ML-native Dataplane Operating Systems

Paper #32

Abstract

Performance tuning has never been more critical in modern datacenters, where rapid advances in networking and I/O speeds expose potential bottlenecks in CPUs, memory, and operating systems. Yet, tuning remains notoriously complex, brittle, and opaque, especially at the microsecond-scale latencies demanded by modern dataplanes. We argue that there is a pressing need to unleash the power of machine learning in datacenter operating systems.

We propose *ML-native Dataplane Operating Systems*, a radical design for ML-based performance tuning. Our design embraces real-time, microsecond-scale ML-based adaptation as a foundational principle, treating ML not as a byproduct but as the backbone, to continuously optimize dataplane performance under dynamic workloads and operating conditions. Our preliminary experiments show that a traditional static dataplane OS incurs severe queuing delays (over milliseconds) under dynamic and parallel workloads, whereas our adaptive approach sustains microsecond-scale tail latencies by dynamically tuning OS parameters in response to runtime signals.

1 Introduction

Performance tuning is a long-standing and critical challenge in systems [12, 19]. In the datacenter, it has become even more critical as CPUs become a bottleneck and I/O devices grow faster and more complex. To meet the rising demand for low tail latency, terabit networking and microsecond-scale datapath operating systems are both increasingly widespread [10, 16, 26, 28, 39], and at the same time, highly sensitive to tuning while tuning them remains poorly understood.

Modern hardware has complex performance properties that make it difficult to even model a datacenter server, let alone tune it with any confidence. For example, a recent paper on offloading RPC serialization [30] showed that a complex set of factors dictate when offloading performs better than CPU copies, including cache hierarchies, memory bandwidth usage and the device's address translation hardware. Manually tuning dataplane OSes for each unique combination of application and hardware configuration is infeasible in the face of heterogeneous devices and changing workloads.

Applying machine learning to systems has recently gained significant traction, tackling challenges in areas such as index structures, resource management, and performance tuning for components like ECN in networking [2, 6, 8, 17, 24, 25, 38]. Given the inherent complexity of performance tuning in dataplane operating systems, applying ML to this domain represents a natural progression. However, existing dataplane operating systems are optimized for the lowest possible latency, not tunability, and thus require a radical rearchitecting to work well with ML-based tuning.

Moreover, datacenter workloads and resource usage fluctuate at microsecond timescales [15, 21], prompting operators to colocate multiple tasks per server to maximize CPU efficiency [13, 32, 36, 41]. These tasks also often have diverse performance objectives such as low tail latency [10, 39] or high sustained throughput. These conditions give rise to three unique system-level challenges. First, the system must complete the entire tuning loop – including detecting system conditions, ML inference, and applying tuning decisions – within the microsecond-scale window. Second, multiple dataplane instances should "learn" how to coordinate effectively with one another to meet their respective goals. Lastly, the tuning mechanism must impose minimal performance overhead on the dataplane.

In this paper, we make the case for *ML-native dataplane operating systems*, which are designed with ML-based performance tuning at their core. Existing operating systems are not designed to take full advantage of the powerful tuning capabilities provided by machine learning models. Many OSes, including Linux, have far fewer tunable parameters than they could—or should—relying instead on a multitude of hard-coded "magic numbers", described as "disappointing" by Linux developer Andrew Morton [5]. As a result, there is no existing framework or wisdom to guide the creation of a comprehensive tuning space, particularly for optimizing parameter combinations across multiple OS components.

Moreover, beyond policy-level decisions, OS-level mechanisms are crucial to support ML-driven tuning. For example, changing DPDK configurations often requires re-initializing hardware drivers or restarting the entire application and the associated libOS. Instead, live upgrades or hot patching mechanisms are essential to avoid disrupting users and to enable seamless tuning.

