Multi-agent Learning for Cooperative Scheduling of Microsecond-scale Services at Rack Scale

by

Ali Hossein Abbasi Abyaneh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Abstract

This work considers the load-balancing problem in dense racks running microsecond-scale services. In such a system, balancing the load among hundreds to thousands of cores requires making millions of scheduling decisions per second. Achieving this throughput while providing microsecond-scale tail latency and high availability is extremely challenging. To address this challenge, we design a fully distributed load-balancing framework. In this framework, servers cooperatively balance the load in the system. We model the interactions among servers as a cooperative stochastic game. In this game, servers make scheduling decisions upon receiving and completing tasks. When a server receives a task, it decides whether to keep the task or migrate the task to another server. Moreover, when a server completes a task, it decides if it needs to steal a task from another server. We propose a distributed multi-agent learning algorithm to find the game's parametric Nash equilibrium. Our proposed algorithm enables servers to make scheduling decisions in tens of nanoseconds based on (possibly outdated) estimates of the load on other servers. We implement and deploy our distributed load-balancing algorithm on a rack-scale computer with 264 physical cores. We compare our load balancing algorithm with state-of-the-art load balancing disciplines. Our proposed solution provides up to 20% more throughput at low tail latency than widely used load balancing policies.

Acknowledgements

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to thank my advisor, Professor Seyed Majid Zahedi, without whose support, dedication, and expertise this thesis would not have been possible. I would also like to thank my parents, my family, and my friends for their support and love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's datacenter applications such as web search, machine translation, and e-commerce fan out requests to hundreds of software services. Hence, end-to-end response times are determined by the slowest response from each of these services [19]. These applications have strict service-level objectives (SLO) [8], often defined in terms of end-to-end tail latency [85]. To meet stringent user-facing SLOs, datacenters services must provide high throughput at tail latencies as low as a few hundred microseconds [8].

Optimizing latency of microsecond-scale services demands efficient queue management and task scheduling techniques. To this end, there have been considerable works on data plane operating systems for multi-core machines [10, 94, 96, 87, 51, 33]. These operating systems provide low overhead services to microsecond-scale latency-sensitive applications. For instance, Shinjuku [94] exploits hardware virtualization techniques to implement low overhead processor sharing, and ZygOS implements low overhead work-stealing to improve distributed First-Come-First-Serve. Unfortunately, these methods do not scale beyond a single multi-core server.

Modern datacenter racks consists of up to thousands of processing units. Scheduling microsecond-scale tasks in such a system is challenging for several reasons. First, the scheduler must schedule millions of tasks per second with low scheduling latency. Second, scheduling decisions should not lead to load imbalance throughout the rack in that load imbalance results in long-tail latencies [17, 12, 131]. Third, the scheduler should consider heterogeneity and adapt to variations both in tasks and rack resources. Existing solutions suffer from high scheduling latency and low scheduling throughput [88, 103, 13, 23, 40, 92], rely on randomized scheduling policies which are suboptimal in heterogeneous systems [36, 80, 135], or are oblivious to fluctuations in computational resources [132].

To the best of our knowledge, RackSched [135] is the latest rack-scale scheduler designed for microsecond-scale services. RackSched implements a centralized scheduler inside the Top of Rack (ToR) switch. This poses additional functionality to the switch and possibly degrades the networking throughput [76]. Moreover, since switch has limited computational resources, RackSched implements power-of-2-choices, which could be inefficient in heterogenous systems [132, 106].

This thesis presents Malcolm, a dynamic distributed load balancer designed for microsecond-scale workloads. Malcolm is a heterogeneity-aware distributed load balancer that leverages multi-agent systems to make optimal scheduling decisions for millions of tasks per second with tens of nanoseconds overhead.

Malcolm takes advantage of a stochastic game in which servers are the players, their actions are the scheduling decisions they make, and they are rewarded based on the quality of these decisions. In this game, servers collaboratively minimize the temporal load imbalance throughout the rack with minimum communication overhead. Servers are allowed to migrate incoming tasks to other servers in the rack and steal tasks from each other. To make optimal migration and stealing decisions, servers periodically broadcast their load to one another. Hence, servers make decisions based on possibly outdated load information. Moreover, individual payoff functions are designed to capture the game's global objective: load balancing the rack with minimum networking overhead.

Our 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) handcrafted machine learning can be effectively exploited to find optimal distributed load balancing policies for microsecond-scale services.

In summary, we make the following contributions.

- **Distributed load-balancing architecture §3.** We provide the architecture of our the distributed load-balancing framework. This architecture enables independent servers to balance the load between themselves.
- **Distributed load-balancing game §4.** We model the interactions between independent servers as a Markov potential game and analyze Markov Nash equilibrium strategies in the game.
- **Distributed policy optimization §5–§6.** We design and implement a fully decentralized multi agent learning algorithm to find optimal solution of the game.
- **Implementation §6.** We implement a prototype of our load balancing framework. We also perform a set of implementation level optimizations to enable multi-agent learning at microsecond-scale deployment.

- **Performance, scalability, and adaptivity §7.** We evaluate the performance, scalability, and adaptivity of Malcolm using a variety of experiments on a rack-scale computer. In our experiments, Malcolm can provide up to 20 % more throughput at low tail latency compared to state-of-the-art load-balancing policies.

Chapter 2

Background and motivation

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 [8]. This is particularly important for tasks with service times in the range of several to tens of microseconds. For such tasks, datacenters systems are expected to support tail-latency SLOs that are a small multiple of task service times.

To optimize the latency of microsecond-scale workloads, efficient queue management and task scheduling have become paramount [8]. There has been significant work on microsecond-scale schedulers for multi-core servers [10, 94, 96, 87, 17, 51, 33]. For example, ZygOS uses work stealing to reduce tail latency [94], and Shinjuku leverages hardware support for virtualization to implement microsecond-scale preemptive scheduling [51]. While these solutions achieve microsecond-scale tail latencies for a single machine, they cannot schedule tasks across multiple machines.

2.1 Rack-scale architecture

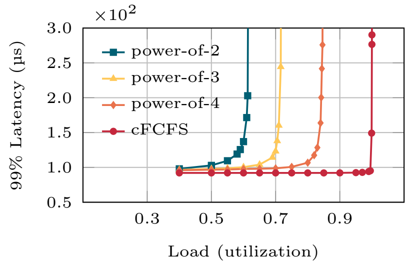
A typical high-density datacenter rack can comprise thousands of interconnected, heterogeneous computing units. In a traditional rack, dense blade servers are connected together via one or two top-of-rack (ToR) switches. In the emerging rack-scale architectures, a *disaggregated* rack hosts a dense pool of compute, memory, and storage blades, all interconnected by a high-bandwidth network fabric. In such architectures, servers are replaced by racks as the basic building blocks of datacenters. Examples of rack-scale architecture include proposals from industry (Intel [1], Google [4], Microsoft [2], and HP [55]) and academia [7, 95, 86, 16, 53, 61, 105].

The increasing rack density poses new challenges for designing rack-scale schedulers. In a rack with 1000 cores and average service time of twenty microseconds, the scheduler must handle, on average, 50 million tasks per second to fully utilize the rack. Hence, the scheduler needs to make one scheduling decision every twenty 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 [135].

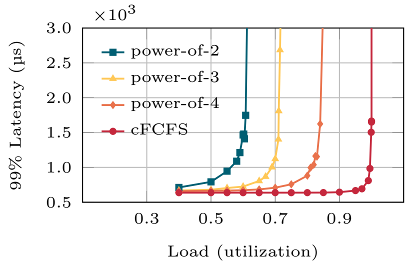
2.2 Centralized scheduling

Centralized first-come-first-serve (cFCFS) scheduling policy minimizes tail latency of workloads with light-tail distribution[107]. In a multi-core machine, a single core is capable of running a centralized scheduler. However, the requirements for centralized rack-scale scheduler exceed the capabilities of a general-purpose processor. To address this challenge, RackSched [135] proposes a two-layer hierarchical scheduler. This two-layer scheduler consists of a high-level inter-server scheduler and low-level intra-server schedulers. Each intra-server schedules requests on cores in a single server. The inter-server scheduler balances the load between servers. To accomplish centralized rack-scale scheduling, RackSched implements the inter-server scheduler inside the ToR programmable switch. The key benefit of this approach is that the ToR switch is on the path of all tasks sent to the rack. Hence, the ToR switch can schedule tasks at line rate.

RackSched design suffers from three main limitations. First, RackSched requires programmable switch, which limits its deployment in racks without programmable switches. Second, RackSched imposes additional functionality to ToR switch. Offloading computation to the switch data plane could ultimately degrade network throughput [76]. Third, due to restricted computational and memory resources available in a programmable switch, RackSched cannot implement cFCFS. Consequently, authors in [135] implement power-of-d-choices to approximate cFCFS. In power-of-d-choices, the inter-server scheduler queries d random servers for each arriving task. The scheduler then sends the task to the server with the lowest load among the d queried servers. The default d in RackSched implementation is 2.



(a) Exponential



(b) Bimodal

Figure 2.1: Power-of-d vs. cFCFS in heterogeneous systems.

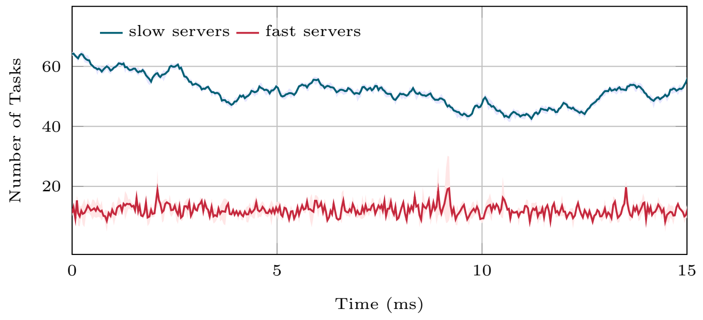


Figure 2.2: Average server loads under power-of-4

2.2.1 Power-of-d-choices in heterogeneous systems

Power-of-d choices is delay optimal in heavy traffic [70]. However, in general, power-of-d is unstable in heterogeneous systems [132, 106]. Moreover, being delay optimal in the heavy traffic regime [1] is a coarse metric. Empirically, load balancing policies that are heavy-traffic delay optimal can perform very poorly [134]. To demonstrate this, we use simulations on representative workloads to evaluate power-of-d-choices in a heterogeneous system. For the simulations, we use two service time distributions: (a) exponential distribution with mean 20 µs (Exp(20)), which represents low-dispersion workloads, and (b) bimodal distribution with 95% of service times follow Exp(20), and the other 5% follow Exp(400), which represents high-dispersion workloads. There are two fast servers and fourteen slow servers. Each fast server has 16 workers, and each slow server has two workers. The intra-server scheduler for all servers is cFCFS.

Figure 2.1 compares ideal cFCFS against power-of-d-choices for inter-server scheduling. The figure shows that power-of-2-choices fails to stabilize the system at loads as low 65%

[1]For a formal definition, look at appendix B.

for both workloads. The maximum sustainable load decreases as the number of queried servers decreases. This is mainly because power-of-d-choices cannot balance the load among heterogeneous servers as the scheduler probes fast servers at the same rate as slow servers. This can be seen in Figure 2.2, 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. The total number of tasks in the fast servers is close to 15, while the load on slow servers fluctuates between 40 to 60 tasks.

2.2.2 Join-Shortest-Queue (JSQ)

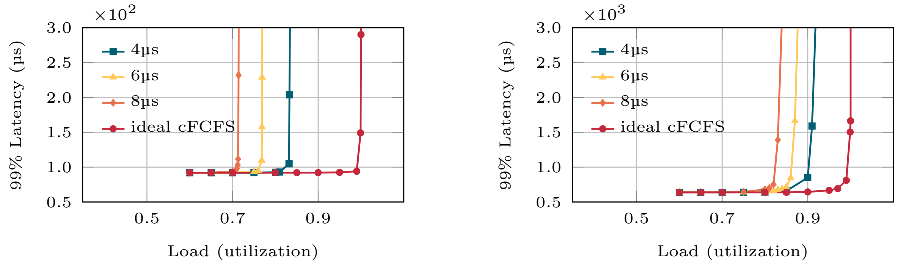
JSQ is a bufferless scheduler in which tasks are immediately sent to the server with the shortest queue length. Although this approach eliminates the queue inside the scheduler, it has several drawbacks. First, queue length is not a competent indicator of load on servers [134, 88], specially when tasks are heterogeneous. Second, upon every arrival, the scheduler needs to probe the load on all servers. Therefore, compared to cFCFS, JSQ introduces additional network overhead. This can be mitigated by probing the load on servers less frequently. However, when load information are delayed, always sending to the server with shortest queue length is not optimal [122, 132].

R2P2 and Hovercraft [58, 57] use Join-Bounded-Shortest-Queue (JBSQ) to load balance microsecond-scale remote procedure calls. JBSQ is a variant of JSQ in which servers have bounded queues. In JBSQ, among the queues with empty slots, requests are routed to the one with the shortest queue length. If there is no such a queue, R2P2 delays the assignment of the requests and stores them inside a centralized queue. This centralized queue can be implemented in either software or a programmable switch.

Compared to JSQ, JBSQ offers lower tail latency for two reasons. First, queue length in JSQ is not a fine indicator of the waiting time. Delayed assignment of tasks mitigates this inaccuracy [88]. However, bounding the queue length only alleviates the shortcomings of JSQ. Moreover, finding the optimal value for the maximum queue size is an unanswered question.

