the probability that node i sends the task to each of the other nodes.

Since migration policy is probabilistic, consulting the policy involves generating a random number uniformly from 0 to 1. Given a random number, load manager uses the cumulative distribution function (CDF) of the migration probabilities to identify the destination of the task (i.e., if it should be accepted or if it should be migrated to another server). Based on measurements on our testbed (see §7.1), this takes only tens of nanoseconds. If the task is accepted, the load manager pushes the task on the centralized lock-free task queue. Otherwise, the load manager migrates the task to the destination node. If the task queue is full, the accepted task fails and a response is sent to the client for a possible retry. Once each task is completed, and a response is ready to be sent, the load manager consults the work-stealing policy to decide whether to send a work-stealing request to other servers. Consulting the work-stealing policy is similar to consulting the migration policy.

Heartbeat manger. Malcolm dedicates a heartbeat manager thread to update policy parameters and maintain migration and work-stealing probabilities. The heartbeat manager also broadcasts heartbeat messages to other nodes at fixed *heartbeat intervals*. Each heartbeat message contains load information of the sender node in addition to the number of migrations and work-stealing requests initiated by the node. At the end each heartbeat interval, the heartbeat manager updates the migration and work-stealing probabilities based on policy parameters and latest load information. During each heartbeat interval, the migration and work-stealing probabilities remain the same. The length of the heartbeat period is a configurable parameter of Malcolm. As a default value, Malcolm sets this parameter to 100 microsecond.

Each heartbeat interval provides a new data point (i.e., the old state, taken actions, and the new state). Each Malcolm node collects data points to form a training dataset for updating policy parameters (i.e., actor and critic parameters). Malcolm uses a common technique in machine learning to split the training dataset into mini-batches to improve the quality of function approximations [29, 95]. Once a mini-batch of D data points is ready, the heartbeat manager updates policy parameters based on Algorithm (1). Between two updates, the policy parameters remain unchanged. The size of the mini-batch is a configurable parameter of Malcolm. As a default value, this parameter is set to 20.

The heartbeat manager is also responsible for running the consensus step according to Algorithm (1). This is done regularly after every P policy updates, which again, is a configurable parameter of Malcolm. As a default value, this parameter is set to 100.

6.2 Policy Optimization

Updating policy parameters. To enable parameter updates at the granularity of a few hundreds of microseconds, the execution time of Algorithm (1) has to be at most several microseconds. Unfortunately, implementing the algorithm using off-the-shelf machine-learning frameworks falls short of meeting this requirement. Libraries such as PyTorch [66] and Tensorflow [8] sacrifice performance for programmability. Among other techniques, these libraries often use automatic differentiation to compute gradient updates. This makes it easy for programmers to implement their

	Malcolm			PyTroch		
Number of nodes	16	32	64	16	32	64
Probability update	0.1	0.2	0.5	34	34	34
Parameter update	0.9	2	8	40	40	42

Table 1. Average execution time in microseconds.

models without worrying about gradient calculations. However, it comes at a great performance cost.

To achieve high performance, in Malcolm, for the critic's value function, we use linear function approximation as $V_\theta(x) = \langle \phi_c(x), \theta \rangle$, where ϕ_c is the critic's vector-valued basis function and θ is the critic's parameter vector. We implement variety of basis functions. For our experiments, we use a customized basis function that includes x_i, x_i^2, x_i^3, x_i^4, and $x_i x_j$ terms for $i, j \in \{1, \ldots, n\}$. For this customized basis function, the size of θ is $O(n^2)$.

For the actors' policy, we use softmax policy with linear function approximation as $\pi_w(x) = \sigma(\langle \phi_a(x), w \rangle)$, where ϕ_a is the actors' vector-valued basis function, w is the actors' parameter matrix[6], and σ is the softmax function. In our implementation, ϕ_a includes x_i and $\log(\text{rank of } x_i)$ terms for $i \in \{1, \ldots, n\}$, where rank of x_i is the index of x_i in the sorted array of x's. This allows the actors to learn policies that are based on the ranking of other nodes with respect to their loads. In a rack with n nodes, w matrix has $O(n^2)$ elements.

We derive closed-form formulas for gradient updates for the critic's and actors' parameters. We implement Algorithm (1) using these closed-form formulas in about 600 lines of C++ code. To speed up vector-to-vector multiplications, our implementation uses x86 AVX2 instructions, which are available in most datacenter servers. For comparison, we also implement Algorithm (1) using PyTorch C++ front-end [3]. Table 1 compares the execution time of updating policy parameters and calculating probabilities in Malcolm against the PyTorch implementation.

Caching gradients. Instead of calculating gradients for the entire mini-batch at once, we calculate and cache incremental gradients as new data points becomes available. Once the last data points for the mini-batch is available, we aggregate all cached incremental gradients to update the parameters using AdamW optimization algorithm [53]. This way, the cost of parameter updates is amortized over several smaller incremental gradient calculations.