Current operating systems do not naturally support deep integration of ML-based tuning, in part because they are unfriendly to the training and continuous learning of models [34]. It is impractical to anticipate all possible workloads and collect sufficient data to train a model prior to deployment. Learning techniques such as reinforcement learning fundamentally rely on high-quality training data and often struggle with unseen patterns. Unfortunately, such data is not available and is extremely challenging to gather from traditional operating systems, particularly after deployment — precisely when continuous data collection is most needed to address gaps caused by "unseen" workloads.

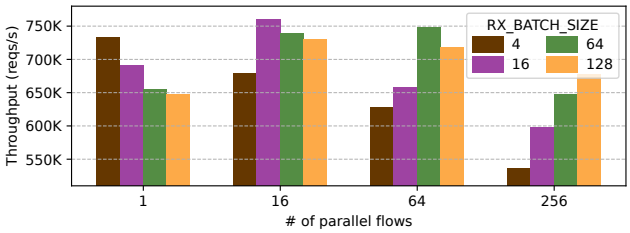

Figure 1. Peak achievable throughput depending on `RX_BATCH_SIZE` and the number of parallel flows. Optimal `RX_BATCH_SIZE` value is different depending on the number of parallel flows.

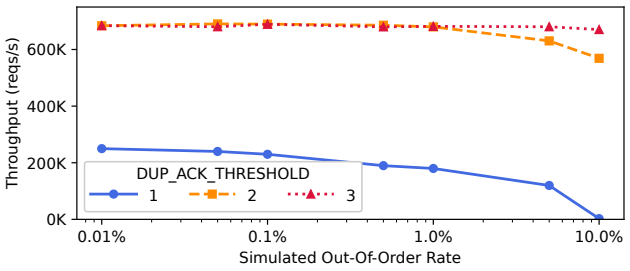

Figure 2. Throughput vs. packet out-of-order rate as a function of the `DUP_ACK_THRESHOLD` value. A smaller `DUP_ACK_THRESHOLD` results in reduced throughput, particularly when the packet out-of-order rate is high.

As famously noted by Ilya Sutskever, "We've achieved peak data, and there'll be no more." We argue that future operating systems must take responsibility for generating high-quality data to enhance model capability. In the domain of dataplane OS, the definition of high-quality data remains elusive. We posit that it must include careful feature extraction, precise mapping of features to performance outcomes, and, most importantly, the ability to collect this data online with practical, low-overhead methods. The current approach typically involves ad-hoc tracing, sampling, and logging added after deployment. This practice often results in poor data quality, excluding critical insights into the root causes of performance changes, and incurs significant overhead, as highlighted in AIOps research [9]. We posit that a standardized format or protocol for data collection is essential, marking a pivotal new consideration in OS design.

An ML-native dataplane OS treats ML-based tuning as the backbone, not as a byproduct. To explore this vision, we present the design, lay out its guiding principles, and identify essential open research questions. We design the operating system to include as many "tuning knobs" as possible across multiple components, enabling a thorough exploration of the performance space. Our system is designed with the following principles in mind: Supporting multi-granularity and cross-component tuning, generating meaningful data, adapting to dynamic workloads, and ensuring reliable tuning.

We prototype our design using an industry-grade dataplane OS, Demikernel [39], and conduct a preliminary evaluation. In the initial stage, we examine and replace "magic numbers" with tunable knobs across five key components in Demikernel, evaluating the effort required. We further demonstrate the potential of ML-driven OS tuning through a simple case study, observing performance gains that exceed our initial expectations. These results highlight promising opportunities and reveal the potential to uncover better performance policies within an ML-native dataplane operating system.

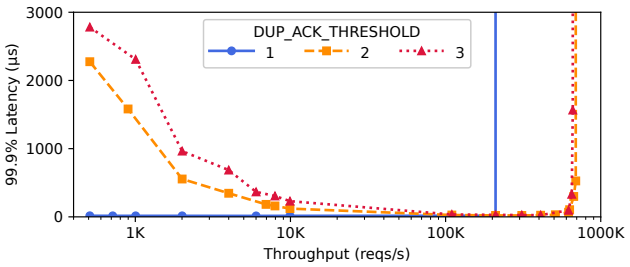

Figure 3. Tail latency vs throughput as a function of `DUP_ACK_THRESHOLD` while simulating 0.1% packet drop rate. Smaller `DUP_ACK_THRESHOLD` makes a significant tail latency reduction, especially at a lower load.