2.2.3 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, the centralized scheduler must queue all incoming tasks and has to be notified every time a server finishes a task. This takes a round-trip time



(a) Exponential (b) Bimodal

Figure 2.3: cFCFS with network delay.

(RTT) of at least few microseconds[2], during which the server remains 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 2.3 illustrates achievable tail latencies by cFCFS under different RTTs for the two representative workloads. As network latency increases, the maximum sustainable utilization of the system dramatically decreases.

2.3 Distributed scheduling

One natural solution for solving the scalability limits of centralized scheduling is using multiple distributed schedulers. Distributed scheduling (a) offers appealing scalability features (b) and provides high-availability.

2.3.1 Client-based 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 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 a result, clients may have to schedule tasks based on out-of-date load information, leading 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% [34, 35].

[2]State-of-the-art datacenter networking stacks offer host-to-host RTT of about 4 µs [52].

2.3.2 Distributed Dispatchers

Another alternative is distributed, server-based scheduling. In this approach, multiple dedicated servers schedule tasks in the cluster. This approach addresses many drawbacks of the client-based solution.

State-of-the-art schedulers designed for second and millisecond scale tasks, like Sparrow [88], Omega [103], and Apollo [13] follow the same approach and distribute scheduling among multiple servers. However, these distributed schedulers do not meet the latency and throughput requirements of microsecond scale workloads in rack-scale computers.

For instance, to make up for the poor performance of load metric in power-of-d, that is, queue length, Sparrow [88] schedules requests using late binding. Late binding is a variant of power-of-d in which the scheduler delays scheduling until one out of d selected servers becomes available to run the task. Although this approach works well with millisecond scale workloads, as we discussed earlier, delayed scheduling harms the latency of microsecond scale services.

Apollo exploits a distributed estimation-based scheduler for scheduling heterogeneous workloads ranging from millisecond to a few hundred seconds [13]. Apollo schedulers schedule each task on a server which minimizes the estimated completion time of that task. To estimate task completion times, schedulers need to have an up-to-date view of the system, that is, resource availability and load on servers. Considering cluster throughput for microsecond scale workloads, this comes with a considerable system overhead.

One simple solution for building a distributed scheduler is replicating a static centralized load balancing policy. For example, both JSQ and power-of-d-choices can be deployed in multiple dispatchers. Replicating JSQ and power-of-d-choices, however, suffers from a major drawback: herd behavior [132, 81]. This means that multiple schedulers decide that a single server is the optimal destination of their tasks which overloads that server. This will lead to a poor delay performance [109].

To deal with this issue, authors in [109, 132] propose for dispatchers to keep local estimates of queue lengths for each server. However, estimating the queue length of each server when tasks are heterogeneous and system configurations are changing is challenging.

2.3.3 Malcolm

To fill the gap between traditional task schedulers and modern microsecond-scale tasks, in this thesis, we propose an adaptive, distributed scheduling framework for rack-scale computers. In our proposed framework, all servers in the rack collectively balance the load between themselves. We model the interactions among servers as a cooperative stochastic game, and use robust, game-theoretic analysis to provide qualitative performance guarantees. Furthermore, to find the game's parametric Nash equilibrium, we design and implement a distributed 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.

Chapter 3

Malcolm Architecture

In this chapter, we present Malcolm architecture. Malcolm is a distributed, hierarchical rack-scale scheduler. Malcolm consists of distributed schedulers for inter-server load balancing and intra-server centralized schedulers. Centralized intra-server schedulers schedule tasks on worker threads. Distributed inter-server schedulers cooperatively balance the load between servers. Figure 3.1 illustrates the key components of Malcolm architecture.

3.1 Intra-server Scheduling

Each server runs multiple worker threads to execute tasks. Malcolm uses a centralized queue to buffer incoming tasks. Using a centralized queue outperforms per-worker queues for microsecond-scale workloads [51]. 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 tasks with light-tailed service-time distributions [107]. And PS minimize tail latency for heavy-tailed workloads [123]. Unfortunately, there is no static, work-conserving policy that minimizes tail latency of both workloads. Policies that perform well under light-tailed workloads perform poorly under heavy-tailed workloads, and vice versa [123]. Among existing solutions, ZygOS [94] approximates cFCFS, while Shinjuku [51] 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.

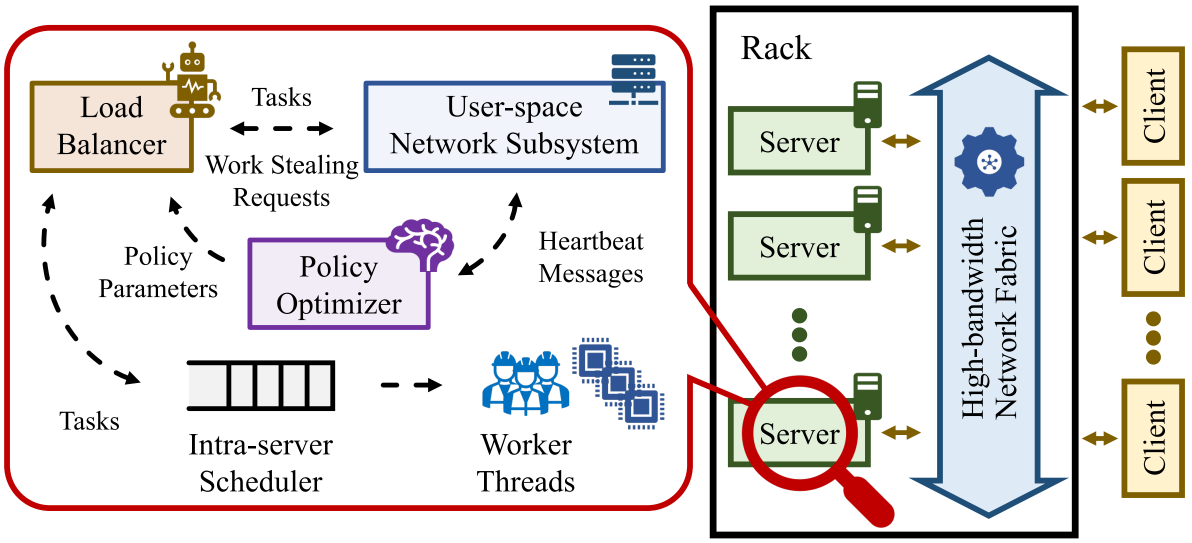


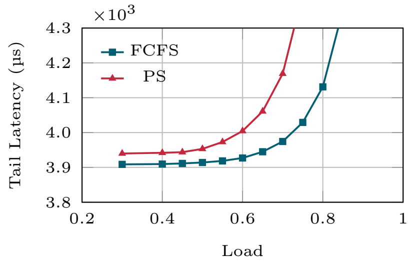
Figure 3.1: Overview of Malcolm architecture.

FCFS and high-dispersion workloads. Optimality of FCFS for light-tailed service-time distributions holds even if distributions have high dispersion. This is illustrated in Figure 3.2 for two representative distributions: (a) HyperExp-1 is a hyperexponential distribution with 50% of service times following Exp(10) and the other 50% following Exp(1000) and (b) HyperExp-2 is a hyperexponential distribution with 99.9% of service times following Exp(25) and the remaining 0.1% following Exp(25025).

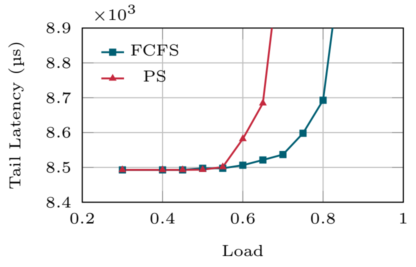
For HyperExp-1, FCFS achieves lower tail latency compared to PS for both 99th-percentile (3.2a) and 99.99th-percentile (3.2b) . For HyperExp-2, PS achieves lower 99th-percentile latencies (3.2c) compared to FCFS. However, in HyperExp-2, only 0.1 percent of the tasks have long service times. Therefore, the 99th-percentile latency does not capture the effect of scheduling these tasks. Intuitively, PS hurts the latency of longer-running tasks. For HyperExp-2, FCFS outperforms PS (3.2d) in terms of 99.99th-percentile latency. We also note that in these simulations, the context-switching overhead is assumed to be zero, which favors PS policy.

3.2 Inter-server Load Balancing

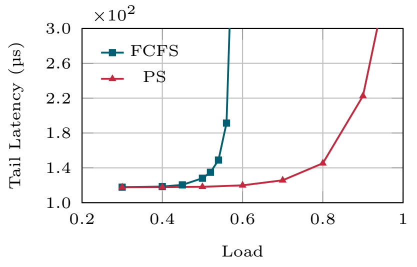
Servers in the rack are interconnected by a high-bandwidth, low-latency network fabric. Servers can be heterogeneous with different computing capacities. Servers exploit userspace



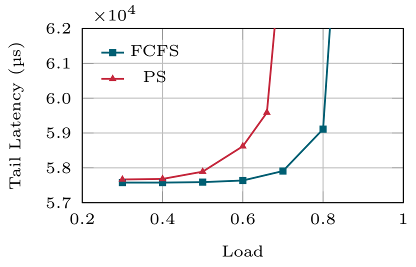
(a) HyperExp-1 (99th)



(b) HyperExp-1 (99.99th)



(c) HyperExp-2 (99th)



(d) HyperExp-2 (99.99th)

Figure 3.2: Tail latency for high-dispersion workloads.

networking stacks to bypass kernel, reducing communication overhead. Clients send their tasks to servers. Different servers can receive tasks at different rates. Servers collectively balance the load in the rack at per-task granularity.

Load balancing. Upon receiving a new task, the load balancer decides whether to accept the task or migrate it to another server §4. This decision is made based on servers' (possibly out-of-date) loads. The load balancer can migrate incoming tasks to less loaded servers when the local load is higher than the load on other servers. Accepted tasks will be scheduled between worker threads by the intra-server scheduler. After processing each task by a worker thread, the load balancer can decide if it needs to steal tasks from other servers. Work-stealing decisions on task completions complement the migration decisions on task arrivals.

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

Load estimation. The load imbalance between two servers is captured by the absolute difference between their loads. When both servers and tasks are homogenous, queue length is an accurate indicator of load on servers. The queue length, however, is not a useful metric in heterogeneous systems. An equal number of waiting tasks on two servers, A and B, does not necessarily mean that the loads on two servers are equal. If A is twice as fast as B, or some tasks are shorter than others, the queue length is not a fine indicator of load. To account for heterogeneity, a more reliable metric is the queue length weighted by the inverse of service rate [104, 132]. This metric closely approximates the expected wait time of the last task in the queue [104]. To estimate service rate, each worker thread maintains a moving average of the inverse of task service times. Average service rate is then estimated by the sum of these moving averages. When tasks are heterogeneous, each worker thread maintains a moving average of inverse service times for each task type.

Instantaneous vs. average load. Temporal load imbalance among servers results in higher tail latency for microsecond-scale workloads. Intuitively, if we can minimize the temporal imbalance, in expectation, the oldest tasks at any given time will finish at almost the same time. Therefore, the main objective of the inter-server 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 servers 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 [113, 43, 42, 108, 93], Malcolm focuses on balancing instantaneous loads to minimize tail latency for microsecond-scale workloads.

Chapter 4

The Distributed Load-balancing Game

In this section, we present the distributed load-balancing game as a framework to balance the load on servers in a rack-scale computer.

Tasks arrive at different servers at different rates. Upon receiving a task, a server decides whether to keep the task or migrate it to another server. Different servers can have different computational capacities. Upon completing a task, a server decides if it needs to steal a task from another server. The state of the game evolves as scheduling decisions collectively shape the load on different servers. Servers calculate their payoffs considering penalties for system load imbalance and task migrations. The goal for servers is to independently find their optimal scheduling policy that maximizes their long-term payoff. We present the game as a stochastic game.

4.1 Background

A game represented in normal form provides a description of the payoff of every agent for every state of the game. In normal form games, the state only depends on players' combined actions. A repeated game is a normal form game that is played several times by the same set of agents. The game which is played repeatedly is usually called the stage game. When the game is repeated infinitely, the agents' objectives are either maximizing average or discounted payoff.

Stochastic games (also known as Markov games) generalize both Markov decision processes (MDPs) and repeated games. An MDP is a stochastic game that a single agent plays, and a repeated game is a stochastic game with a single game state. Intuitively, a stochastic game is a set of normal form games. Players play a game from this set. At any round, the played game depends on the game which is played at the previous round and on all agents' actions, which is defined as the state of the game. The game played at a given round is called a stage game.

The game starts at a random state and follows these steps: During round r, players simultaneously take action. At the end of the round, each player receives a payoff that depends on all players' actions and the game state. Finally, the state transitions to the next state.

4.2 Game Formulation

We model the distributed load-balancing as a stochastic game. The game consists of N heterogeneous servers, represented by N agents. Time is divided into rounds[1]. For microsecond-scale workloads, the duration of each round could be tens to hundreds of nanoseconds. At each round, server i receives a new task with probability p_i and completes a task with probability q_i. This, in essence, assumes that inter-arrival times and task service times have geometric distribution. We relax these assumptions later when we present our multi-agent learning algorithm.

Upon receiving a new task, a server decides whether to accept the task or migrate it to another server. This decision could be made based on (possibly outdated) estimates of the load on servers. An accepted task is put in the server's first-come-first-serve task queue. Once a server completes a task, it could decide if it needs to steal a task from another server. Work-stealing decisions on task completions complement the migration decisions on task arrivals. When the load is not balanced, servers migrate incoming tasks to less-loaded servers. Migration decisions could be suboptimal because the information is propagated with a non-zero delay. Moreover, load estimates on servers could become inaccurate if the system goes through unexpected changes. Hence, if migration decisions are sub-optimal, then the less-loaded server can start stealing tasks from other servers.