7 EVALUATION

We use a diverse set of synthetic benchmarks to evaluate the performance of Malcolm on homogeneous and heterogeneous rack configurations. Furthermore, we evaluate the scalability and adaptability of Malcolm.

7.1 Experimental Methodology

Experimental environment. For our experiments, we use a heterogeneous cluster of seven servers. The cluster consists of two servers of type I, three servers of type II, one server of type III, and one server of type IV. Each type I server has 1TB DDR4 main memory and an AMD EPYC 7H12 processor with 64 physical cores running at 2.6 GHz. Each type II server has 16GB DDR4 main memory and an AMD Ryzen Threadripper PRO 3945WX processor with 12 physical cores running at 2.2 GHz. The type III server has 64GB DDR4 memory and an 8-core AMD EPYC 3201

[6]We maintain two matrices for work-stealing and migration policies.

Type	Instances	Processor	Memory
I	2	AMD EPYC 7H12, 64 cores, 2.6GHz	1TB DDR4
II	3	AMD Ryzen 3945WX, 12 cores, 2.2 GHz	16GB DDR4
III	1	AMD EPYC 3201, 8 cores, 1.5 GHz	64GB DDR4
IV	1	AMD Ryzen 1950X, 16 cores, 3.6 GHz	32GB DDR3

Table 2. Testbed server types.

processor operating at 1.5 GHz. Finally, the type IV server has 32GB DDR3 memory and a 16-core AMD Ryzen Threadripper 1950X Processor running at 3.6 GHz. Table 2 summarizes server types.

Type I and type IV servers are equipped with a dual-port 100Gb/s NVIDIA MCX556A ConnectX-5 VPI NIC. Type II and type III servers have a single-port 100Gb/s NVIDIA MCX555A ConnectX-5 VPI NIC. All NIC ports are configured to run in the InfiniBand mode. Servers are connected through an NVIDIA SB7800 switch.

Load generation. To ensure accurate tail-latency measurements at heavy load, clients are implemented as open-loop load generators [37, 49, 74, 102]. In all of the experiments, inter-arrival times are exponentially distributed. Each client starts connections with all load managers. Client threads send each generated task to a randomly selected load manager. Clients measure the end-to-end latency of tasks upon receiving their responses. For failed tasks, the latency is set to the max unsigned integer value.

Cluster configurations. We consider the following two cluster configurations.

- **Homogeneous.** In the homogeneous configuration, we deploy five nodes each with 10 worker threads on each type I server. The rest of servers run client threads.
- **Heterogeneous.** In the heterogeneous configuration, we deploy four 14-worker nodes on one of the type I servers and four 3-worker nodes on the other one. We further deploy two 3-worker nodes on each type II server. The rest of cores are used to run client threads.

All servers run Ubuntu LTS 20.04 distribution with kernel version 5.15. For each node, each thread (load manager, heartbeat manager, and workers) is pinned to a separate physical core.

Synthetic benchmarks. For synthetic benchmarks, we use the following four workloads.

- **Exp**(75) is an exponential distribution with mean equal to 75 μs. This benchmark represents single-type workloads (e.g., single-query data storage services).
- **Bimodal**(80:50, 20:250) is a multimodal distribution where 80% of tasks take 50 μs, and the remaining 20% take 500 μs. This benchmark represents multi-type workloads (e.g., *get* and *range* queries to key-value storage systems).
- **HyperExp-1**(50:50, 50:500) is a hyperexponential distribution where 50% of service times are sampled from Exp(50), and the remaining 50% are sampled from Exp(500). Hyperexponential distributions are popular choices in performance evaluation studies to model highly variable workloads [15, 72, 89]. Compared to Bimodal, this benchmark is more realistic as it replaces constant service times with exponentially distributed service times with different means.
- **HyperExp-2**(75:50, 20:500, 5:5000) is a hyperexponential distribution where 75% of service times follow Exp(50), 20% follow Exp(500), and 5% follow Exp(5000). This benchmark represents workloads with more diverse types (e.g., *get*, *range*, and *join* queries to a database).

Alternative baselines. To evaluate Malcolm, we compare it against alternative load balancing policies. In particular, we implement the following four policies.