	Network (DPDK)					
	TCP	UDP	Else	Scheduler	Memory	**Total**
# Parms	11	2	6	5	5	**29**

Table 1. The number of existing parameters in Demikernel. The search space of OS parameter tuning is large.

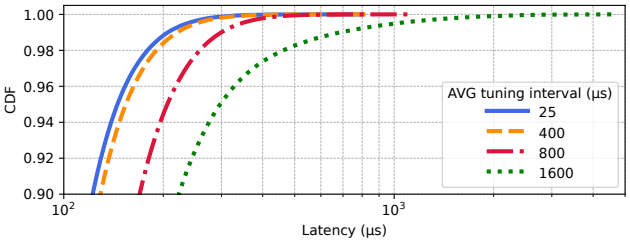

Figure 4. Tail latency depending on the `RX_BATCH_SIZE` tuning interval. Slow tuning can damage performance.

	Tuning domain	Input	ML algorithms	ML deployment	Timing (scale)
ACC [38]	ECN [31]	Net. conditions*	DDQN [35]	Commodity switch	Online (μs)
Credence [2]	Switch buffers	Per-packet states*	Random Forest [4]	Programmable switch	Online (μs)
TCP-RL [25]	TCP IW & CC	Net. conditions*	UCB [11], A3C [23]	Host-local	Online (s)
Configanator [24]	TCP, HTTP	Handshake data	Bayesian Opt. [33], MAB	Host-local	Offline (N/A)
Config-Snob [6]	Net. protocols	Handshake data	Bayesian Opt., MAB [37]	Host-local	Offline (N/A)
Our design	Entire dataplane	Runtime signals*	TBD	Host-local	Online (μs)

* Runtime-dynamic states

Table 2. A comparison of existing ML-enabled parameter tuning systems.

2 Background and Motivation

With recent technology trends, kernel-bypassed dataplane OSes are seeing greater adoption in production data centers, such as Demikernel [39] at Microsoft and Snap [21] at Google. They have a very simple design goal: achieving high I/O performance for μs-scale datacenter applications. However, bypassing the OS kernel alone is not sufficient to meet the strict performance requirements of these applications. Existing dataplane OSes are configured with a large parameter space. Finding the right parameter configurations for optimal application performance is far from a trivial task. To illustrate this, we take Demikernel [39] as a case study and present preliminary experimental results below, using the setup described in §4. While we use Demikernel in our case study, our design does not depend on it and should generally apply to other dataplane OSes as well.

Demikernel reduces I/O latency by employing polling-based dataplane OS stacks. In each polling round, `RX_BATCH_SIZE` determines the maximum number of packets retrieved from the NIC RX queue into the network stack's RX buffer. Then, only the first payload in each flow's RX buffer is delivered to the application. With this I/O design, the optimal `RX_BATCH_SIZE` depends on runtime conditions. One factor is the number of parallel flows, which determines how many payloads are delivered to the application per polling round. If `RX_BATCH_SIZE` is much larger than the flow count, more packets accumulate in RX buffers than those delivered to the application, causing faster saturation under lower load. Conversely, if `RX_BATCH_SIZE` is too small, more polling rounds are needed to retrieve all packets, reducing throughput. To demonstrate this, we show the peak HTTP server throughput across different numbers of parallel flows and `RX_BATCH_SIZE` values in Figure 1. The results indicate that no single `RX_BATCH_SIZE` performs best for all workloads. We also observed that the optimal setting depends on I/O stack design and other runtime conditions, such as application-level overhead per request.