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

4.2.1 Load Metric

A system is load balanced if at every round, all servers have roughly the same load. In homogeneous systems, the load on a server is captured by the length of its queue. In such systems, assigning an incoming task to the server with the shortest queue balances the load among servers, an approach widely known as *join the shortest queue (JSQ)*. However, queue length is not an accurate indicator of load on servers in the presence of heterogeneity either in tasks or computational resources. Consider a system with two servers, A and B, where A is much faster than B. Suppose that A has one task more than B. Sending a newly arrived task to B equalizes the queue lengths, but it will likely lead to an imbalanced system in the future. To account for heterogeneity, a better metric is the length of the queue weighted by the inverse of the average service rate [104, 132]. Average service rates can be computed dynamically to reflect fluctuations in tasks execution times and servers' computational capabilities. With this metric, incoming tasks are assigned to servers with the shortest expected completion time, a policy that is known as *shortest expected delay (SED)* routing [104].

4.2.2 States and Actions

The state of each server represents its load. The load on server i at round r is estimated by $x_{i,r} = L_{i,r}/\mu_{i,r}$, where $L_{i,r}$ and $\mu_{i,r}$ are the queue length and service rate of server i at round r, respectively. The load on a server increases if it accepts new incoming tasks, or if it receives migrated tasks from others. Similarly, the load on a server decreases if other servers steal tasks from it, or if it finishes a task. The state of the game at round r is defined to be $x_r = (x_{1,r}, \ldots, x_{N,r})$. The game state evolves over time as agents make scheduling decisions.

We denote the set of scheduling actions taken by agent i at round r by $a_{i,r}$. At round r, the set is empty if server i neither receives a new task nor completes one. It has one element if i either receives a new task or completes one. And it has two elements if i both receives a task and completes one. For example, suppose that server 1 receives a new task at round 10 and completes a running task at the same round. If server 1 accepts the incoming task and steals another one from server 2, then $a_{1,10} = \{\textit{accept}, \textit{steal from 2}\}$. We use $a_r = (a_{1,r}, \ldots, a_{N,r})$ to aggregate all actions taken by all servers at round r.

4.2.3 Strategies

A strategy provides a complete description of how an agent plays the game. Let $h_r = (x_0, a_0, x_1, a_1, \ldots, x_r)$ denote the history of the game at round r. For every possible history, a deterministic strategy prescribes a single action. To allow randomization, a *behavioral* strategy specifies a probability distribution over actions for any given history. In the distributed load-balancing game, a behavioral strategy returns two probability distributions, one over migration actions 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 servers to take scheduling actions based on the current system load and not the history of states and actions. For example, on the arrival of a new task, an overloaded server could randomly pick two other servers and migrate the new task to the one with the lowest load. Similarly, a server could start stealing tasks once its load falls below a (dynamic) threshold. As can be seen in these examples, stationary strategies form a rich class of scheduling policies, which includes well-known policies such as power-of-d-choices, JIQ, and JBT.

4.2.4 Payoffs

The payoff function of agent i at round r is defined as $u_i(x_r, a_r) = -I_i(x_r) - C_i(a_r)$. With this game, we want servers to have almost the same share of the load at any time. We want to achieve this objective with the minimum number of migrations. Therefore, payoff function captures two costs: the cost of load imbalance, I, and the cost of task migrations, C. For C, we use a linear function to penalize each migration with a constant average cost. The load-imbalance function, I, is motivated by the degree-of-queue-imbalance metric [134]. We define the load-imbalance cost function as:

$$I_i(x_r) = \frac{1}{N-1} \sum_{j,k} |x_{j,r} - x_{k,r}|.$$

When all servers have the exact same share of the load, the load imbalance is zero. The load-imbalance cost function captures the main goal of servers, which is to balance the load in the system at every round. The inclusion of migration cost in the payoff function ensures that this goal is achieved with the minimum number of migrations. It is important to note here that balancing instantaneous loads at every round is different from balancing long-term average loads. The former leads to the latter but not the other way around.

Let π_i denote the strategy of agent i, and let $\pi_{-i} = (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_N)$ represent the strategy of all agents other than i [2]. The value function represents the long-term value of a state x for agent i under strategy $\pi = (\pi_i, \pi_{-i})$, and it is defined as:

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

Where $V_i^{\pi}(x)$ is the value of agent i under strategy π starting at state x, and δ is the discount factor. 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.

4.3 Equilibrium

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 the following equation for all states x.

$$\pi_i^* = \arg\max_{\pi_i} \; V_i^{(\pi_i, \pi_{-i})}(x).$$

Agents do not generally know what strategies the other agents will adopt. Therefore, to analyze the outcome of the game, the notion of best response cannot be used by itself. Instead, it can be leveraged to define a stronger notion called *Nash equilibrium*. A Nash equilibrium of a stochastic game is a strategy profile π^* that satisfies the following equation for all agents i and states x.

$$\pi_i^* = \arg\max_{\pi_i} \; V_i^{(\pi_i, \pi_{-i}^*)}(x).$$

Intuitively, a Nash equilibrium is a strategy profile in which no agent benefits from unilaterally changing their strategy, even if they knew the strategies of others.

[2]Throughout this thesis, we use subscript $-i$ to refer to all agents other than agent i.

4.3.1 Equilibrium Analysis

In general, the problem of finding a Nash equilibrium is computationally expensive. Theoretically, the complexity of computing a sample Nash equilibrium of a general-sum finite game with two or more agents is known to be *PPAD-complete* [18]. In practice, it is a common belief that in the worst case, computing a sample Nash equilibrium takes time that is exponential in the size of the game. Fortunately, a Nash equilibrium could be computed in polynomial time for the distributed load-balancing game. This is because, as we show in the rest of this section, the distributed load-balancing game is a *Markov potential game (MPG)*. And for an MPG, a Nash equilibrium can be obtained in polynomial time by solving a corresponding MDP [69].

Definition 1. *A stochastic 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}.*

$$V^i_x(\pi_i, \pi_{-i}) - V^i_x(\pi'_i, \pi_{-i}) = \Phi_x(\pi_i, \pi_{-i}) - \Phi_x(\pi'_i, \pi_{-i}).$$

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. It is easy to see that the following potential function satisfies the condition in the definition of MPG games for the distributed load-balancing game

$$\begin{aligned}\phi(x, a) &= -I_i(x) - \sum_i C_i(a). \\ \Phi_x(\pi) &= \mathbb{E}\left[\sum_{r=0}^{\infty} \delta^r \phi(x_r, a_r) \mid a_r \sim \pi, x_0 = x\right].\end{aligned} \tag{4.1}$$

Claim 1. *The distributed load balancing game is a Markov Potential game with the potential function $\Phi(\cdot)$ defined in Equation (4.1).*

Proof. See appendix (§A.1). □

Using this potential function, the problem of finding a Nash equilibrium is reducible to the following optimal control problem for all states x [69].

$$\underset{\pi}{\text{maximize}} \;\; \Phi_x(\pi).$$

Although this optimal control problem can be solved in polynomial time, the order of the polynomial might be too large for the theoretically efficient algorithms to be practical

[89, 64]. To allow practical solutions, we assume that agents' strategies lie in a parametric set. Given a parameter vector w, a parametric strategy $\pi_w(x)$ maps states x to distributions over actions. This assumption simplifies derivations, allowing practical use of function approximation for finding near-optimal policies.

With parametric functions, the original optimal control problem is replaced with the following parametric problem.

$$\underset{w}{\text{maximize}} \;\; \Phi_x(\pi_w). \tag{4.2}$$

If parametric strategies are expressive enough, we can expect to achieve arbitrarily close performance to that of the optimal non-parametric solution. Given some mild assumptions, which are satisfied by the distributed load-balancing game [3], the parametric optimal control problem (4.2) is guaranteed to have a solution [69]. This solution constitutes a parametric Nash Equilibrium of the distributed load-balancing game. To find the optimal parametric strategies, in the next section, we present a novel distributed multi-agent learning algorithm. Our proposed distributed algorithm solves the centralized control problem (4.2) in a distributed manner and can be implemented and deployed in practice.

[3]For proofs, see appendix (§A.1).

Chapter 5

Algorithm

This chapter presents the distributed algorithm we propose and deploy to solve the distributed load balancing game. The game can be solved using optimal control, centralized or decentralized optimization, and multi-agent reinforcement learning (MARL). We solve the distributed load balancing game using MARL because it allows the load balancer to adapt to the changes in the rack and requires no prior information about the underlying system. The proposed algorithm is online, fully distributed, and simple to deploy.

5.1 Single-Agent Reinforcement Learning

An RL agent learns through interacting with an environment where at every round, r, the agent observes a state, x_r, takes action, a_r, receives a payoff, u_r, and state transitions to x_{r+1}. State transitions and payoff function are assumed to have the Markov property, that is, x_{r+1} and u_r only depend on current state and action, x_r and a_r. Agent actions are sampled from her policy, $\pi(x, a)$, which is a mapping from each state-action pair to the probability of taking that action in that state. Generally, the RL agent has no prior knowledge of the state transition and payoff functions and learns them through trials and errors. The goal of agent is learning a policy, $\pi(\cdot)$, that maximizes the expected discounted sum of payoffs, $J(\pi) = \mathbb{E}_\pi[R]$ where $R = \sum_r \gamma^r u(x_r, a_r)$ is the return, and γ is the discount factor.

5.1.1 Q-Learning

Q-learning [119] is one of the most popular methods in reinforcement learning. Q-learning takes advantage of the action-value function for policy π which is defined as the expected return starting at a state, taking an action, and following policy π afterwards: $Q^\pi(x,a) = \mathbb{E}_\pi[R|x_0 = x, a_0 = a]$. Q-learning agent directly learns the optimal action-value function, $Q^*(x,a) = max_\pi Q^\pi(x,a)$. The optimal action-value function can be recursively written as $Q^*(x_r, a_r) = \mathbb{E}[u(x_r, a_r) + \max_a Q(x_{r+1}, a)]$. Hence, for every (state, action, payoff, next state) the agent observes, (x_r, a_r, u_r, x_{r+1}), she can update her prediction of the optimal value function using temporal-difference (TD) learning [110]:

$$\begin{aligned} Q(x_r, a_r) &\leftarrow Q(x_r, a_r) + \alpha\delta_r \\ \delta_r &= u_r + \max_a Q(x_{r+1}, a) - Q(x_r, a_r) \end{aligned} \tag{5.1}$$

where α is the learning rate, and δ_r is called the TD error. The standard Q-learning algorithm uses a look-up table to store values, making it impractical for continuous or large action-state spaces. Therefore, the action-value function can be estimated using function approximation techniques such as neural networks [83]. The optimal action at a given state is the action that maximizes the action-value function at that state. When action space is large, deriving the optimal action using the value function is computationally expensive.

5.1.2 Policy Gradient

Policy gradient methods directly learn the optimal policy, $\pi_w(\cdot)$, which is parametrized with the parameter vector w. Since the objective is maximizing the expected return, policy gradient algorithms use gradient ascent to move policy parameters w toward the direction suggested by the gradient of the objective, i.e., $\nabla_w J(w)$. Gradient of the expected return can be derived using policy gradient theorem [111]:

$$\nabla_w J(w) = \mathbb{E}_\pi \left[Q^\pi(x,a) \nabla_w \ln \pi_w(a,x)\right] \tag{5.2}$$

Policy gradient methods often differ with each other on how they compute $Q^\pi(x,a)$. Monte Carlo policy gradient, REINFORCE [124], uses simple unbiased monte carlo sampling to estimate action-value function, that is, $R = \sum_r \gamma^r u_r$. REINFORCE works because the expectation of sample return is equal to the action-value function. Another class of policy gradient methods directly learn an approximation of action-value function. This approximation is called the critic, the policy is called the actor, and the method is named actor-critic.

5.2 Multi-agent Reinforcement Learning

The most trivial way to apply RL to the multi-agent setting is to let each agent learn independently using Q-learning, or a policy gradient algorithm [112, 28]. However, since agents are updating their policies independent from one another, the environment becomes non-stationary from each agent point of view [67, 28]. Hence, independent Q-learning is not guaranteed to converge in a general multi-agent system. Moreover, policy gradient methods are notorious for having high variance estimation of the gradient. Since the agent payoff in a multi-agent system depends on the action of all agents, conditioning the action-value function of each agent only on her own actions increases this variance [67].

5.2.1 Centralized Agent

One way to deal with multi-agent settings' difficulties is to train a centralized policy for all agents. The centralized policy observes the state, receives the payoff of all agents, and selects each agent's action. The problem reduces to a single MDP, and all single-agent RL methods become applicable. However, the centralized architecture suffers from scalability issues and is not suitable to solve the distributed load balancing game.

5.2.2 Centralized training, decentralized execution

To address the challenges of multi-agent learning, a popular approach is *centralized training with decentralized execution* (CTDE) [67, 29, 28]. 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 reasons. First, it is not robust as the centralized controller becomes a single point of failure. Second, it is not scalable as requiring all agents to communicate with a single controller could cause long network delays.

5.2.3 Decentralized training, decentralized execution

To meet the requirements of an adaptive microsecond-scale scheduler, Malcolm adapts a decentralized approach to training. In particular, the load-balancing policies are trained separately in a decentralized manner. Each server learns a separate critic and a separate policy. First, learning an action-value function could be expensive in terms of computational and communication costs. To avoid these costs, policy optimizers directly learn a parametrized value function, $V^{\pi}_{\theta_i}$. Second, to train these parametrized value functions, policy optimizers use the stage potential function as payoff(Equation (4.1)).

