How I learned to stop worrying and love learned OS policies

Divyanshu Saxena
UT Austin

Jiayi Chen
UT Austin

Sujay Yadalam
UW-Madison

Yeonju Ro
UT Austin

Rohit Dwivedula
UT Austin

Eric H. Campbell
UT Austin

Aditya Akella
UT Austin

Christopher J. Rossbach
UT Austin

Michael Swift
UW-Madison

Abstract

While machine learning has been adopted across various fields, its ability to outperform traditional heuristics in operating systems is often met with justified skepticism. Concerns about unsafe decisions, opaque debugging processes, and the challenges of integrating ML into the kernel—given its stringent latency constraints and inherent complexity — make practitioners understandably cautious. This paper introduces *Guardrails for the OS*, a framework that allows kernel developers to declaratively specify system-level properties and define corrective actions to address property violations. The framework facilitates the compilation of these guardrails into monitors capable of running within the kernel. In this work, we establish the foundation for Guardrails, detailing its core abstractions, examining the problem space, and exploring potential solutions.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Operating systems**.

Keywords

Operating Systems, Machine Learning for Systems

ACM Reference Format:
Divyanshu Saxena, Jiayi Chen, Sujay Yadalam, Yeonju Ro, Rohit Dwivedula, Eric H. Campbell, Aditya Akella, Christopher J. Rossbach, and Michael Swift. 2025. How I learned to stop worrying and love learned OS policies. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730384

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HOTOS 25, May 14–16, 2025, Banff, AB, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730384

1 Introduction

The Operating System (OS) is responsible for managing a wide variety of resources for a diverse range of applications, all running together, with each application having unique and dynamic demands. To add further to the complexity of OS tasks, the same OS can be deployed in various scenarios – from datacenter servers to edge devices to end-user equipment. The magic behind this flexibility are the myriad OS policies (and associated interfaces) that have been designed and refined over the years to provide good performance for a general set of applications.

However, simply refining these heuristics through yet more programmer effort, intuition, and observation is not enough for modern applications with tight and strict performance needs. Decades of refinement of OS policies still leaves us with an Operating System that may spend up to 500 ms allocating a huge page [19], that may still starve bandwidth [4], and may idle cores when ready tasks are still available in the runqueue [18]!

Learning – in particular, deep learning – has proven to be a promising tool in dealing with dynamic environments and out-performing human-designed heuristics for a variety of systems policies including query optimization [21, 22], indexing [7, 16], cloud configuration tuning [14, 17], compilation [15], and traffic engineering [23, 28]. OS policies haven't remained untouched either – researchers have demonstrated significant benefits when using learning for OS policies such as I/O latency prediction [12], file readahead [2], and congestion control [1], among others. This has been largely attributed to the ability of deep learning models to use a wide variety of features, infer future behaviors accurately, and tailor decisions to each context better than human-crafted intuitive heuristics designed with generality in mind.

However, there is a reluctance among OS practitioners in using learned models inside the OS kernel. In our own experience and in our engagements with other researchers

[*]Saxena and Chen contributed equally to this work.

and industry practitioners, we identify four main reasons for this reluctance.

First, there might be misbehaviors where a learned policy may take unsafe decisions leading to drastically bad system states. Machine learning algorithms typically optimize a statistical loss function. The loss function formulation is thus critical and it may be hard to encode all bad system states in such a function. Further, deep learning algorithms are seldom able to compute global optima at all inputs, so there might still be data points where the learning algorithm does not ensure optimal behavior. Similarly, unsafe ML behavior may arise due to updates in the kernel (and associated drivers), rendering the training data behind the policy stale.

Second, the opaque-box nature of deep learning models leads to issues in reproducibility and security. It may be hard to reason about why a learned model took a particular decision, and debugging via replicating the exact same set of features in a dynamic system might not be feasible. This leads to practitioners not trusting learned policies in the OS kernel, where reliability is critical. Likewise, an adversarial application could influence the learned model to make bad decisions harming the performance of benign workloads

Third, running a learned policy will inevitably have performance overheads, e.g., to run the model on the critical path, store its weights, and perform online training (if necessary). Priors works address overhead by employing simple models [12] and accelerating in-kernel inference [2, 11, 30] but provide no way for practitioners to assess if inference overhead is justified and to bound performance impact, constraining the use of ML profitably in the kernel.