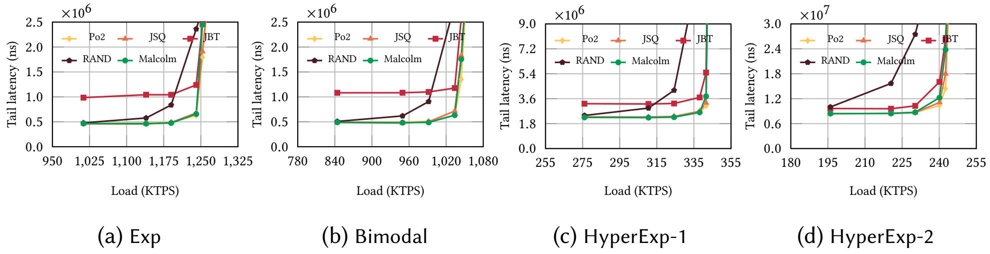


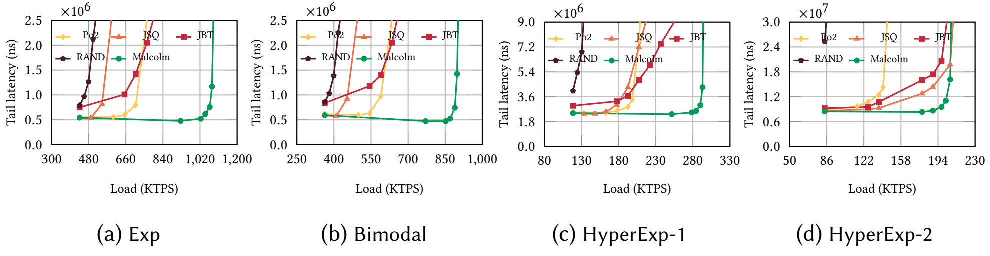
Fig. 5. 99th-percentile latency of synthetic workloads for homogeneous configuration.

- **Random (RAND)** Load managers accept all tasks that are sent to them as long as the task queue is not full. Since clients select load managers uniformly at random, this policy represents uniform distribution of tasks among nodes without any further load management among nodes.
- **Power-of-2 (Po2).** For each new task, load managers randomly select two nodes and send the task to the one with shorter queue length. Power-of-d-choices policy is a popular scheduling mechanism, and variants of it have been widely used in practice [60, 64, 102]. Due to its use of randomness, Po2 performs relatively well even when load managers have out-of-date load information about other nodes [59]. Power-of-d-choices policy performs near optimally in homogeneous settings, but it could perform very poorly in the presence of heterogeneity.
- **Join-shortest-queue (JSQ).** Load managers forward each new task to the node with the shortest queue length. R2P2 [42] and HovercRaft [41] use a variant of JSQ, called join-bounded-shortest-queue (JBSQ). In JBSQ, queues have bounded capacity, and if there is no empty slot in any queue, the task waits in the node's queue. One of the main drawbacks of JSQ and JBSQ is that several load managers could select the same node for their tasks, a concept commonly called *herd behavior.*
- **Join-below-threshold (JBT).** Load mangers forward each new task to a node whose queue length is below a fixed threshold. If no such node exists, the task is forwarded to a randomly selected node. Although centralized JBT is proved to be throughput optimal in heavy-traffic regimes, the optimal threshold is a function of the load on nodes and approaches infinity as load increases [78]. Moreover, similarly to JSQ, decentralized JBT suffers from herd behavior.

Load-propagation period. For Po2, JBT, and JSQ, each node always use its own up-to-date queue length. To propagate load information, nodes periodically broadcast their load to each other. Shorter periods lead to more up-to-date load information, and longer periods lead to out-of-date load information. Between two load broadcasts, in Po2, JBT, and JSQ, nodes update their local load information as they migrate tasks to each other. For instance, if node A migrates a task to node B with queue length of L, then node A updates its local estimation of node B's queue length to L + 1. For our experiments, we set the length of the load-propagation period to 100 µs which is the same as Malcolm's heartbeat interval. Later in this section, we study the sensitivity of different policies to the length of the load-propagation period.

7.2 Performance

We compare the performance of Malcolm against alternative load-balancing mechanisms in terms of 99th-percentile latency. For this, we use our synthetic workloads and consider the homogeneous and heterogeneous configurations.



(a) Exp (b) Bimodal (c) HyperExp-1 (d) HyperExp-2

Fig. 6. 99th-percentile latency of synthetic workloads for heterogeneous configuration.

Homogeneous configuration. We first consider the homogeneous configuration. Figure 5 compares Malcolm against other baselines in terms of the tail latency under different loads. For all benchmarks, Malcolm performs as good as the best among the other baselines. The results show that Malcolm can achieve low latency under up to maximum load of 93% for all benchmarks. For example, for Exp, Malcolm maintains low tail latency for up to 1245 KTPS[7]. Note that the maximum theoretical load for Exp is 1333.3 KTPS (= 100 × 13.3 KTPS).

Malcolm supports high task rates at lower tail latencies because it minimizes temporal load imbalance among nodes. Po2 and JSQ perform well as they balance load near optimally in homogeneous settings. RAND fails to maintain a low tail latency at higher task rates as it solely minimizes long-term load imbalance among nodes. JBT performs poorly as it uses a static threshold for load balancing, which is not guaranteed to work well for different workloads at different loads.