To allow each policy optimizer to locally calculate the potential function, they periodically broadcast heartbeat messages. As we show in §7.4, these broadcasts could happen very infrequently. In each heartbeat message, servers 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: $\theta_i = \frac{1}{N}\sum_{j\in N}\theta_j$. After each consensus update, policy optimizers reach an agreement on the global parametrized 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 [110].

To learn parameterized value functions, for every (current state, next state, payoff) observation, i.e. (x_r, x_{r+1}, ϕ_r), critics are minimizing the following loss:

$$\mathcal{L}(\theta_i) = \frac{1}{2}||V^{\pi}_{\theta_i}(x_r) - (\phi_r + \gamma V^{\pi}_{\theta_i}(x_{r+1}))||^2.$$

$u_r + \gamma V^{\pi}_{\theta_i}(x_{r+1})$ is a estimate of $V^{\pi}_{\theta_i}(x_r)$ and is called the bootstrapped estimate. Using gradient descent, the update rule can be written as:

$$\theta_i \leftarrow \theta_i + \alpha(\phi_r + \gamma V_{\theta_i}(x_{r+1}) - V_{\theta_i}(x_r))\nabla_{\theta_i} V_{\theta_i}.$$

Where α is the learning rate.

Moreover, actors are trained using policy gradient theorem (Equation (5.2)). Note that $Q^{\pi}(x_r, a_r)$ in Equation (5.2) is equivalent to $u_r + V^{\pi}_{\theta_i}(x_{r+1})$. To reduce the gradient estimate variance, actors subtract $V^{\pi}_{\theta_i}(x_r)$ from $Q^{\pi}(x_r, a_r)$ [110]. The update rule for actors can be written as:

$$w_i \leftarrow w_i + \beta \cdot \delta_r \cdot \nabla_{w_i} \log \pi_{w_i}(a_r, x_r)$$

Where β is the learning rate and $\delta_r = \phi_r + \gamma V_{\theta_i}(x_{r+1}) - V_{\theta_i}(x_r)$.

In the next chapter, we discuss the implementation details of our proposed algorithm which makes it feasible for microsecond-scale deployment.

Algorithm 1: Distributed Policy Optimization

```
Input: $\alpha$ critic learning rate, $\beta$ policies learning rate.
Randomly initialize $\theta_i, w_i$; $\forall i \in N$.
repeat
    Decentralized execution step:
    for all agent $i \in N$ do
        Take action according to $\pi_{w_i}(x_r)$
    end
    State transitions to $x_{r+1}$
    Decentralized training step:
    for all $i \in N$ do
        Compute global payoff, $\phi_r$ (Equation (4.1))
        $\delta_r \leftarrow \phi_r + \gamma V_{\theta_i}(x_{r+1}) - V_{\theta_i}(x_r)$
        Critic: $\theta_i \leftarrow \theta_i + \alpha \cdot \delta_r \cdot \nabla_{\theta_i} V_{\theta_i}$
        Actor: $w_i \leftarrow w_i + \beta \cdot \delta_r \cdot \nabla_{w_i} \log \pi_{w_i}(a_r, x_r)$
    end
    Decentralized consensus step:
    for all $i \in N$ do
        if $r \equiv 0 \pmod{C}$ then
            $\theta_i \leftarrow \frac{1}{N} \sum_{j \in N} \theta_j$
        end
    end
until;
```

5.2.4 Discussion: MARL for Markov Potential Games

Efficient learning in Markov potential games are rapidly attracting attention [69, 78, 79, 72, 32, 62]. Mguni et.al. in [78] propose independent policy optimzation for mean-field potential games. [69] reduces the MPG into a team game and allows agents to learn independently using the potential as the global reward. Authors in [79] propose a decentralized actor-critic algorithm with consensus step. Agents independently optimize their policies and cooperatively learn the potential function. The latter involves a consensus between all agents. More recently, [62, 32] study the convergence of independent policy gradient methods for MPGs. The intuition behind this study is: individual value functions are aligned with the potential function. Hence, moving toward the direction suggested by the gradient of expected return is aligned with moving in the direction suggested by the gradient of potential function.

Chapter 6

Implementation

We have implemented a prototype for Malcolm in 4000 LOC of C++. The code of Malcolm is open-source and available at `https://anonymous.4open.science/r/Malcom-A3E2/`.

6.1 Load Balancing

6.1.1 User-space networking

To provide low-latency host-to-host communication, Malcolm uses eRPC [52], 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 [31] and RDMA [3].

Modern datacenter network hardware provides microsecond-scale latency at very high throughput. Exploiting this remarkable speed requires changing how applications commonly access the network. Traditionally, applications use the heavy-weight networking stacks of the kernel. Consequently, applications must invoke system calls to access the NIC. System calls are not cost-free in that they require a context switch from userspace to the kernel space. Furthermore, with kernel space networking, usually, packets are delivered using interrupts. However, Linux interrupts are known to add considerable costs [51]. Userspace networking eliminates syscalls by removing the kernel involvement after the initial setup. During the initial setup, the NIC's packet queues are mapped to the application memory space. Therefore, the userspace library can disable interrupts and polls the NIC queues instead. Then, sending and receiving packets takes place using MMIO instructions.

6.1.2 Intra-server scheduling

Malcolm's design allows recent dataplane operating systems and intra-server schedulers such as ZygOS [94] and Shinjuku [51] to be deployed for scheduling tasks between worker threads. Malcolm is orthogonal to these works. However, we implement a default task scheduler based on cFCFS discipline.

6.1.3 Inter-server scheduling

In our implementation, we use a *gateway* thread. The gateway thread is responsible for networking, intra-server scheduling, and inter-server load balancing. The gateway thread runs eRPC event loop to receive tasks and send responses. On arrival of each task, the gateway consults the migration policy to decide whether to accept the task or migrate it to another server. For a rack with n servers, the migration policy is represented by a $n - way$ categorical distribution. One of events in this distribution corresponds to accepting the task. The remaining $n - 1$ ones corresponds to the events where the gateway decides to send the task to each of $n - 1$ neighbours.

Consulting the policy then involves sampling an event from this categorical distribution. Based on measurements on our machines (see §7.1), this takes only tens of nanoseconds. If the task is accepted, the gateway pushes the task to a lock-free task queue. Worker threads busy wait on the task queue. Once a task is pushed to the queue, one idle worker, if there is any, dequeues the task. Once the task is completed, and a response is ready to be sent, the gateway 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.

6.2 Policy Optimization

Malcolm dedicates a policy thread to update policy parameters and update migration and work-stealing distributions based on the latest load information. First, the policy thread runs Algorithm (1) to update policy parameters at fixed intervals. The length of these policy-optimization intervals is a configurable parameter of Malcolm. By default, this parameter is set to 1.6 ms. Between two updates, the policy parameters remain unchanged. Second, based on the latest broadcasted loads, the policy thread periodically updates the migration and work-stealing categorical distributions. The frequency of these updates is also a configurable parameter of Malcolm. The default value for this parameter is 100 µs. Between two updates, the probability mass functions of migration and work-stealing distributions do not change.

It is trivial to show that every sub-chain of a Markov chain is a Markov chain. Hence, if we update server policies once in a while, the model proposed in §4 remains Markovian. As a result, servers can broadcast their load to each other only at specific times. We call the time between every two consecutive loads broadcast a window. This will allow agents to select their strategies via infrequent message passing.

6.2.1 Updating policy parameters

Delegating decision-making at microsecond scale to an RL agent requires the following two conditions. First, consulting the agent to update migration and work-stealing distributions should not incur more than a few hundred nanoseconds cost. Second, updating the policy parameters should not add more than a few microsecond overheads to the system. Unfortunately, existing machine learning frameworks fall short of these conditions.

Libraries such as PyTorch [91] and Tensorflow [6] sacrifice performance for programmability. Among other techniques, these libraries often use automatic differentiation to compute gradients. This makes it easy for programmers to implement their models without worrying about gradient calculations. However, it comes at a great performance cost, specially when models are small.

In Malcolm, we use linear function approximation for the critic and the actor[1]. As a result, closed-form formulas for gradient updates can be easily derived. We implement Algorithm (1) using these closed-form formulas in about 600 lines of C++ code. To speed up vector-vector multiplications, our implementation uses x86 SIMD instructions, which are available in most datacenter servers. For comparison, we also implement Algorithm (1) using PyTorch C++ front-end. Table 6.1 compares the execution time of updating policy parameters and calculating migration and work-stealing distributions in Malcolm against the PyTorch implementation.

6.2.2 Caching gradients for mini-batches

It is commonly known that using minibatch stochastic optimization methods is beneficial for training function approximators [41, 125]. However, batching means that the agent

[1]The actor also uses a softmax. The softmax creates a probability distribution over actions. Moreover, it provides appealing exploration/exploitation tradeoff.

	Malcolm				**PyTroch**			
Rack Size	4	8	16	32	4	8	16	32
Updating Distributions	0.1	0.1	0.2	0.5	34	34	34	34
Updating Parameters	0.5	0.9	2	8	40	40	40	42

Table 6.1: Average execution time in microseconds.

updates its parameters only at specific points in time. Consequently, the agent only incurs the learning cost at times which are a multiple of the batch size. Therefore, there will be an undesirable trade-off between the minimum window length and the batch size. To break this cost down, we compute and cache gradients of the agent's parameters at every window. At the end of the batch, we update the parameters using the cached gradients and AdamW optimization algorithm [66]. By doing so, the cost of policy parameter optimization is amortized over several smaller gradient update calculations.

6.2.3 Incremental calculation of probabilities

We also amortize the cost of calculating migration and work-stealing probabilities over several smaller computations.

Let π_i denote the strategy of agent i, and x be the state of the game. The linear policy can be written as:

$$\pi_i(x) = \sigma(\underbrace{W_i x + B_i}_{f_i(x)})$$

Where $\sigma(\cdot)$ is the softmax function, W_i is the $N \times N$ matrix of weights, and B is the $N \times 1$ vector of biases. It is trivial to see that the computational complexity of computing strategy given vector of states and matrices of weights is $o(N^2)$. Moreover, computing a state strategy can be decoupled into two consecutive steps: (a) Applying the affine transformation, $f_i(\cdot)$, (b) applying the softmax function, $\sigma(\cdot)$. Now, assume that only the j^{th} element of the state vector, x_j, changes. We can write :

$$x^{j+} = x + \Delta_{x_j} \cdot e_j$$

Where e_j is a vector with all elements equal to zero except the j^{th}, which is one. And, Δ_{x_j} is the change in the j^{th} element. To compute the strategy of the new state vector, x^{j+}, we

Rack size	4	8	16	32
Incremental prob. updates	7	12	22	43
Incremental para. updates	40	70	159	547

Table 6.2: Cost break down in nanoseconds

first compute $f_i(x^{j+})$:

$$\begin{aligned} f_i(x^{j+}) &= f\left(x + \Delta_{x_j} \cdot e_j\right) \\ &= W_i\left(x + \Delta_{x_j} e_j\right) + B_i \\ &= \Delta_{x_j} W_i e_j + W_i x + B_i \\ &= \Delta_{x_j} W_i e_j + f_i(x) \\ &= \Delta_{x_j} col_j(W_i) + f_i(x) \end{aligned} \tag{6.1}$$

Hence, if we know $f_i(x)$, computing $f_i(x^{j+})$ can be reduced to calculating $\Delta_{x_j} col_j(W_i)$ with $o(N)$ running time. If the entire state vector changes, we can follow the same approach as 6.1 and apply the affine transformation step by step:

$$f_i(x^+) = f_i(x) + \sum_{j \in N} \Delta_{x_j} col_j(W_i) \tag{6.2}$$

Therefore, on every load update a server receives, we use equation 6.1 and build the next state's strategy gradually. This will allow us to take advantage of the unutilized time between received load update messages. With $N = 8$, vectorized implementation of each incremental update only takes 12 ns. This negligible overhead will be hidden by the lag between load update messages sent by different neighbors. At the end of the window, to compute next the strategy of next state, the agent only needs to apply the second step of policy update, that is, applying the softmax function.

Table 6.2 summarizes the cost breakdown of the aforementioned cost optimizations on our testbed (see §7.1). Amortizing the total computation cost over multiple incremental computations enables us to: (a) update the probability mass functions of migration and work-stealing distributions at the time granularity of tens of microseconds and (b) update the policy parameters at the time granularity of a few hundreds of microseconds.

Chapter 7

Evaluation

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

7.1 Experimental Methodology

7.1.1 Experimental environment.

Our experimental environment consists of two high-performance and one low-power machines connected through an NVIDIA Mellanox SB7800 switch. Each high-performance machine has 1TB DDR4 main memory and 128 physical cores (2 $\times$ AMD EPYC 7H12), each operating at 2.6 GHz. The low-power machine has 64GB DDR4 memory and an 8-core CPU (AMD EPYC 3201) operating at 1.5 GHz. Each physical core is pinned to a single worker thread. Each machine has a dual-port 100Gb/s NIC (Mellanox MT27800). All NIC ports are configured to run in the InfiniBand mode.

All machines run Ubuntu LTS 20.04 distribution with Linux kernel version 5.4.0-90. On each machine, we allocate 8 GB of 2 MB huge pages, required by eRPC. We pin each thread to a single core using linux pthread_setaffinity_np system call. Morever, we use userspace CPU frequency scaling governor and set all cores frequency to max.