Lastly, collecting diverse enough features may require extensive instrumentation in the kernel, which is difficult given the complexity and criticality of the kernel. Several systems policies require interactions with the environment in order to train learned models [20], but doing the same within the kernel is challenging.

This paper proposes a framework to alleviate the first three of the above four issues. Our framework helps OS practitioners regulate the decision quality of opaque-box learned models in the kernel and bound their overheads to achieve overall better performance.

It is built on the key abstraction of **guardrails for the OS**: just as highway guardrails provide drivers the flexibility to navigate safely within a roadway while preventing access to dangerous areas, OS guardrails are systems constructs to enable flexible yet safe machine learning policies for the kernel, safeguarding against catastrophic outcomes. The concept of guardrails has been previously proposed in the context of generative AI models [8, 13, 24], e.g., focusing on the safety of conversational agents. In this paper, we describe the unique challenges faced in the OS setting and propose a general framework for designing efficient guardrails for OS decision-making tasks.

We argue that an effective OS guardrail must consist of two components: a specified *property* the guardrail monitors and a *principled corrective action* the system takes when the property is violated. We develop a taxonomy for properties and actions reflecting real-world requirements and practical and effective mitigating actions, respectively. We argue for and propose a guardrail interface that simplifies specification and enables automated synthesis.

2 Background

Learned policies have been demonstrated to perform significantly better than vanilla OS policies *on average* for tasks such as prefetch read ahead [2], predicting I/O latencies [12], congestion control [1], and data placement in tiered memory systems [5, 9] and hybrid storage systems [25].

However, there may be cases where models may cause arbitrarily bad system behavior, e.g., a learned congestion control may lead to a sudden drop in bandwidth utilization and fail to recover from it [29], or a learning-based data placement engine may perform poorly if the workload is write-intensive and has random access pattern [25]. In Section 5, we demonstrate how LinnOS [12], a learned model for predicting SSD I/O latencies, fails to predict slow accesses accurately leading to performance degradation.

A key step in mitigating bad behavior is to *detect* it. This can be applied to specific OS tasks. Examples include the accuracy property for an I/O latency classifier (that classifies accesses between 'fast' and 'slow'): 'Accuracy of the classifier > 90% over a time window of a given size'; or a performance property for a learned memory manager: 'Page fault latencies must not exceed 50ms'. Likewise, system-level misbehavior of a particular subsystem or the OS as a whole can also be detected. For instance, the CPU subsystem may need to follow a starvation-freedom property, e.g., 'No ready task should be starved for more than 100ms'.

In general, violations of the above properties are possible even for hand-coded heuristics, e.g., even in today's kernels, page fault latencies can go to up to hundreds of ms [19]. However, learned OS policies are typically held to stricter standards. This is common for most automated systems: driverless cars may be safer than human drivers (e.g., they may not doze off mistakenly), but are held to high standards due to their autonomy. Similarly, learned OS policies must provide additional confidence for practitioners to move from hand-coded heuristics.

Prior works that have tried to regulate such bad behaviors for learned models either use ad-hoc mechanisms for specific models [1, 12] or provide a high-level interface that still leaves a lot of burden on developers [27]. For instance, Orca [1] is a learned congestion controller that uses Cubic

	Property	Description	Models that need this property
Input	**P1**. In-distribution inputs	Ensure model output used only if inputs remain in-distribution	*All models*. Prolonged sequences of out-of-distribution data may indicate domain shift and require retraining.
Outputs	**P2**. Robustness decisions	Ensure similar inputs yield similar outputs and behavior within a time window.	*Congestion control*. Check if the model is sensitive to noisy measurements.
	P3. Out-of-bounds outputs	Ensure outputs are within legal bounds	*Memory allocation*. Ensure allocation by the model is within available memory.
Behavior	**P4**. Decision quality	Ensure decisions yield good-enough performance	*Cache replacement*. Decisions of the model must yield better hit rates than randomly selecting elements.
	P5. Decision overhead	Ensure inference latency is less than performance gains from policy.	*All models*. Latency and total resource usage overhead of inference should be reasonably low.
	P6. Fairness and liveness	Ensure general system goals are upheld by decisions.	*CPU scheduling*. Check scheduling delay and variance are within bounds.