In the TCP stack, `DUP_ACK_THRESHOLD` controls the number of received duplicate acknowledgments (ACKs) before triggering fast retransmission. The optimal `DUP_ACK_THRESHOLD` depends on multiple factors: network condition, flow size, and the optimization goal (tail latency, throughput, etc.). As shown in Figure 2, when more packets arrive out-of-order, a smaller `DUP_ACK_THRESHOLD` reduces the TCP throughput due to more unnecessary packet retransmissions. However, larger `DUP_ACK_THRESHOLD` can also result in worse performance under other workload conditions. Figure 3 shows that, at a lower traffic rate, larger `DUP_ACK_THRESHOLD` values lead to longer tail latency as a result of delayed retransmissions.

Unfortunately, the large parameter space in deployed dataplane OSes makes manual tuning an intractable problem. Table 1 shows that Demikernel's I/O stack itself exposes at least 29 parameters, each with a wide value range. Note that Demikernel is designed for low latency rather than tuning, so it potentially has even more parameters relevant to performance optimization. Additionally, dataplane libraries and drivers (*e.g.* DPDK [1]) used alongside the OS can introduce further tunable parameters. Even for a fixed workload, finding the right parameter configuration requires deep expert knowledge and extensive experimentation. As demonstrated earlier, the choice of optimal parameters is sensitive to workload changes, making manual tuning even less feasible. Moreover, parameter tuning should be performed promptly to adapt to real-time changes effectively. As shown in Figure 4, the system experiences a noticeable tail latency increase if the interval of parameter tuning is larger than 500 μs, even in our simple test case shown in Figure 1. This threshold could be even more stringent in scenarios where system conditions are highly fluctuating.

Prior automatic parameter tuning solutions. Dataplane OSes are not the first class of systems that face this problem. There has already been extensive prior work that leverages machine learning to automatically tune system parameters. However, none of these approaches satisfy our goal of online tuning for dataplane OSes using microsecond-scale runtime signals. Table 2 compares prior ML-based parameter tuning solutions. ACC [38] deploys Double Deep Q-network (DDQN) [35], a deep reinforcement learning algorithm, on programmable switches. The algorithm takes switch queue depth, throughput, and flow information as input to tune ECN parameters. Credence [2] builds a packet arrival predictor on switches for efficient buffer sharing. It implements

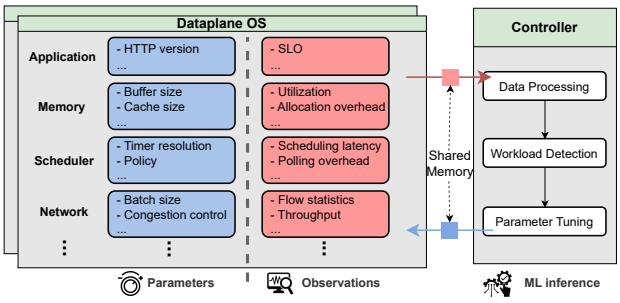

Figure 5. System architecture.

a random forest on programmable switches, trained using packet-level queue length traces. TCP-RL [25] applies reinforcement learning to tune TCP initial congestion window and congestion control algorithm parameters online. It uses UCB [11] and A3C [23] models, takes web server observed network conditions, and adjusts CC parameters at a timescale of seconds. Configanator [24] and Config-Snob [6] use Bayesian optimizations and multi-armed bandit algorithms to tune transport (TCP and QUIC) and application layer (HTTP) parameters during connection handshake.

3 ML-native Dataplane Operating Systems

The idea of ML-native OSes builds on dataplane OSes designed to efficiently process data-intensive I/O operations with microsecond-scale latency. Dataplane OSes are responsible for executing performance-critical I/O tasks, including network processing, CPU scheduling, memory management, and application multiplexing [10, 21, 26, 27, 39]. These systems inherit many functionalities from traditional monolithic kernels and, in doing so, expose a rich set of parameters that govern system performance. Our system is built on top of this architecture and incorporates all of these components into its adaptive tuning framework.

This section presents the design of ML-native dataplane OSes. We begin by outlining the core principles, then describe the system architecture, and lastly highlight open research questions.