Since our testbed consists of three physical servers, we run multiple *virtual* servers on each physical server to evaluate the performance of Malcolm on multi-server racks. With eRPC, the difference between communication latencies when servers run on the same

Client/Server	Request Size (B)	Median Latency (µs)	Tail (99th) Latency (µs)
Same Machine	64	3.9	4.4
	256	3.8	4.4
	1024	3.9	4.4
Separate Machines	64	4.2	4.8
	256	4.2	4.8
	1024	4.2	4.8

Table 7.1: Communication latency for eRPC requests.

machine and when they run on separate machines is negligible. Table 7.1 provides measured end-to-end latencies for 250k eRPC requests sent from a client process to a server process. The client pings the server in a closed-loop manner, that is, the client waits to receive a response before sending a new request. When the server runs on a separate physical machine, the median latency is 4.2 µs. When the server runs on the same machine as the client, the median latency is 3.8-3.9 µs. The difference is about 7%.

7.1.2 Synthetic benchmarks

For synthetic benchmarks, we use the following four workloads.

- **Exp**(50) is an exponential distribution with mean equal to 50 µs. This benchmark represents single-type workloads (e.g., single-query data storage services).
- **Bimodal**(90:50, 10:500) is a multimodal distribution where 90% of tasks take 50 µs, and the remaining 10% take 500µs. 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 [99, 15, 117]. 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).

7.1.3 Real-world benchmark

We use RocksDB [22] as a real-world benchmark to evaluate the performance of Malcolm. RocksDB is an open-source production-scale key-value store developed by Facebook. We create a dictionary of one million key-value pairs. Using this dictionary, we create a set of replicated RocksDB databases. Clients send GET(n) queries to servers with n randomly selected keys. Servers then respond with n corresponding values. We consider the following two benchmarks. First, RocksDB-1 has 90% GET(16) queries and 10% GET(64) queries. Second, RocksDB-2 has 50% GET(16) queries and 50% GET(64) queries. On our machines, the average service time of each GET(16) and GET(64) is about 30 µs and 100 µs, respectively.

7.1.4 Load generation

To ensure accurate tail-latency measurements at heavy load, clients are implemented as open-loop load generators [102]. In all of the experiments, inter-arrival times are exponentially distributed. Furthermore, we measure the end-to-end requests completion time latency at clients. Tail latencies are calculated using the entire data obtained by all clients. Clients run from within the rack and use eRPC to communicate with servers.

7.1.5 Alternative baselines

We compare the performance, stability, and efficiency of Malcolm against the following alternative load balancing mechanisms.

- **Client-based power-of-2.** For each task, clients randomly select two servers and send the task to the one with shorter queue length. Power-of-2-choices is a popular scheduling mechanism, and variants of it have been widely used in practice [135, 100, 84, 88, 82]. However, power-of-d could perform poorly in the presence of heterogeneity, either in service time distributions or in server rates.
- **Join-shortest-queue (JSQ).** Distributed dispatchers forward each new task to the server with the shortest queue length. R2P2 [58] and HovercRaft [57] 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 dispatcher's queue. One of the main drawbacks of JSQ and JBSQ is that several dispatchers could send their tasks to the same server, a concept commonly known as "herd behavior." The other drawback is that a single dispatcher could send consecutive tasks to the same

server before new server loads are propagated through the network. To avoid drawbacks, Malcolm dynamically coordinates load-balancing strategies between servers.

- **Join-below-threshold (JBT).** Distributed dispatchers forward each new task to a randomly selected server with queue length below a fixed threshold. If no such server exists, the task is forwarded to a random server. Although centralized JBT is proved to be throughput optimal in heavy-traffic regimes, the optimal threshold is a function of the load on servers and approaches infinity as load increases [106]. Moreover, similar to JSQ, decentralized JBT suffers from herd behavior and infrequent information propagation. Finally, JBT with a fixed threshold on queue lengths performs poorly in the presence of heterogeneity as queue length does not represent the load on a server when different servers have different computing powers.

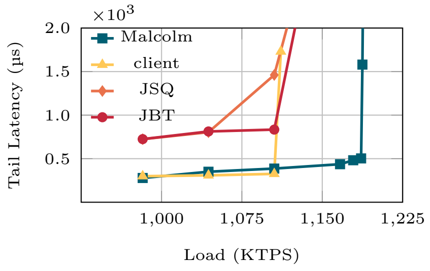
Load propagation. For client-based power-of-2, servers piggyback their load information with responses to clients. For JBT and JSQ, servers periodically broadcast the length of their task queue. The period of these broadcasts is equal in JBT, JSQ, and Malcolm. We show later in §7.4.1 that although JBT and JSQ perform relatively good with up-to-date load information, they exhibit poor performance when load information is out-of-date.

7.2 Performance

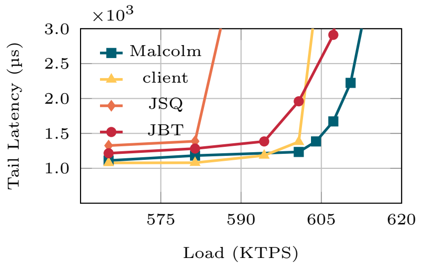
We compare the performance of Malcolm against alternative load-balancing mechanisms in terms of 99th-percentile latency. Note that since load imbalance leads to long tail latencies [17, 12, 131], tail latency is a good measure of load imbalance as well as quality of scheduler's decisions. For this, we use synthetic and real-world workloads and consider homogeneous and heterogeneous configurations.

7.2.1 Synthetic workloads.

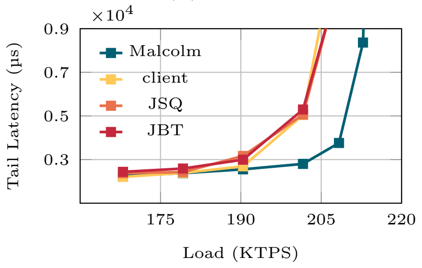
First, we consider a homogeneous rack with 8 servers, each of which with 8 worker threads. Figure 7.1 compares the tail latency of Malcolm against other baselines under different loads. For all benchmarks, Malcolm outperforms other baselines. For Exp, Malcolm maintains low tail latency at up to 1180 KTPS load, whereas the other baselines cannot exceed 1100 KTPS. Note that the maximum theoretical load for Exp is 1280 KTPS (= 64 $\times$ 20 KTPS). For Bimodal, HyperExp-1, and HyperExp-2, this value is about 674, 233, and 166 KTPS, respectively. Figure 7.1 shows that Malcolm can achieve low latency under



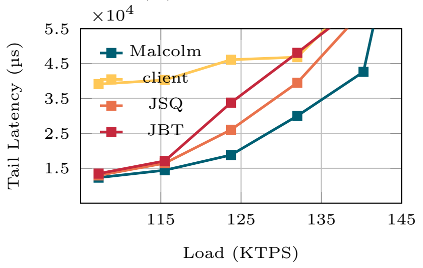
(a) Exp



(b) Bimodal



(c) HyperExp-1



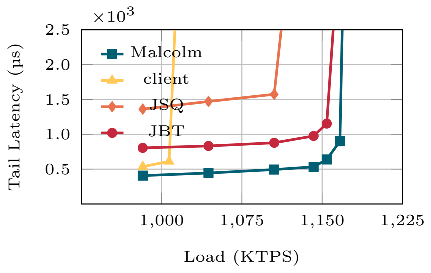
(d) HyperExp-2

Figure 7.1: 99th-percentile latency for synthetic workloads in homogeneous rack.

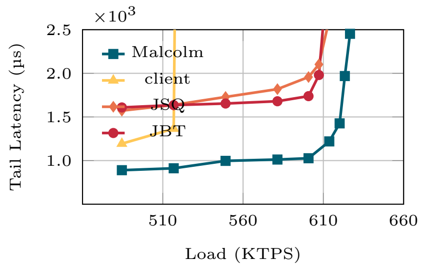
up to maximum load of 92%, 90%, 90%, and 85% for Exp, Bimodal, HyperExp-1, and HyperExp-2, respectively.

Malcolm supports higher throughput at lower tail latencies because it minimizes temporal load imbalance among servers. JSQ and JBT perform poorly as they suffer from herd behavior and infrequent information propagation. These affect Malcolm to a much lesser extent as servers in Malcolm coordinate their load-balancing strategies. In other words, servers in Malcolm converge to a Nash equilibrium. Herd behavior is not an equilibrium because at least one server benefits from her strategy. Client-based power-of-2 also performs poorly because at any given time, each client has up-to-date information on a fraction of servers that have piggybacked their loads to the client. This could lead to low quality scheduling decisions. Moreover, the client performs poorly for high-dispersion service-time distributions as power-of-2 is negatively affected by heterogeneity, even in tasks.

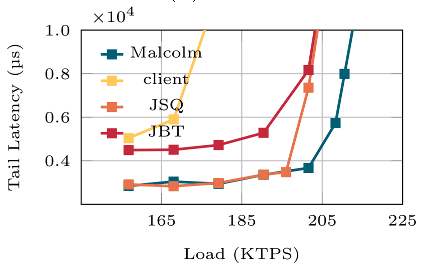
Next, we consider a heterogeneous rack with two fast and nine slow servers. Each fast server has 14 worker threads, and each slow server enjoys four worker threads. Figure 7.2 shows tail latency of different baselines under different loads. Malcolm again outperforms all the alternative baselines for all workloads. Compared to other baselines, for lower loads, Malcolm improves tail latency by up to a factor of two. And for the same tail latency, Malcolm achieves up to 20% more throughput. Malcolm can reach a maximum



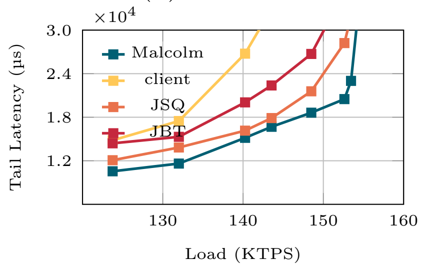
(a) Exp



(b) Bimodal



(c) HyperExp-1



(d) HyperExp-2

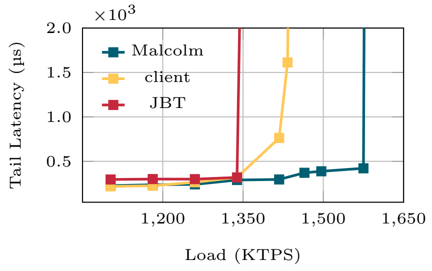
Figure 7.2: 99th-percentile latency for synthetic workloads in heterogeneous rack.

load of up to 91%, 91%, 90%, and 93% at low tail latency for Exp, Bimodal, HyperExp-1, and HyperExp-2, respectively. Client-based power-of-2 performs very poorly for all the benchmarks. This behavior is expected in a heterogeneous rack as discussed in §2. Herd behavior and infrequent information propagation degrade performance of JSQ and JBT. Performance degradation is less for workloads with higher average service time because load information are fresher for a fixed load-propagation period.

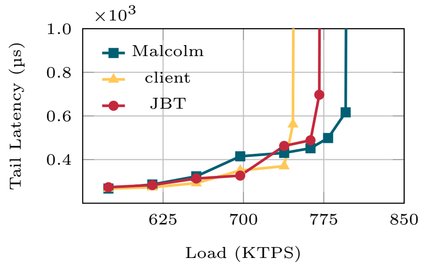
7.2.2 Real-world workloads.

We use the same homogeneous and heterogeneous configurations to evaluate the performance of Malcolm for RocksDB benchmarks. Figure 7.3 compares 99th-percentile latency of different baselines for the homogeneous rack configuration[1]. Malcolm outperforms all other baselines for both workloads by learning optimal coordinated load-balancing strategy which minimizes load imbalance among servers. In doing so, Malcolm provides up to 10% more throughput at lower tail latency. Client-based power-of-2 performs poorly due to heterogeneity in task types. JBT also performs poorly because of infrequent load propagation and herd behavior. Figure 7.4 illustrates the results for the heterogeneous configuration.

[1]JSQ is omitted because it performs very poorly for this workload.

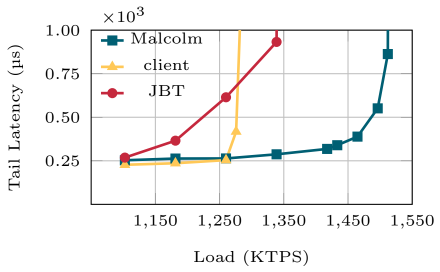


(a) RocksDB-1

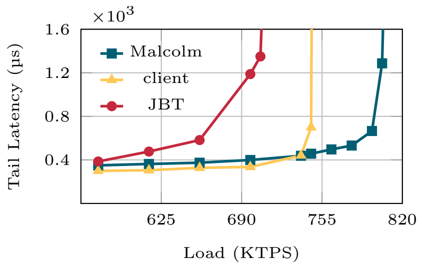


(b) RocksDB-2

Figure 7.3: 99th-percentile latency for RocksDB benchmarks in homogeneous rack.



(a) RocksDB-1



(b) RocksDB-2

Figure 7.4: 99th-percentile latency for RocksDB benchmarks in heterogeneous rack.

Malcolm again outperforms other baselines by providing up to 20% more throughput for a fixed tail latency. Client-based power-of-2 load balancer performs poorly as both tasks and servers are heterogeneous.