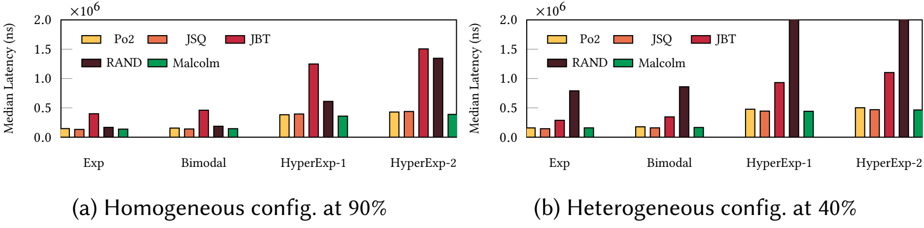
Heterogeneous configuration. Next, we consider the heterogeneous configuration. Figure 6 shows tail latency of workloads under different baselines as a function of load. For the heterogeneous configuration, Malcolm outperforms all the alternative baselines across all workloads. Compared to other baselines, for lower loads, Malcolm improves tail latency by up to a factor of four. And for the same tail latency, Malcolm achieves up to 60% more throughput compared to the best alternative among other baselines. Malcolm can again reach a maximum load of up to 93% at low tail latency for all workloads.

RAND perform very poorly for all the benchmarks. POWD also performs poorly, a behavior that is expected in a heterogeneous rack as discussed in §2. JSQ and JBT suffer from herd behavior for workloads with low average service time (Exp and Bimodal) as tasks arrive and depart at a higher rate than load information is propagated. Performance degradation is less for workloads with higher average service time because load information becomes more up-to-date for a fixed load-propagation period. This affects Malcolm at a much lesser extent as nodes in Malcolm coordinate their load-balancing strategies.

Figure 7 shows that average latencies for the homogeneous and the heterogeneous configurations at 90% and 40% utilization, respectively. The average latencies follow the same pattern as tail latencies. At high task rates, Malcolm maintains low average latency and 99th-percentile latency for all workloads in both of the homogeneous and the heterogeneous configurations.

7.2.1 Comparison Against RackSched. We compare against RackSched [102] using simulations[8]. We simulate the HyperExp-1 workload on the same homogeneous and heterogeneous configurations outlined in §7.1. For each experiment, similarly to the real-deployment experiments, we generate tasks in an open-loop manner. We report tail-latency results for the first 100K tasks that are created.

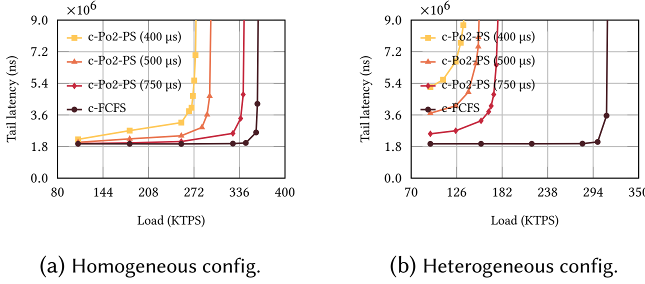
[7]KTPS is an abbreviation for 1000 tasks per second.
[8]We do not have access to a programmable switch.



(a) Homogeneous config. at 90%

(b) Heterogeneous config. at 40%

Fig. 7. Median latency under different baselines.



(a) Homogeneous config.

(b) Heterogeneous config.

Fig. 8. 99th-percentile latency of HyperExp-1 under c-Po2-PS (representing RackSched [102]) and c-FCFS.

This ensures that tail-latency results include latency of tasks with the same service-time distribution as intended in the workload.

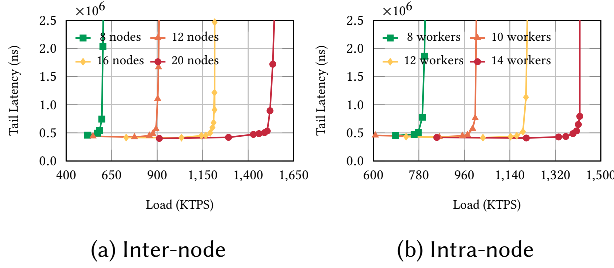
To represent RackSched, we implement a centralized power-of-2 scheduler with processor sharing policy for intra-node scheduling (c-Po2-PS). The network delay is assumed to be zero and nodes use load piggybacking. The preemption interval is set to 400 µs, 500 µs, and 750 µs. We also simulate ideal cFCFS as a reference for the theoretically optimal policy. The results for c-Po2-PS and cFCFS do not capture any network or system overheads.

Figure 8 shows the 99th-percentile latency of HyperExp-1 workloads under c-Po2-PS and c-FCFS for the homogeneous and heterogeneous configurations. The first key observation is that the tail latency under c-Po2-PS quickly goes up as the load increases for the heterogeneous configuration. This is mainly because power-of-2-choices policy performs poorly in the presence of heterogeneity. The second key observation is that PS hurts the tail latency of longer-running tasks, which constitute 50% of tasks in the HyperExp-1 workload. Decreasing the length of the preemption interval increases the tail latency as longer-running tasks are preempted more often.