3.1 Principles

P1: Supporting multi-granularity and cross-component tuning. ML-native OSes should expose a rich set of cross-component parameters, allowing the system to activate and tune the most relevant subsets at runtime with appropriate granularity. Mechanisms are required to apply tuning decisions without compromising system correctness (e.g., concurrency and ordering) or disrupting user applications.

P2: Generating meaningful data. ML-native OSes should allow flexible, built-in observation points throughout the system to capture high-quality, system-level reasoning data and the corresponding outcomes. These observation points enable offline training, online data collection, and continuous learning. Note that while observation points and tunable parameters may overlap, they do not represent the exact same set.

P3: Adapting to dynamic workloads. ML-native OSes should leverage ML models to detect workload changes, make tuning decisions, orchestrate relevant tunable parameters, and collect feedback.

P4: Ensuring reliable tuning. ML-native OSes should be designed to handle inevitable "unseen patterns", provide fallback options, and ensure system liveness and responsiveness under all conditions.

3.2 Design

Figure 5 illustrates the main system components. Each application runs with a dedicated dataplane OS. Across the essential dataplane components, the OS extensively identifies tunable *parameters* (in the blue boxes) and *observation* points (in the red boxes). Each dataplane OS is linked with a centralized *controller* library via shared memory. This controller is implemented as an independent userspace module that runs in the background, ensuring the control logic operates with only minimal overhead on the dataplane OS.

The system is designed to support a large tuning space and train a model that learns the relative "importance" of parameters, enabling selective tuning at runtime. Parameters can be applied either within a single OS instance or across multiple OSes on the machine. For example, `RX_BATCH_SIZE` or DPDK RX queue size is tuned within an individual OS, while shared resource scheduling policies (*e.g.* CPU core allocation optimized for temporal locality or NUMA-aware placement) are applied across OSes.

The system also includes a large number of observation points that collect essential performance metrics capturing system software dynamics, hardware behaviors, and performance outcomes, with causal information (*e.g.* timestamps) embedded in the data collection process. For example, each OS runtime measures queuing and processing delays as well as workload statistics, while the controller reads relevant performance counters, such as LLC misses, from hardware modules (*e.g.* Intel PMU [7]).

Each dataplane OS continuously writes runtime observation data into the shared memory region. The controller's *data processing* module busy polls shared memory regions using a dedicated core, to collect observation data. Then, the controller spawns ML worker threads using spare cores to run *workload detection* and *parameter tuning* modules. The tuning decision is written into the shared memory regions which are polled by the runtime of each dataplane OS to be applied to corresponding parameters. The data collected during the online tuning process is also fed into offline training, enabling continuous learning and improving model quality

as the system evolves and supports a broader range of applications. The approach for μs-scale, non-interruptive data exchange between dataplane OSes and the controller library is inspired by Caladan [10].

3.3 Open Research Questions

We believe the radical design and ML-native OS vision open up exciting research opportunities.

Q1: Unexplored performance space. No prior work enables the desired level of clarity in performance tuning. In today's compute landscape, can ML uncover untapped performance, like research discovering new proteins?

Q2: Granularity of tuning. From a systems perspective, tuning granularity is critical. Can a model learn that some parameters need fine-grained tuning in networking but only coarse-grained adjustments for CPU or memory, or vice versa? How can we train the model to capture and apply such patterns?

Q3: The microsecond level. In large-scale datacenters such as Google and Microsoft, workloads exhibit fluctuations and bursts at μs timescales [15, 21, 40]. Dataplane OSes also operate at μs-scale [10, 39]. Making tuning decisions within such tight latency budgets at runtime remains challenging.

Inspired by guidelines from Google [20], we propose a hybrid strategy: well-understood and recurring scenarios are handled via offline-trained lookup tables for zero-cost inference, while ML is selectively applied to unseen or less predictable cases for extrapolation. We leverage lightweight ML models – such as Gradient Boosted Regression Trees (GBRTs) [29], Random Forests [4], and Deep Q-Networks (DQNs) [14] – which have demonstrated μs-scale inference when given compact, well-refined features [2, 20, 38]. This strategy maintains adaptability while respecting strict computational budgets, enabling responsive decisions at the dataplane.