7.2.3 Load imbalance

Malcolm outperforms alternative baselines for all benchmarks since Malcolm minimizes instantaneous load imbalance between servers. To show this, we consider a heterogeneous rack with two fast and 6 slow servers. Each fast server runs 14 worker threads, and each slow server runs four workers. Figure 7.5 depicts average load in terms of expected wait time on fast and slow servers over a period of 1 second at 85% load. As illustrated by this figure, Malcolm equalizes instantaneous load of fast and slow servers in about 100 ms. Client-based power-of-2 fails to stabilize the system as the load on slow servers grows over time. JSQ and JBT stabilize the system but fail to avoid load imbalance among servers. This is mainly because the two policies are sensitive to the frequency of load-information propagation. Moreover, queue length is not an adequate indicator of load in the presence

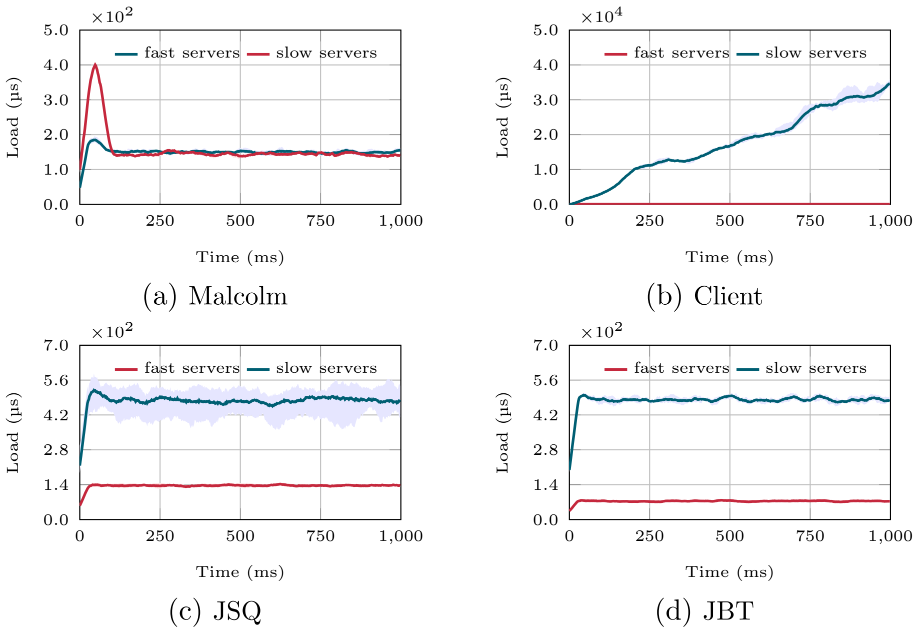


Figure 7.5: Load over time for different load schedulers.

of heterogeneity.

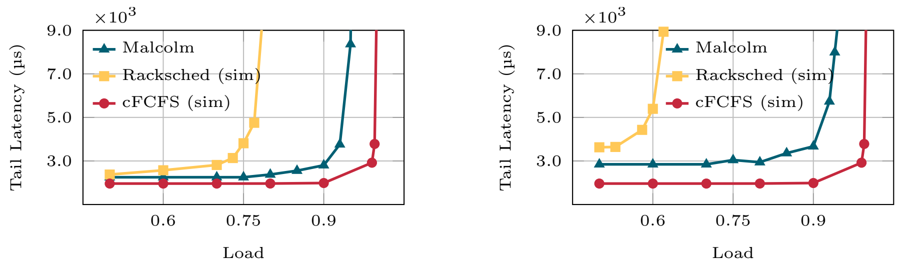
7.2.4 Comparison Against RackSched

We compare Malcolm against RackSched [135] and centralized FCFS using simulations[2]. We simulate the same homogeneous and heterogeneous configurations in §7.2. For each experiment, similar 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. This ensures that tail-latency results include latency of tasks with the same service-time distribution as intended in the workload.

To simulate RackSched, we implement an ideal centralized power-of-2 scheduler with load piggybacking (i.e., network delay is assumed to be zero). For RackSched, we also implement PS policy for intra-server scheduling. The preemption interval is set to 500 µs. We also simulate ideal cFCFS as a reference for the theoretically optimal policy. The results for RackSched and cFCFS do not capture any network or system overheads. For Malcolm, we present data from actual deployments.

[2]We do not have access to a programmable switch.

The results are shown in Figure 7.6. The 99th-percentile latency of ideal RackSched quickly goes up as the load increases for both homogeneous and heterogeneous configurations. One reason for this is that RackSched uses power-of-2 choices. In general, power-of-2-choices is unstable in the presence of heterogeneity, both in service times and service rates. The other reason for this is that RackSched uses PS for intra-server scheduling. PS hurts the latency of longer-running tasks. In fact, if the preemption interval decreases simulations take longer. This is mainly because longer-running tasks stay in the system for a longer periods of time. Malcolm, on the other hand, keeps tail latency low for both configurations even at the heavy-traffic load by dynamically equalizing the load on fast and slow servers.



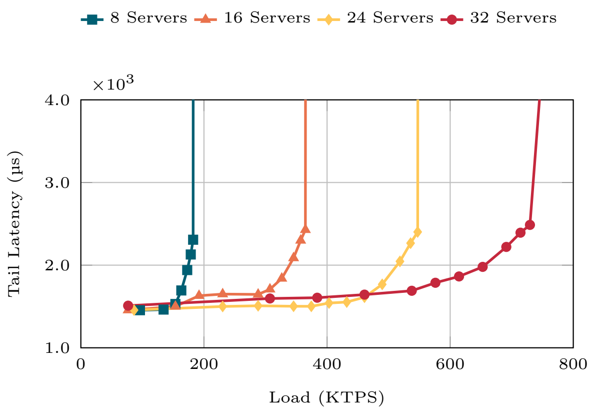
(a) Homogeneous (b) Heterogeneous

Figure 7.6: RackSched (sim.) vs. Malcolm (real deployment).

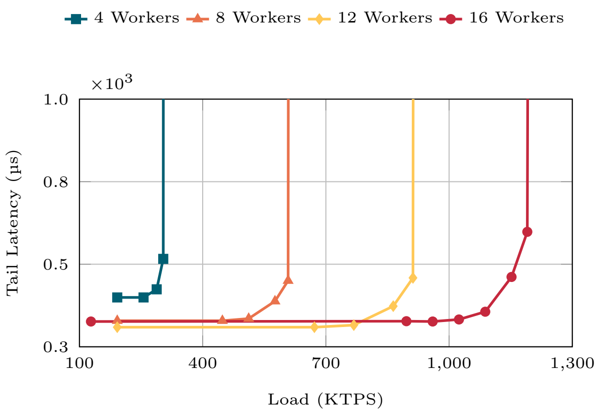
7.3 Scalability Analysis

We conduct two experiments to measure the scalability of Malcolm and its implementation. First, we fix the number of worker threads per server to six and increase the number of servers. Second, we fix the number of servers to two and increase the number of worker threads. In the first experiment, clients generate synthetic workloads according to Exp(250). In the second experiment, service times follow Exp(50). We are interested in the 99th-percentile latency of tasks.

Figure 7.7a shows the results for the first experiment. As can be seen, the throughput of Malcolm increases almost linearly as more servers are added. For Exp (250), with six worker threads, the theoretical maximum throughput for 8, 16, 24, and 32 servers is 192, 384, 576, and 768 KTPS, respectively. For all configurations, the load can reach up to 95% at a tail latency that is only a few multiples of the average service time. Figure 7.7b illustrates the result of the second experiment. As can be seen again, the throughput of our user-space intra-server scheduler increases almost linearly up to 16 worker threads. Beyond that, an extra inter-server scheduler has to be added to each server.



(a) Inter-server



(b) Intra-sever

Figure 7.7: Scalability of Malcolm.

7.4 Adaptability

Malcolm is a dynamic load balancer that learns to adapt to variations in computational resources as well tasks service times. Moreover, Malcolm learns to load balance the system with infrequent load information sharing.

7.4.1 Sensitivity to load-broadcasting period

So far, in all experiments, we set the length of the load-broadcasting (and policy-updating) period to 100 µs. In this section, we study the performance of Malcolm under different period lengths. We compare the results against JBT and JSQ. We consider a homogeneous rack with 8 servers, each running 8 worker threads. Clients generate tasks according to Exp(50) at 85% load. Figure 7.8 shows the measured 99th-percentile latency achieved by Malcolm, JBT, and JSQ for different load-broadcasting intervals. As can be seen, with Malcolm, the tail latency remains low for load-balancing intervals as long as 1 ms. The main reason for this is that the policy optimizers in Malcolm, in the process of learning optimal cooperative load-balancing policy, implicitly learn the system dynamics for any fixed load-balancing 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 JBT and JSQ, as they both fail to provide acceptable tail latencies as the length of the load-broadcasting period increases.

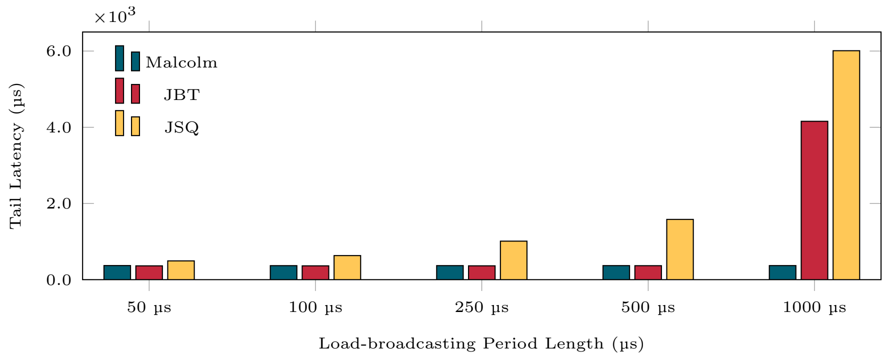


Figure 7.8: Tail latency measurement for different load-broadcasting intervals.

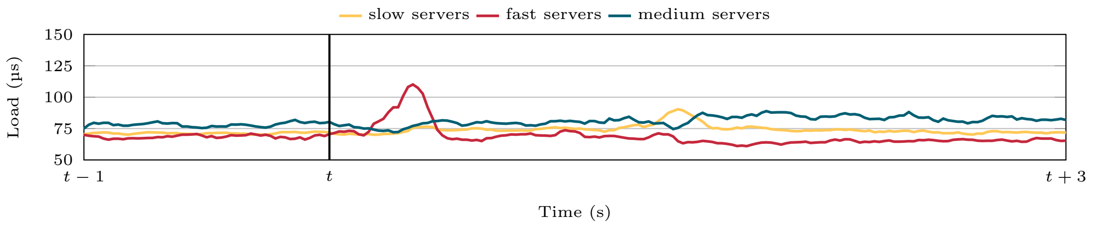
7.4.2 Sensitivity to fluctuations in load and service rate

We study the adaptability of Malcolm when servers' arrival rate or service rate changes. We consider a rack with three types of servers – fast servers with eight workers, medium servers with six workers, and slow servers with four workers. The rack consists of 12 servers, four of each type. Service times follow Exp(50). We conduct two experiments. First, we fix the load at 540 KTPS. We start by equally dividing the traffic between all servers. At time t, we change the shape of the traffic by sending 60% of all tasks to fast servers. Figure 7.9a shows the load in terms of expected wait time on different server types before and after the change. It can be seen that the load on fast servers increases; however, the system quickly adapts and learns to minimize load imbalance.

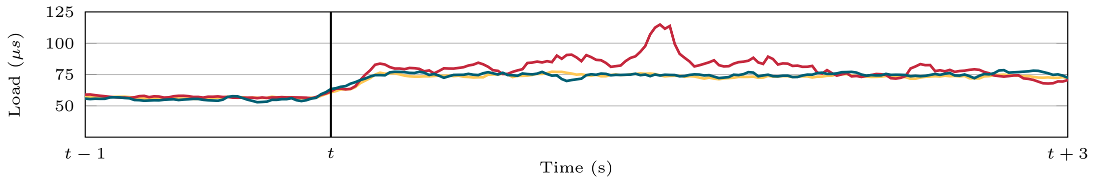
Second, we set the initial arrival rate to 360 KTPS. The initial traffic is equally divided between servers. At time t, the arrival rate increases to 540 KTPS. The additional 180 KTPS are routed to the fast servers. Although the load on fast servers initially increases, the system quickly converges to an equilibrium where load imbalance is very small.

Third, we fix the load and slow down the fast servers by 50%. Figure 7.9c shows the load on different server types before and after the change. The load on fast servers initially increases but quickly converges to other servers.

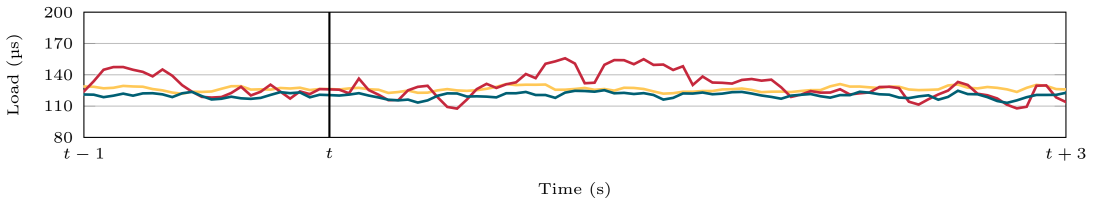
In the experiments, Malcolm learns to adapt to the changes in the system parameters. Additionally, Malcolm's immediate response to drastic changes does not make the system



(a) Change in traffic shape



(b) Change in arrival rate.



(c) Change in service rates

Figure 7.9: Sensitivity to arrival and service rates.

unstable. Note that average loads reflect the changes with some delay. This has two main reasons. First, we use the exponential moving average to track server 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.