For preemption interval of 500 µs, c-Po2-PS keeps tail latency low until 297 KTPS and 156 KTPS for the homogeneous configuration and the heterogeneous configuration, respectively. This is achieved under ideal settings without any network or system overheads. In comparison, as shown in Figure 5c and Figure 6c, in a real-world deployment, Malcolm achieves 14% and 85% higher throughput (up to 338 KTPS and 290 KTPS) compared to c-Po2-PS for the homogeneous configuration and the heterogeneous configuration, respectively. Malcolm keeps tail latency low for both configurations even at the heavy-traffic load by dynamically equalizing the load on all nodes.

7.3 Scalability Analysis

We conduct two experiments to measure the scalability of Malcolm and its implementation. First, we fix the number of worker threads per node to six and increase the number of nodes. Second, we



(a) Inter-node (b) Intra-node

Fig. 9. Scalability of Malcolm.

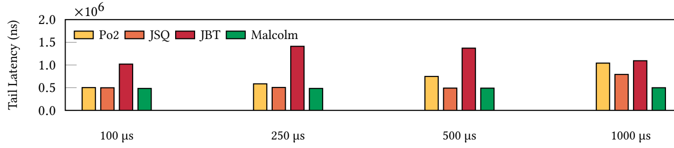


Fig. 10. 99th-percentile latency of Exp workload for different load-propagation periods.

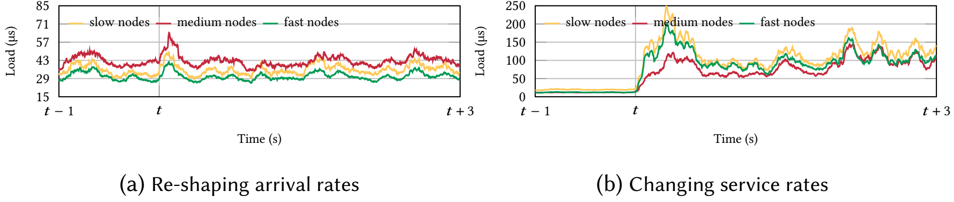
fix the number of nodes to 8 and increase the number of worker threads. For both experiments, clients generate the synthetic Exp workload. We measure the 99th-percentile latency of tasks.

Figure 9a shows the results for the first experiment. As can be seen, the throughput of Malcolm increases almost linearly as more nodes are added. For the Exp workload, with six worker threads, the theoretical maximum throughput for 8, 12, 16, and 20 nodes is 640, 960, 1280, and 1600 KTPS, respectively. For all configurations, the load can reach up to 93% at a tail latency that is only a few multiples of the average service time. Figure 9b illustrates the result of the second experiment. As can be seen again, the throughput of our user-space intra-node scheduler increases linearly up to 14 worker threads.

7.4 Adaptability

Sensitivity to load-propagation period. So far, in all experiments, we set the heartbeat interval to 100 µs. In this section, we study the performance of Malcolm under different interval lengths. We compare the results against Po2, JBT, and JSQ. We consider the homogeneous configuration and the Exp workload at 80% load. Figure 10 shows the measured 99th-percentile latency achieved by Malcolm, Po2, JBT, and JSQ for different heartbeat intervals. As can be seen, under Malcolm, the tail latency remains low for heartbeat intervals as long as 1 ms. The main reason for this is that the policy optimizers in Malcolm, in the process of learning cooperative load-balancing policy, implicitly learn the system dynamics for any fixed heartbeat interval. As a result, the learned policy automatically encodes load dynamics for different scheduling decisions at the given load-balancing frequency. This, however, is not the case for Po2, JBT and JSQ, as they make sub-optimal scheduling decisions with out-of-date load information as the length of the load-propagation period increases.

Sensitivity to fluctuations in load and service rate We study the adaptability of Malcolm when nodes' arrival rate or service rate changes. We consider three types of nodes – fast nodes with eight workers, medium nodes with six workers, and slow nodes with four workers. We deploy 12 nodes, four of each type. Service times follow Exp(75). We conduct two experiments. First, we fix the load at 90%. We start by equally dividing the traffic between all nodes. At time t, we change the shape of the traffic by sending 90% of all tasks to fast nodes. Figure 11a shows the load in terms of expected



(a) Re-shaping arrival rates

(b) Changing service rates

Fig. 11. Sensitivity to changes in arrival and service rates.

wait time on different node types before and after the change. It can be seen that the load on fast nodes increases. As fast nodes experience a higher load, they start migrating their tasks to the other nodes. This leads to an increase in the load on the slow and medium nodes. However, the system quickly adapts and learns to minimize load imbalance. Therefore, the load rapidly converges to its value prior to the change at time t.

Second, we fix the load at 80% and slow down the fast nodes by 50%. Figure 11b shows the load on different node types before and after the change. The load on fast nodes initially increases. Similarly to the first experiment, fast nodes migrate their tasks, which leads to an increase in the load on the slow and medium nodes. Malcolm quickly adapts and learns to minimize load imbalance between nodes. Unlike the first experiment, the overall load on all servers increases as the service rate on fast servers has increased after time t.