Q4: Unified observing protocol. Finally, as ML-native dataplane OSes, we aim to develop a unified observing protocol that different system types – including database systems, microkernels, and microVMs [3, 18] – can adopt. This would allow the tuning interface to be shared across systems, enabling them to benefit from each other and accelerate the ML-native OS era.

4 Preliminary Evaluation

We implement a prototype of our design by extending Demikernel [39]. Our testbed consists of two hosts, each equipped with dual Intel Xeon Gold 6326 CPUs, 256 GB RAM and a Mellanox ConnectX-5 100 GbE NIC, running Ubuntu 20.04. One host runs an HTTP server with the dataplane OS, while the other generates HTTP GET requests using Caladan open-loop TCP load generator [10], with packet arrival times following a Poisson distribution to emulate Google's datacenter workload [22].

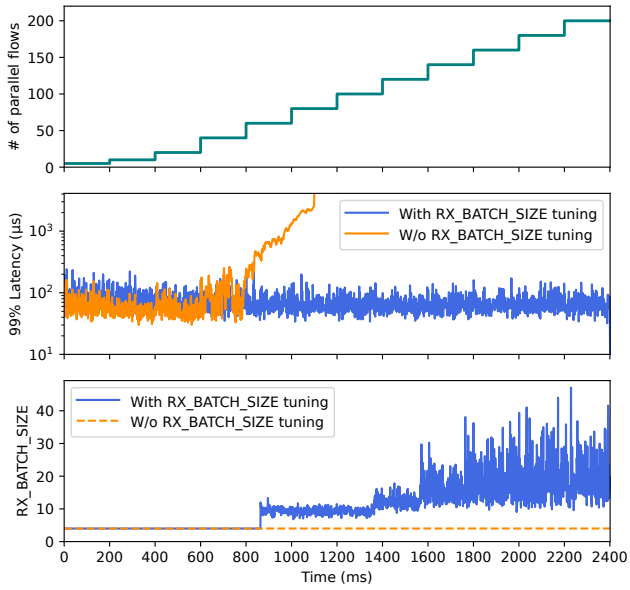

Figure 6. By tuning `RX_BATCH_SIZE` based on runtime OS signals, the system can maintain low tail latency even under dynamic workload conditions.

As a proof of concept, our prototype implements a simple decision tree trained using historical experimental data (similar to that in Figure 1), including the number of parallel flows, `RX_BATCH_SIZE`, and the measured total throughput. The dataplane OS feeds the runtime signal of flow statistics into the decision tree, and dynamically tunes the `RX_BATCH_SIZE` based on the model output.

To evaluate our prototype, we generate 0.6 million requests per second from the client, while gradually increasing the number of parallel flows. Figure 6 presents the 99th-percentile latency alongside the average `RX_BATCH_SIZE`, computed over each 1 ms time window. Without `RX_BATCH_SIZE` tuning, the OS uses a static `RX_BATCH_SIZE` of 4. This setting performs well under moderate load (e.g., fewer than 40 flows), but with more than 60 flows, the small batch size becomes a bottleneck, causing queuing delays to exceed milliseconds. In contrast, our system dynamically selects `RX_BATCH_SIZE` based on runtime signals, enabling the dataplane OS to adapt to changing workloads and maintain tail latency around 100 μs which is comparable to the ideal performance under a constant load.

5 Conclusion

We present the vision and design of ML-native dataplane operating systems, built around four key principles: supporting multi-granularity and cross-component tuning, generating high-quality and meaningful runtime data, adapting efficiently to dynamic workloads, and ensuring reliable, consistent tuning decisions. Our initial prototype illustrates the

benefits of transforming traditionally hard-coded "magic numbers" into tunable parameters, enabling the system to self-optimize under varying conditions. Beyond these early results, we aim to broaden the scope of tunable dimensions, strengthen the feedback loops between components, and explore more advanced machine learning models to answer the research questions that guide this work. Ultimately, our goal is to establish dataplane operating systems as intelligent, self-adaptive platforms capable of sustained performance improvement in complex, evolving environments.