Chapter 8

Related Works

Load balancing in distributed systems has been extensively studied before. Most of the theoretical works in the literature study the behavior of a centralized scheduler [134, 45, 44, 47, 82, 63]. Centralized models either use a centralized dispatcher [63], or reduce the distributed load balancing problem into several centralized ones [39]. Despite their simplicity, due to the scalability issues, implementing a centralized load balancer is almost infeasible in practice.

Wierman and Zwart [123] have shown that there is no static work-conserving scheduling policy that minimizes tail latency of both heavy-and light-tailed workloads. On the one hand, First-Come-First-Serve–FCFS–optimizes tail latency of workloads with light-tail distribution [107]. On the other hand, Processor Sharing–PS–and Shortest Remaining Processing Time–SRPT–are known to minimize tail latency of heavy-tailed workloads [123].

Based on the workload distributions, the latest works in the system community exploit either of the following. (a) centralized FCFS for low-dispersion workloads, (b) PS for high-dispersion workloads [94, 51, 135].

ZygOS [94] approximates cFCFS via frequent work stealing. Shinjuku [51] implements processor sharing with either single or multi-queue policies. The single queue policy places all incoming tasks into a single FCFS queue. The dispatcher in the multi-queue policy keeps one queue per task type. Incoming tasks are separated based on their types and placed into their corresponding queues. On every dispatching decision, the Shinjuku dispatcher selects a non-empty queue and dequeues the task at the head of that very queue.

Ideal cFCFS requires a centralized scheduler that queues incoming tasks and dispatches one task every time a worker becomes idle in the cluster. The centralized scheduler needs to have enough memory to queue all incoming tasks. This centralized scheduler does not scale well and quickly becomes the system bottleneck.

Join-Shortest-Queue (JSQ) is one the most popular load balancing disciplines [58, 57, 38, 126]. In this policy, tasks are immediately dispatched to the server with the shortest queue length. Hence, JSQ is a bufferless scheduler with zero dispatching delay. When load information is up-to-date, a single dispatcher JSQ minimizes the mean execution time of tasks in homogenous systems [25, 126, 121, 30]. However, when load information is outdated, joining the shortest queue is not always optimal [81, 132].

R2P2 and Hovercraft [58, 57] load balance microsecond-scale remote procedure calls. To do so, R2P2 implements a centralized Join-Shortest-Bounded-Queue load balancing policy in either software or a programmable switch. JBSQ is a variant of JSQ where servers have bounded queues. Since queue length is not a fine indicator of load on servers [88], both JBSQ and JSQ suffer from poor load metrics.

Zhang et al. in [130] introduce a variant of JSQ called Min-Worker-Set (MWS) to load balance serverless platforms. In Min-Worker-Set, the task is scheduled on the virtual machine that minimizes queueing time, execution time, and cold starts. Blowfish [56] uses JSQ to load balance requests between shard replicas in a distributed data store.

Join-Idle-Queue (JIQ) is another popular load balancing policy [68, 49]. In JIQ, the dispatcher assigns tasks only to idle servers, if there is any, or to a randomly selected server. Although JIQ is not delay optimal in heavy traffic, it performs strictly better than random scheduling [133].

Power-of-d-choices has been widely used in practice for load balancing [88, 100, 84, 135]. The power-of-d dispatcher queries d randomly selected servers and sends the incoming task to the least loaded one [82]. This approach is proved to yield short queues in homogeneous systems. However, in general, power-of-d-choices is unstable in heterogeneous systems [37, 132]. In addition, short queues are not necessarily equivalent to low latency because the queue length is not a fine indicator of the waiting time. However, power-of-d involves prohibitive message passing between the servers and dispatchers. This communication overhead becomes restrictive when network delays are comparable to execution times. Moreover, with multiple power-of-d dispatchers, the system can suffer from herd behavior.

To mitigate these issues, more recently, sparrow [88] authors introduced late-binding. With late binding, the dispatcher queries d servers, but servers will not immediately respond. Alternatively, each server puts a reservation for the dispatcher's request at the end of its task queue. Once the reservation reaches the head of the queue, the server requests the task from the dispatcher. The dispatcher schedules the task on the first responding server. EC-Cache [98] is a cluster-level caching mechanism that uses late-binding to load balance I/O requests. More recently, authors in [45] analytically studied the response time of late-binding when arrivals have Poisson distribution. Late binding is suboptimal when tasks execution times are comparable to network delay (see §2.2.3).

Nasir et al. in [84] adapt power-of-d-choices to distributed stream processing engines. In doing so, each scheduler estimates its own load on workers. Then, each worker only tries to load balance only its own load on all servers. The intuition is: if all schedulers keep their own share of the load balanced among all servers, the system remains load balanced. Chaos [100] is a distributed graph processing system. To keep all the storage engines busy, Chaos proposes a batching technique that is inspired by power-of-d-choices.

Authors in [37] adapt power-of-d choices to heterogeneous systems. To do so, servers are qualitatively categorized into two groups, namely, fast and slow servers. Instead of choosing d servers, d_f and d_s servers are chosen from fast and slow servers, respectively. Then, the load balancer tries to utilize fast idle servers, if there are any. Otherwise, the load balancer chooses idle slow servers with probability p_s or busy fast servers with probability $1 - p_s$. If there is no idle server among queried servers, with probability p_f a fast server is chosen for job placement and with probability $1 - p_f$ a slow one. Although this approach addresses some drawbacks of power-of-d-choices, categorizing servers into fast and slow servers is not always practical.

To the best of our knowledge, Racksched [135] is the latest rack-scale scheduler designed for microsecond-scale services. RackSched implements a centralized scheduler inside the Top of Rack switch. The design suffers from several drawbacks. First, offloading additional computation to the ToR switch could degrade network throughput [76]. Second, the ToR switch has limited memory and computational capacities. Hence, RackSched implements power-of-d-choices to approximate cFCFS.

Distributed versions of JSQ and power-of-d-choices suffers from herd behavior (see §2.3.2). To deal with this issue, in C3 [109], dispatchers keep local estimates of queue length on each server.

Local Shortest Queue (LSQ) [115] is a load balancing policy that is introduced to deal with outdated load information on servers. Under the LSQ framework, each dispatcher keeps its local estimate of server queue lengths and schedules tasks on the shortest among them.

Join-Below-Threshold (JBT) is another load balancing technique that has been studied recently in the literature [47, 133]. In JBT, the dispatcher keeps track of servers whose load is below a threshold. New tasks are assigned to servers that are randomly picked from the below-threshold list. If there is no server with a load below the threshold, the task will be dispatched to a random server. Zhou et al. in [133] extend this approach to heterogeneous systems: each server reports its service rate to the dispatcher. The dispatcher then chooses servers from the below-threshold list in proportion to servers' service rates. This approach solves the communication overhead of power-of-d and accounts for heterogeneity. Nevertheless, the system still suffers from herd behavior when there are multiple dispatchers. Moreover, the optimal threshold is a function of arrival rate and tends to infinity as the arrival rate increases.

Apollo deploys an optimistic distributed scheduler for scheduling workloads ranging from millisecond to a few hundred seconds [13]. In Apollo, each task is scheduled on a server which minimizes the estimated completion time of the task. Apollo schedulers hold an up-to-date view of the system. This information, that is, resource availability and load on servers, is used for estimating task completion times. Omega [103] is another optimistic scheduler. In Omega, if two schedulers concurrently send their tasks to the same server, the server only accepts one of them. Hence, Omega avoids herd behavior when communication delay is a negligible fraction of computation cost.

The diversity of applications, on the one hand, and the heterogeneity of datacenter resources, on the other hand, makes scheduling a difficult problem. Moreover, interference between colocated workloads makes the problem more difficult. Paragon [20] is a heterogeneity and interference aware scheduler. Paragon is a greedy scheduler that schedules tasks on servers with minimum interference. PARTIES [14] is another QoS aware resource management framework that makes sharing a node between several latency-critical applications possible. Heracles [65] is a feedback-based controller that collocates batch tasks on the same machines. None of the existing QoS-aware schedulers meets the requirements of a high throughput low latency microsecond scale scheduler.

Dean and Barroso in [19] propose hedge requests where clients send requests to more than one worker. Once the client receives the response, it cancels other outstanding requests. Authors in [101] explore load balancing in systems where multiple processors share memory. Quincy [48] is a task-level scheduler that balances data locality, starvation freedom, and fairness. Dremel [77] is a hierarchical scheduler that decomposes each query into a serving tree.

Hadoop fair scheduler [128] implements a centralized scheduler for the entire cluster. Facebook Corona scheduler [5], Mesos [46], and Yarn [116] are other examples of centralized schedulers.

There have also been several attempts to incorporate game theory for load balancing. Most of these methods are static and minimize the average completion time of tasks [108, 43, 42, 93]. Furthermore, the majority of these works rely on clients for load balancing servers [43, 42, 26]. Hence, none of these attempts are suitable for microsecond-scale deployment.

Authors in [43] model the static load balancing as a cooperative game. In this model, servers are heterogeneous, and tasks are homogenous. Each computer is modeled as an M/M/1 queue. The objective is to minimize the average execution time of tasks.

Grosu et al. in [42] model the load balancing in a heterogeneous system as a non-cooperative game. In this model, clients are responsible for load balancing servers. Similar to [43], clients are minimizing the mean execution time of their tasks.

[26] studies the convergence time to Nash Equilibrium in a load balancing model where jobs are players and play selfishly to minimize their cost. Subrata et.al in [108] model load balancing in computational grids as a non-cooperative game. The grid is assumed to be heterogeneous. The players in this game are dispatchers who selfishly minimize the mean execution time of their tasks. Hence, dispatchers are competing for resources. However, when clients selfishly minimize their requests completion times, the system can become unstable at loads as low as 50%.

Yun and Proutiere in [127] study distributed load balancing in a heterogeneous cluster where service rate depends not only on servers but on users as well. In this model, servers are sharing their resources in time among their clients using a processor sharing policy. In the beginning, clients randomly choose a server, and afterward, periodically probe other servers and migrate their task if migration improves their service rate. Authors characterize this system as a game where players are clients who are updating their strategy. Also, it has been proved that when the number of clients is fixed, this system has a pure strategy nash equilibrium, which is also proportionally fair when the user population is large enough.

Reinforcement learning has also been investigated for scheduling and resource management. FIRM [97] is a resource management framework that leverages reinforcement learning for identifying applications that are causing resource contention and resources which are in contention. Decima [71] uses RL to learn the best scheduling policy considering workload characteristics.

Tosounidis et al. in [114] study a load balancing strategy in a data center network using deep Q-learning with software-defined networking management technology. The Q-learner agent in this paper forms a centralized scheduler that uses the SDN feature, that is, decoupled control plane from the data plane. The state of the system is defined using an N by N matrix and a vector representing the utilization of servers. The reward function is defined to maximize system throughput.

The database community has also explored machine learning for optimizing database queries [118, 75, 60]. Neo [73] uses machine learning to generate query execution plans.

SageDB [59] takes advantage of machine learning to learn the optimal query plans. ReJOIN [74] uses reinforcement learning to optimize join order selection.

Chapter 9

Future Works and Conclusion

9.1 Future Works

9.1.1 Beyond single rack

Malcolm can be scaled beyond a single rack. To do so, Malcolm can be deployed in a hierarchical manner. Each rack plays the role of a server in the distributed load balancing game. And one server per rack plays the role of the load-balancer. Within each rack, servers play the distributed load-balancing game together. Across racks, load-balancer servers play the load-balancing game together. Design, analysis, and implantation of this hierarchical datacenter-scale scheduler are promising directions for future work.

9.1.2 Reconfiguration

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

9.1.3 Single-digit microsecond-scale workloads

Malcolm can be effectively exploited for load balancing single-digit microsecond-scale tasks. Malcolm is orthogonal to dataplane operating systems such as ZygOS [94] and Shinjuku [50]. Hence, Malcolm can be used in conjunction with these intra-server tasks schedulers to schedule single-digit microsecond-scale workloads. Moreover, Malcolm can be implemented inside userspace networking stacks such as eRPC [52]. Furthermore, one could also implement Malcolm in programmable network hardware accelerators such as smart network interface cards (smartNIC) or FPGA-based NICs.

9.1.4 Communication cost

In Malcolm, we approximated communication cost (see §4) using linear cost model. This was enabled by the low latency high bandwidth networking subsystem (see Table 7.1). When end-to-end latencies have high variance or non-deterministic average, one should dynamically approximate the communication cost. Exploiting Malcolm in a system with high dispersion end-to-end network delay is an interesting direction for future works.

9.2 Conclusion

We presented Malcolm, a distributed rack-scale scheduler designed for microsecond scale services. Malcolm is a heterogeneity-aware and dynamic load balancer. We modeled interactions between servers as a stochastic cooperative game. We proved that the distributed load balancing game is a Markov Potential Game. Hence, optimal scheduling strategies can be obtained in polynomial time. We proposed a fully decentralized learning algorithm to find optimal policies in this game. In the algorithm, agents take advantage of the global potential function. To coordinate their strategy, servers periodically broadcast their load over the network. We evaluated Malcolm in a rack-scale computer using a variety of real-world and synthetic benchmarks. Malcolm provides up to 20% higher throughput at low tail latency compared to state-of-the-art load balancers.

References

[1] Intel® rack scale design (Intel® RSD). https://rb.gy/uxvjjt.

[2] Rack-scale computing. https://rb.gy/ps9fzo.