In both experiments, Malcolm learns to adapt to the changes in the system parameters. Additionally, Malcolm's immediate response to drastic changes does not make the system unstable. Note that average loads reflect the changes with some delay. This has three main reasons. First, we use the exponential moving average to track node loads. It is commonly known that moving averages reflect changes with some delay. The second reason is the delay associated with the nature of queuing systems. It takes some time for the task queues to reflect the changes in arrival and service rates. Finally, in Malcolm, the policy is updated in mini-batches at fixed intervals of 2 ms. As a result, Malcolm responds to change in system parameters with some delay.

8 DISCUSSION

Communication overhead. Each Malcolm node broadcasts heartbeat messages every p µs. In each heartbeat message, nodes include their load and the number of tasks they have migrated and stolen. Furthermore, every $C \times p$ µs, nodes broadcasting their value-function parameters. The default p and C in Malcolm are 100, and 2000. Assuming an m-byte representation of loads and weights, for a rack with n nodes, the one-way communication overhead is $o(mn/p + mn^3/2Cp)$ in terms of bytes/s. Considering a rack with 32 nodes, 4-byte floating-point representation, and 256 byte packet sizes, the default Malcolm configuration leads to less than 1.5 MB/s bandwidth consumption for each node.

Maintainability and hyper-parameter tuning. Malcolm nodes use a distributed, multi-agent reinforcement learning algorithm to find load-balancing strategies. The learning algorithm has a number of hyper-parameters. Different hyper-parameters impact performance differently. According to our experiments, the performance of Malcolm seems to be most impacted by the learning rates of the actors and the critic. We tune these parameters by searching over a limited range of values to maximize Malcolm's performance. In real-world deployment, as system parameters change, re-tuning of the hyper-parameters might become necessary.

Beyond single rack. Malcolm can be scaled beyond a single rack. In doing so, Malcolm can be deployed in a hierarchical manner. Each rack plays the role of a node in the load-distribution

game. And one node per rack plays the role of the load-balancer. Within each rack, nodes play the distributed load-balancing game together. Across racks, load-balancer nodes play the load-balancing game together. Design, analysis, and implantation of this hierarchical datacenter-scale scheduler are promising directions for future work.

Deployment. Other than rack-scale architectures, Malcolm can be deployed on dense racks in the traditional server-centric architectures. Moreover, Malcolm supports a variety of stateless and stateful workloads, such as micro-services, server-less tasks, in-memory key-value store, and high-throughput ML inferences. As shown in §7, Malcolm is able to achieve high throughput at low latency even when there are different types of tasks. Therefore, we expect Malcolm to perform well when each rack runs multiple different services.

Reconfiguration. If a node fails or a new node is added, other Malcolm nodes should learn a new load balancing policy. We have shown in §7 that Malcolm nodes learn to stabilize the system in less than a few hundreds of milliseconds. This fast convergence to the cooperative load-balancing strategy allows Malcolm nodes to quickly adapt to configuration changes. Implementation of policy reconfiguration is a future work.

9 RELATED WORKS

Centralized load balancing. Theoretical aspects of centralized load balancing have been extensively studied in the literature [32–34, 51, 60, 101]. Racksched [102] is a recent centralized rack-scale scheduler designed for microsecond-scale workloads. RackSched implements power-of-d-choices. Although simple, power-of-d-choices can perform very poorly in the presence of heterogeneity [28, 100]. Malcolm is a decentralized rack-scale scheduler for microsecond-scale workloads that performs well in homogeneous and heterogeneous systems.

Scheduling RPC requests. R2P2 [42] and Hovercraft [41] are two other recent works on load balancing. These works use join-bounded-shortest-queue (JBSQ). JBSQ is a variant of JSQ in which servers have bounded queues. JSQ and JBSQ use a poor load metric (i.e., length of task queue) and suffer from herd behavior. They are also sensitive to load-propagation frequency. Malcolm can be used to schedule RPC requests among servers in a rack. Malcolm implements an adaptive load-balancing algorithm that coordinates scheduling decisions between multiple load balancers. Malcolm performs well even for low load-propagation frequencies.

Decentralized load balancing. Distributed load balancing has also been widely studied [61, 64, 73, 102]. There have also been several attempts to incorporate game theory in distributed load balancing frameworks [30, 31, 67, 80]. While these works perform well for long-running tasks, they are not designed for microsecond-scale workloads. Malcolm is designed to balance the load in a distributed manner for tasks with microsecond-scale service times.

10 CONCLUSION

In this paper, we presented Malcolm, a distributed rack-scale scheduler designed for microsecond-scale services. We modeled interactions between nodes as a stochastic cooperative game. We proposed a decentralized learning algorithm to find load-balancing policies in this game. We empirically showed that our proposed algorithm is adaptive and scalable while outperforming state-of-the art alternatives. In homogeneous settings, Malcolm performs as well as the best alternative among other baselines. For lower loads, in heterogeneous settings, Malcolm improves tail latency by up to a factor of four compared to other baselines. And for the same tail latency, Malcolm achieves up to 60% more throughput compared to the best alternative among other baselines.