Action API	Description	Example use
A1. REPORT(function_ptr, system_state, input **)**	Logs relevant system context when the property is violated (e.g., which inputs triggered violation).	Record out-of-distribution inputs (P1) and poor-quality decisions (P4).
A2. REPLACE(old_function_ptr, new_function_ptr **)**	Swaps out a misbehaving learned policy (or function) with a known-safe fallback.	After out-of-bound decisions (P3) or repeated poor quality decisions (P4) disable learned policies.
A3.RETRAIN(model_ptr, input **)**	Retrain model with new out-of-distribution data.	After sensitivity to noisy measurements (P2) or invalid outputs (P3) are detected.
A4.DEPRIORITIZE({function_ptrs}, {priorities} **)**	Deprioritize/kill tasks to free resources or relax constraints.	Out-of-memory killer (P6).

Figure 1: Guardrail properties and actions. Color codes in the left table distinguish the various types of properties. Color codes in the right table illustrates the properties on which the actions apply in the examples - note that these are not exhaustive.

for fine time-scale CC and a learned model that makes adjustments to TCP at slow time-scales. By designing the controller in such a way, Orca is able to capitalize on the benefits of TCP Cubic such as convergence properties, predictable behavior and reduced overheads. However, this technique is strongly coupled with the design of Orca and congestion control; it cannot be extended to other models or for properties of the form described above.

SOL [27] offers a unified API for monitoring and regulating learned agents. SOL's API revolves around developer-defined callbacks for tasks like input validation, output assessment, performance evaluation, and mitigation actions. However, SOL has key limitations that hinder its use for regulating models *in the kernel*. For example, the properties and actions in SOL are tightly coupled with the learned agent. This precludes richer end-to-end system properties (such as ensuring fairness in the scheduler, or good end-to-end memory access latencies) that may be affected by multiple learned agents, and also cannot capture the many rich corrective actions possible. Further, its high-level callback-based API punts a lot of work to the developer, e.g., writing performance assessment functions entails significant effort before a model can be reliably deployed.

An orthogonal approach to detecting bad behavior at run time is to train models that provably satisfy the desired system properties, via techniques such as shielding [3, 31] and learned integrated with verification [10, 29]. However, such techniques often need a model of the system to verify against leading to two challenges. First, developing such models for the OS is difficult. Second, such models are only an approximation and actual system behavior at run time may differ, requiring runtime monitoring of property satisfaction anyway.

3 Guardrails

We propose **Guardrails** that enable a **systematic, uniform and declarative** way of specifying and enforcing constraints on learned models in the OS. Guardrails allow kernel developers to specify **a variety of desired system properties and actions** that can then be **automatically compiled** into monitors that run in the kernel. Guardrails serve as an always-on safety net, preventing the learned policy from making unbounded errors and giving OS developers confidence in deploying such policies in production environments.

The guardrail abstraction consists of:

(1) **Properties:** Declarative expressions of desired behaviors and constraints, such as performance bounds, resource limits, or safety invariants. They define catastrophic behaviors to avoid or expected behaviors to achieve.
(2) **Actions:** Prescriptions for system responses when a property is violated, focusing on regulating the learned model—e.g., deciding when to use, retrain, or replace it or adjusting system parameters to enhance its performance.

We next describe properties and actions for the guardrail abstraction in detail.

3.1 Properties

Properties describe the criteria under which a learned policy is considered "safe enough" for the OS. When a property is violated, it is a signal that something is going wrong with the learned model, and it has to be steered back into the middle of the road. While guardrails may be valuable for any policy (as evidenced by policy failures in current operating systems), we identify three categories of properties specifically for learned policies based on what they are expressed over: (i) the *input state* of a model (e.g., the distribution of requests), (ii) the *model's output* (e.g., predicted values or discrete decisions), and (iii) the *resulting system behavior* (e.g., resource usage, latency, throughput). Specific property types and examples are summarized in Figure 1.