[3] RDMA core userspace libraries and daemons. https://github.com/linux-rdma/rdma-core.

[4] Under the hood of googles tpu2 machine learning clusters. https://rb.gy/3xmprc.

[5] Under the hood: Scheduling mapreduce jobs more efficiently with corona. Dec 2018.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and implementation (OSDI)*, pages 265–283, 2016.

[7] Krste Asanović. Firebox: A hardware building block for 2020 warehouse-scale computers. https://www.usenix.org/node/179918, 2014.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM (CACM)*, 60(4):48–54, mar 2017.

[9] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[10] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)*, 34(4):1–39, 2016.

[11] Dimitri P Bertsekas et al. *Dynamic programming and optimal control: Vol. 1*. Athena scientific Belmont, 2000.

[12] Geoffrey Blake and Ali G Saidi. Where does the time go? characterizing tail latency in memcached. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 21–31. IEEE, 2015.

[13] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, 2014.

[14] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 107–120, 2019.

[15] E. G. Coffman and R. C. Wood. Interarrival statistics for time sharing systems. *Communications of the ACM (CACM)*, 9(7):500–503, 1966.

[16] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A network stack for rack-scale computers. In *Proceedings of the Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM)*, page 551–564, 2015.

[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 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)*, pages 35–48, 2019.

[18] Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM (CACM)*, 56(2):74–80, 2013.

[20] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[21] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the*

Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM), page 431–442, 2014.

[22] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 33–49, 2021.

[23] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[24] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep RL: A case study on PPO and TRPO. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[25] Atilla Eryilmaz and Rayadurgam Srikant. Asymptotically tight steady-state queue length bounds implied by drift conditions. *Queueing Systems*, 72(3):311–359, 2012.

[26] Eyal Even-Dar, Alex Kesselman, and Yishay Mansour. Convergence time to nash equilibrium in load balancing. *ACM Transactions on Algorithms (TALG)*, 3(3):32–es, 2007.

[27] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[28] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[29] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NeurIPS)*, page 2145–2153, 2016.

[30] G.J. Foschini and J. Salz. A basic dynamic routing problem and diffusion. *IEEE Transactions on Communications*, pages 320–327, 1978.

[31] Linux Foundation. Data plane development kit (DPDK). http://www.dpdk.org, 2015.

[32] Roy Fox, Stephen McAleer, Will Overman, and Ioannis Panageas. Independent natural policy gradient always converges in markov potential games. *arXiv preprint arXiv:2110.10614*, 2021.

[33] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. *Caladan: Mitigating Interference at Microsecond Timescales*, pages 281–297. 2020.

[34] Jason Gaitonde and Éva Tardos. Stability and learning in strategic queuing systems. In *Proceedings of the 21st ACM Conference on Economics and Computation, (EC)*, pages 319–347, 2020.

[35] Jason Gaitonde and Éva Tardos. Virtues of patience in strategic queuing systems. In *Proceedings of the 22nd ACM Conference on Economics and Computation (EC)*, pages 520–540, 2021.

[36] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 27–38, 2014.

[37] Kristen Gardner, Jazeem Abdul Jaleel, Alexander Wickeham, and Sherwin Doroudi. Scalable Load Balancing in the Presence of Heterogeneous Servers. *ACM SIGMETRICS Performance Evaluation Review*, page 37–38, 2021.

[38] Kristen Gardner and Cole Stephens. Smart dispatching in heterogeneous systems. *ACM SIGMETRICS Performance Evaluation Review*, pages 12–14, 2019.

[39] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *Proceedings of the IEEE INFOCOM*, volume 4, pages 2253–2262. IEEE, 2004.

[40] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, 2016.

[41] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[42] Daniel Grosu and Anthony T Chronopoulos. Noncooperative load balancing in distributed systems. *Journal of Parallel and Distributed Computing*, 65(9):1022–1034, 2005.

[43] Daniel Grosu, Anthony T Chronopoulos, and Ming-Ying Leung. Load balancing in distributed systems: An approach using cooperative games. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–pp. IEEE, 2002.

[44] Tim Hellemans, Tejas Bodas, and Benny Van Houdt. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 3(2):1–35, 2019.

[45] Tim Hellemans and Benny Van Houdt. 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):1–22, 2018.

[46] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[47] Illés Antal Horváth, Ziv Scully, and Benny Van Houdt. 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):1–21, 2019.

[48] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 261–276, 2009.

[49] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619, 2015.

[50] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[51] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[52] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[53] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 690–695, 2016.

[54] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys)*, pages 1–16, 2019.

[55] Kimberly Keeton. The Machine: An architecture for memory-centric computing. https://rb.gy/2xgd7j.

[56] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, 2016.

[57] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)*, pages 1–17, 2020.

[58] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[59] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[60] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning, 2019.

[61] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. XFabric: A reconfigurable in-rack network for rack-scale computers. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–29, 2016.

[62] Stefanos Leonardos, Will Overman, Ioannis Panageas, and Georgios Piliouras. Global convergence of multi-agent policy gradient in markov potential games. *arXiv preprint arXiv:2106.01969*, 2021.

[63] Hwa-Chun Lin and Cauligi S Raghavendra. A dynamic load-balancing policy with a central job dispatcher (lbc). *IEEE Transactions on Software Engineering*, 18(2):148, 1992.

[64] Michael L Littman, Thomas L Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 394–402, 1995.

[65] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[66] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[67] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, page 6382–6393, 2017.

[68] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011.

[69] Sergio Valcarcel Macua, Javier Zazo, and Santiago Zazo. Learning parametric closed-loop policies for Markov potential games. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[70] Siva Theja Maguluri, R. Srikant, and Lei Ying. Heavy traffic optimal resource allocation algorithms for cloud computing clusters. In *Proceedings of the 24th International Teletraffic Congress (ITC)*. International Teletraffic Congress, 2012.

[71] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 270–288. 2019.

[72] Weichao Mao, Tamer Başar, Lin F Yang, and Kaiqing Zhang. Decentralized cooperative multi-agent reinforcement learning with exploration. *arXiv preprint arXiv:2110.05707*, 2021.

[73] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul23. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11).

[74] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[75] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. *In 9th Biennial Conference on Innovative Data Systems Research, (CIDR)*, 2019.

[76] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Computer Communication Review*, 49(1):18–23, 2019.

[77] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[78] David Mguni, Joel Jennings, and Enrique Munoz de Cote. Decentralised learning in systems with many, many strategic agents. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[79] David H Mguni, Yutong Wu, Yali Du, Yaodong Yang, Ziyi Wang, Minne Li, Ying Wen, Joel Jennings, and Jun Wang. Learning in nonzero-sum stochastic games with potentials. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 7688–7699. PMLR, 2021.

[80] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 15–28, 2017.

[81] Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 11(1):6–20, 2000.

[82] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[83] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[84] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE)*, pages 137–148. IEEE, 2015.

[85] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[86] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–17, 2014.

[87] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 361–377, 2019.

[88] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[89] Christos H Papadimitriou and John N Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[90] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[91] 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. Pytorch: An imperative style, high-performance deep learning library. pages 8026–8037, 2019.

[92] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *Proceedings of the Annual Conference on the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 207–218, 2013.

[93] Satish Penmatsa and Anthony T Chronopoulos. Game-theoretic static load balancing for distributed systems. *Journal of Parallel and Distributed Computing*, 71(4):537–555, 2011.

[94] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, (SOSP)*, pages 325–341, 2017.

[95] 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. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[96] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 145–160, 2018.

[97] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 805–825, 2020.

[98] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 401–417, 2016.

[99] Robert F. Rosin. Determining a computing center environment. *Communications of the ACM (CACM)*, 8(7):463–468, 1965.

[100] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 410–424, 2015.

[101] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245, 1991.

[102] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'06, page 18, 2006.

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

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

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

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

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

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

[109] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[110] Richard S Sutton and Andrew G Barto. MIT press, 2018.

[111] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 12. MIT Press, 2000.

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

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

[114] Vasileios Tosounidis, Georgios Pavlidis, and Ilias Sakellariou. Deep q-learning for load balancing traffic in sdn networks. In *11th Hellenic Conference on Artificial Intelligence*, pages 135–143, 2020.

[115] Shay Vargaftik, Isaac Keslassy, and Ariel Orda. Lsq: Load balancing in large-scale heterogeneous systems with multiple dispatchers. *IEEE/ACM Transactions on Networking*, 28(3):1186–1198, 2020.

[116] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[117] Edward S. Walter and Victor L. Wallace. Further analysis of a computing center environment. *Communications of the ACM (CACM)*, 10(5):266–272, 1967.

[118] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

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

[120] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, 1989.

[121] R.R. Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15(2):406–413, 1978.

[122] Ward Whitt. Deciding which queue to join: Some counterexamples. *Operations research*, 34(1):55–62, 1986.

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

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

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

[126] Wayne Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, pages 181–189, 1977.

[127] Se-Young Yun and Alexandre Proutiere. Distributed proportional fair load balancing in heterogenous systems. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 17–30, 2015.

[128] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 265–278, 2010.

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

[130] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the 28th Symposium on Operating Systems Principles, (SOSP)*, pages 724–739, 2021.

[131] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event scheduling for optimizing tail latency in a traditional OS kernel. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 759–772, 2021.

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

[133] Xingyu Zhou, Jian Tan, and Ness Shroff. Heavy-traffic delay optimality in pull-based load balancing systems: Necessary and sufficient conditions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2(3):1–33, 2018.

[134] Xingyu Zhou, Fei Wu, Jian Tan, Kannan Srinivasan, and Ness Shroff. 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):1–41, 2018.

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

APPENDICES

Appendix A

Proofs

A.1 Proof of Potential Property

Claim 2. *The distributed load balancing game defined in §4 is a Markov Potential game with the potential function $\Phi(\cdot)$ defined in Equation (4.1).*

Proof. The proof is based on Theorem 2 in [69].
First, we write the individual payoffs function as the summation of the stage potential function and a term which is non common between all players (condition (i) of Theorem 2 in [69]):

$$u_i(x, a) = \phi(x, a) + \sum_{j \neq i} C_j(a_j).$$

Second, the gradiant of non common term $\sum_{j\neq i} C_j(a_j)$ with respect to state is zero (condition (ii) of Theorem 2 in [69]):

$$\nabla_x \left[\sum_{j \neq i} C_j(a_j) \right] = 0 \quad \rightarrow \quad \mathbb{E}\left[\nabla_x \left[\sum_{j \neq i} C_j(a_j) \right] \right] = 0.$$

Moreover, the payoff is a proper function. A function is said to be proper if: i) $\exists \mathbf{x}$ such that $\mathbb{E}[f(\mathbf{x})] > -\infty$, ii) $\forall \mathbf{x} \rightarrow \mathbb{E}[f(\mathbf{x})] < \infty$. The second condition clearly holds by definition for the payoff function in that it is always non-positive. The first condition holds by setting the $x_{i,r}$ to t = 0 when the cluster starts with zero tasks in the servers' queue.

Finally, we show that the level sets $\{\mathbb{E}[u_i(x_{i,r}, a_{i,r})] \geq B\}_{t=0}^{\infty}$ are non-empty for some scalar B and $\{(x_{i,r}, a_{i,r})\}_{r=0}^{\infty}$ in support of the trajectory induced by the game.

$$\mathbb{E}[u_i(x_{i,r}, a_{i,r})] = \mathbb{E}\left[-\frac{1}{N-1}\sum_{j,k}|x_{j,r} - x_{k,r}| - C_i(a_{i,r})\right] \geq B$$
$$\iff \mathbb{E}\left[\frac{1}{N-1}\sum_{j,k}|x_{j,r} - x_{k,r}| + C_i(a_{i,r})\right] < -B$$

Furthermore,

$$\mathbb{E}\left[\frac{1}{N-1}\sum_{j,k}|x_{j,r} - x_{k,r}| + C_i(a_{i,r})\right] \leq \left[\frac{1}{N-1}\sum_{j,k}\mathbb{E}\,|x_{j,r} - x_{k,r}| + c\right]$$
$$\leq \left[\frac{1}{N-1}\sum_{j,k}[\mathbb{E}[x_{j,r}] + \mathbb{E}[x_{k,r}]] + c\right] \leq 2N\max\left(\{\mathbb{E}[x_{r,k}]\}_{k\in N}\right) + c$$

Assuming that queue lengths are bounded, which is always true in practice, the maximum load remains bounded. This guarantees that the game defined in §4 has a parametric nash equilibrium. Moreover, the distributed load balancing game defined in §4 is a Markov Potential game with the potential function $\Phi(\cdot)$ defined in Equation (4.1)

□

Appendix B

Definitions

B.1 Heavy-traffic delay optimality

Consider a queuing system with N servers. In this system, λ_Σ and σ_Σ^2 denote the mean and variance of the arrival process. Assume that the mean and variance of service time process of server i are μ_i and v_i^2, respectively. Let $\mu_\Sigma \doteq \sum_{i=1}^N \mu_i$ and $v_\Sigma^2 \doteq \sum_{i=1}^N v_i^2$. Moreover, let $\epsilon \doteq \mu_\Sigma - \lambda_\Sigma$.

Definition 2. *A load balancing policy is heavy-traffic delay optimal in steady-state if the steady-state queue length vector $\bar{\mathbb{Q}}^\epsilon$ satisfies*

$$\limsup_{\epsilon \downarrow 0} \epsilon \, \mathbb{E}\left[\sum_{i \in N} \bar{Q}_i\right] \leq \frac{\sigma_\Sigma^2 + v_\Sigma^2}{2}.$$