ACKNOWLEDGMENTS

This work was partially supported by the NSERC-RGPIN-2019-04936, NSERC-DGECR-2019-00475, CFI-JELF-38850, and ORF-RI-38850 grants.

REFERENCES

[1] 2022. Memcached key-value store. https://memcached.org/.
[2] 2022. MongoDB. https://www.mongodb.com/.
[3] 2022. PyTorch C++ API. https://pytorch.org/cppdocs.
[4] 2022. RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core/.
[5] 2022. Redis data structure store. https://redis.io/.
[6] 2022. RocksDB. https://rocksdb.org/.
[7] 2022. Volt Active Data. https://www.voltactivedata.com/.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and implementation (OSDI)*. 265–283.
[9] Krste Asanović. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. https://www.usenix.org/node/179918.
[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Communications of the ACM (CACM)* 60, 4 (mar 2017), 48–54.
[11] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE micro* 23, 2 (2003), 22–28.
[12] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2016), 1–39.
[13] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the "micro" back in microservice. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 645–650.
[14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 285–300.
[15] E. G. Coffman and R. C. Wood. 1966. Interarrival Statistics for Time Sharing Systems. *Communications of the ACM (CACM)* 9, 7 (1966), 500–503.
[16] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-Scale Computers. In *Proceedings of the Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM)*. 551–564.
[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-driven tail-aware balancing of μs-scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 35–48.
[18] Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. 2009. The complexity of computing a Nash equilibrium. *SIAM J. Comput.* 39, 1 (2009), 195–259.
[19] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Communications of the ACM (CACM)* 56, 2 (2013), 74–80.
[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.
[21] Mark E. Femal and Vincent W. Freeh. 2005. Boosting Data Center Performance Through Non-Uniform Power Allocation. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC)*. 250–261.
[22] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[23] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. 2016. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NeurIPS)*. 2145–2153.
[24] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org/.
[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. *Caladan: Mitigating Interference at Microsecond Timescales*. 281–297.
[26] Jason Gaitonde and Éva Tardos. 2020. Stability and learning in strategic queuing systems. In *Proceedings of the 21st ACM Conference on Economics and Computation, (EC)*. 319–347.
[27] Jason Gaitonde and Éva Tardos. 2021. Virtues of patience in strategic queuing systems. In *Proceedings of the 22nd ACM Conference on Economics and Computation (EC)*. 520–540.

[28] Kristen Gardner, Jazeem Abdul Jaleel, Alexander Wickeham, and Sherwin Doroudi. 2021. Scalable Load Balancing in the Presence of Heterogeneous Servers. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 37–38.
[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.
[30] Daniel Grosu and Anthony T Chronopoulos. 2005. Noncooperative load balancing in distributed systems. *J. Parallel and Distrib. Comput.* 65, 9 (2005), 1022–1034.
[31] Daniel Grosu, Anthony T Chronopoulos, and Ming-Ying Leung. 2002. Load balancing in distributed systems: An approach using cooperative games. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 10–pp.
[32] Tim Hellemans, Tejas Bodas, and Benny Van Houdt. 2019. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 3, 2 (2019), 1–35.
[33] Tim Hellemans and Benny Van Houdt. 2018. On the power-of-d-choices with least loaded server selection. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 2, 2 (2018), 1–22.
[34] Illés Antal Horváth, Ziv Scully, and Benny Van Houdt. 2019. Mean field analysis of join-below-threshold load balancing for resource sharing servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 3, 3 (2019), 1–21.
[35] Intel. 2022. Intel® Rack Scale Design (Intel® RSD). https://rb.gy/uxvjjt/.
[36] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. 186–197.
[37] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 345–360.
[38] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1–16.
[39] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 690–695.
[40] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. https://rb.gy/2xgd7j.
[41] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)*. 1–17.
[42] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 863–880.
[43] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2018. Splinter: Bare-metal Extensions for Multi-tenant Low-latency Storage. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 627–643.
[44] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 81–92.
[45] Benjamin C Lee. 2016. Datacenter design and management: A computer architect's perspective. *Synthesis Lectures on Computer Architecture* 11, 1 (2016), 1–121.
[46] Donghwan Lee, Niao He, Parameswaran Kamalaruban, and Volkan Cevher. 2020. Optimization for reinforcement learning: From a single agent to cooperative agents. *IEEE Signal Processing Magazine* 37, 3 (2020), 123–135.
[47] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. 2016. XFabric: A reconfigurable in-rack network for rack-scale computers. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 15–29.
[48] Stefanos Leonardos, Will Overman, Ioannis Panageas, and Georgios Piliouras. 2021. Global Convergence of Multi-Agent Policy Gradient in Markov Potential Games. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
[49] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 1–14.
[50] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444.
[51] Hwa-Chun Lin and Cauligi S Raghavendra. 1992. A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering* 18, 2 (1992), 148.