References

[1] [n. d.]. Data plane development kit. https://www.dpdk.org/.

[2] Vamsi Addanki, Maciej Pacut, and Stefan Schmid. 2024. Credence: Augmenting Datacenter Switch Buffer Sharing with {ML} Predictions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 613–634.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[4] Aristide Tanyi-Jong Akem, Michele Gucciardo, and Marco Fiore. 2023. Flowrest: Practical flow-level inference in programmable switches with random forests. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.

[5] Andrew Morton. [n. d.]. Disappointing Control and Magic Numbers. https://lwn.net/Articles/325485/.

[6] Manaf Bin-Yahya, Yifei Zhao, Hossein Shafieirad, Anthony Ho, Shijun Yin, Fanzhao Wang, and Geng Li. 2024. {Config-Snob}: Tuning for the Best Configurations of Networking Protocol Stack. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 749–765.

[7] Intel Corporation. 2017. Intel 64 and IA-32 Architectures Performance Monitoring Events.

[8] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[9] Josu Diaz-De-Arcaya, Ana I Torre-Bastida, Gorka Zárate, Raúl Miñón, and Aitor Almeida. 2023. A joint study of the challenges, opportunities, and roadmap of mlops and aiops: A systematic survey. *Comput. Surveys* 56, 4 (2023), 1–30.

[10] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[11] Aurélien Garivier and Eric Moulines. 2008. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415* (2008).

[12] Brendan Gregg. 2014. *Systems performance: enterprise and the cloud*. Pearson Education.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[14] Yanhua Huang. 2020. Deep Q-networks. *Deep reinforcement learning: fundamentals, research and applications* (2020), 135–160.

[15] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. 2018. {PerfIso}: Performance isolation for commercial {Latency-Sensitive} services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 519–532.

[16] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. 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 '14)*. USENIX Association, Seattle, WA, 489–502.

[17] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[18] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. 2024. Sabre:{Hardware-Accelerated} Snapshot Compression for Serverless {MicroVMs}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 1–18.

[19] Mike Kosta Loukides. 1996. *System Performance Tuning* (1st ed.). O'Reilly & Associates, Inc., USA.

[20] Martin Maas. 2020. A taxonomy of ML for systems problems. *IEEE Micro* 40, 5 (2020), 8–16.

[21] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, Huntsville, Ontario, Canada, 399–413. https://doi.org/10.1145/3341301.3359657

[22] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*. 319–330.

[23] Volodymyr Mnih. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783* (2016).

[24] Usama Naseer and Theophilus A Benson. 2022. Configanator: A Data-driven Approach to Improving {CDN} Performance.. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1135–1158.

[25] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. 2019. Dynamic TCP initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1231–1247.

[26] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[27] John Ousterhout. 2021. A linux kernel implementation of the homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 99–115.

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

[29] Peter Prettenhofer and Gilles Louppe. 2014. Gradient boosted regression trees in scikit-learn. In *PyData 2014*.

[30] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, et al. 2023. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 200–215.

[31] K Ramakrishnan, Sally Floyd, and D Black. 2001. Rfc3168: The addition of explicit congestion notification (ecn) to ip.

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

[33] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[34] Ray AO Sinurat, Anurag Daram, Haryadi S Gunawi, Robert B Ross, and Sandeep Madireddy. 2023. Towards Continually Learning Application Performance Models. *arXiv preprint arXiv:2310.16996* (2023).

[35] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[36] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[37] Joannes Vermorel and Mehryar Mohri. 2005. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*. Springer, 437–448.

[38] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN tuning for high-speed datacenter networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 384–397.

[39] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, Virtual Event, Germany, 195–211. https://doi.org/10.1145/3477132.3483569

[40] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*. 78–85.

[41] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 379–391.