[52] Michael L Littman, Thomas L Dean, and Leslie Pack Kaelbling. 1995. On the complexity of solving Markov decision problems. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*. 394–402.
[53] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
[54] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*. 6382–6393.
[55] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. 2012. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 45th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 317–328.
[56] Sergio Valcarcel Macua, Javier Zazo, and Santiago Zazo. 2018. Learning Parametric Closed-Loop Policies for Markov Potential Games. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
[57] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 619–630.
[58] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. 2019. Thoughts on Load Distribution and the Role of Programmable Switches. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 18–23.
[59] Michael Mitzenmacher. 1997. How useful is old information?. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 83–91.
[60] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 12, 10 (2001), 1094–1104.
[61] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*. IEEE, 137–148.
[62] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 3–17.
[63] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 361–377.
[64] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 69–84.
[65] Christos H Papadimitriou and John N Tsitsiklis. 1987. The complexity of Markov decision processes. *Mathematics of Operations Research* 12, 3 (1987), 441–450.
[66] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. (2019), 8026–8037.
[67] Satish Penmatsa and Anthony T Chronopoulos. 2011. Game-theoretic static load balancing for distributed systems. *J. Parallel and Distrib. Comput.* 71, 4 (2011), 537–555.
[68] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, (SOSP)*. 325–341.
[69] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 13–24.
[70] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-aware thread management. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 145–160.
[71] Microsoft Research. 2013. Rack-scale Computing. https://rb.gy/ps9fzo.
[72] Robert F. Rosin. 1965. Determining a Computing Center Environment. *Communications of the ACM (CACM)* 8, 7 (1965), 463–468.
[73] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 410–424.
[74] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus Closed: A Cautionary Tale. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 18.

[75] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 351–364.

[76] Jori Selen, Ivo Adan, Stella Kapodistria, and Johan van Leeuwaarden. 2016. Steady-state analysis of shortest expected delay routing. *Queueing Systems* 84, 3 (2016), 309–354.

[77] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 255–270.

[78] Alexander L Stolyar. 2015. Pull-based load distribution in large-scale heterogeneous service systems. *Queueing Systems* 80, 4 (2015), 341–361.

[79] Alexander L Stolyar and Kavita Ramanan. 2001. Largest weighted delay first scheduling: Large deviations and optimality. *Annals of Applied Probability* (2001), 1–48.

[80] Riky Subrata, Albert Y Zomaya, and Bjorn Landfeldt. 2007. Game-theoretic approach for load balancing in computational grids. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 19, 1 (2007), 66–76.

[81] M Aater Suleman, Milad Hashemi, Chris Wilkerson, Yale N Patt, et al. 2012. Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 305–316.

[82] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[83] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems (NeurIPS)*. 1057–1063.

[84] Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the 10th International Conference on Machine Learning (ICML)*. 330–337.

[85] Xueyan Tang and Samuel T Chanson. 2000. Optimizing static job scheduling in a network of heterogeneous computers. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE, 373–382.

[86] Paul Teich. 2017. Under The Hood Of Googles TPU2 Machine Learning Clusters. https://rb.gy/3xmprc.

[87] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 18–32.

[88] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, et al. 2012. TAO: How Facebook serves the social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 791–792.

[89] Edward S. Walter and Victor L. Wallace. 1967. Further Analysis of a Computing Center Environment. *Communications of the ACM (CACM)* 10, 5 (1967), 266–272.

[90] Xiaofeng Wang and Tuomas Sandholm. 2002. Reinforcement learning to play an optimal Nash equilibrium in team Markov games. In *Proceedings of the 15th International Conference on Neural Information Processing Systems (NeurIPS)*. 1603–1610.

[91] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3-4 (1992), 279–292.

[92] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 87–104.

[93] Adam Wierman and Bert Zwart. 2012. Is tail-optimal scheduling possible? *Operations Research* 60, 5 (2012), 1249–1257.

[94] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (1992), 229–256.

[95] D Randall Wilson and Tony R Martinez. 2003. The general inefficiency of batch training for gradient descent learning. *Neural Networks* 16, 10 (2003), 1429–1451.

[96] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. 167–180.

[97] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control* (2021), 321–384.

[98] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. 2018. Fully decentralized multi-agent reinforcement learning with networked agents. In *Proceedings of the International Conference on Machine Learning (ICML)*. 5872–5881.

[99] Wenli Zheng and Xiaorui Wang. 2015. Data center sprinting: Enabling computational sprinting at the data center level. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)*. 175–184.

[100] Xingyu Zhou, Ness Shroff, and Adam Wierman. 2021. Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers. *Performance Evaluation* 145 (2021), 102146.

[101] Xingyu Zhou, Fei Wu, Jian Tan, Kannan Srinivasan, and Ness Shroff. 2018. Degree of queue imbalance: Overcoming the limitation of heavy-traffic delay optimality in load balancing systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 2, 1 (2018), 1–41.

[102] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A microsecond-scale scheduler for rack-scale computers. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1225–1240.

Received August 2022; revised October 2022; accepted November 2022