Input properties. For model inputs, the key thing is to ensure that inputs fit within the expected distribution upon which the model was trained (P1). If model inputs change, such as a new set of workloads are run, the model may no longer make quality decisions. This can be done by tracking statistical properties of the input features (range, quartiles, etc.) and periodically ensuring they match training data.

Output properties. For model decisions (output), one key property is to ensure that decisions are robust and not subject to large changes based on small input changes (P2). When this happens, small differences in features can have outsize impact on decisions, which leads to unpredictable behavior. Thus, one property to check would be that a small variance in inputs should not lead to large variance in model outputs. Likewise, suppose a model starts to produce illegal decisions (P3), such as prefetching chunks from a file beyond the memory limit for a process, or scheduling a thread on a non-existent NUMA node. In that case, the guardrail should detect the model is misbehaving.

Behavioral properties. Behavioral properties do not look specifically at model inputs or outputs, but instead look at the system operation for signs that it is working well. As such, they could also apply to traditional heuristic policies. Most important, such guardrails should check overall decision quality to ensure that the learned policy is generally improving over standard heuristics (P4), or that the inference cost of using the learned model is offset by the performance gains (P5). Likewise, they can check general system-level goals like fairness and liveness (P6), for example by monitoring cumulative resource usage and delay by different threads or workloads in the system.

Developers can choose to check for a subset of these properties. Taken as a whole, these three guardrails ensure that if anything goes wrong with a learned policy, it will be promptly detected. We next turn to what to do in that case.

3.2 Actions

Actions allow a system to automatically recover when a learned policy misbehaves, and are key to providing safety. Figure 1 lists a set of actions a guardrail can take when a property has been violated. At the simplest level, one action is to report the violation for offline analysis, which could involve logging information about the violated property, increasing logging levels generally, or recording model inputs and outputs (A1). This response does not correct the situation, and is best used for loose guardrails that are for early warning of problems, such as drifts in input distribution, rather than severe problems like out-of-bounds model outputs.

A second action available with learned policies is to fall-back to existing system policies (A2). Most systems deploying learned policies supplement but do not replace existing ones, and as most OS policies rely on limited history and state, they are often able to start making decisions immediately. When guardrails detect poor decision quality, high decision overhead, or other output problems, a guardrail can disable the learned policy temporarily.

The actions so far described do not address the problem of poor decisions but rather mitigate the impact. A third action possible for guardrails is to trigger retraining on newer data, such as the changed input distribution (A3). We envision offline training, so this is an asynchronous process that must be protected to prevent abuse from malicious processes by intentionally triggering frequent retraining.

While the preceding actions all relate to the model, a final set of possible actions relate to changing the workload and environment (A4). For example, if a policy fails to meet performance goals under a given workload, it could de-prioritize or terminate a low-priority task to release its resources, much as the Linux out-of-memory killer can kill a task to free memory. Likewise, cloud systems may terminate workloads that over-consume a resource [26] to ensure enough resources are available for the remaining workloads. This decision is a bit drastic and is thus best used when falling back to existing policies is not feasible, as it is the only other way to immediately affect system behavior.

3.3 Overall Framework

Adding guardrails to a system is a non-trivial task that nonetheless promises high rewards. To use guardrails, we design an intuitive interface (described in Section 4) with which system developers can add call-outs, specifying what properties to check for and what actions to trigger. In the interface, the conditions verified by properties must be specified. For learned policies, many of these can be determined automatically, e.g., the performance metric to track can be extracted from the reward function. Others, though, require system knowledge and may depend on specific hardware or deployment characteristics, such as reasonable values for system-wide fairness. In these cases, OS practitioners may find it better to deploy guardrails with relaxed properties and automatically tighten the properties based on system behavior.

The provided guardrails are then automatically compiled into 'guardrail monitors' that run inside the kernel, either as eBPF programs or as kernel modules. A key feature of guardrails is that they allow incremental deployment: more guardrails can be incrementally added to check for more properties; or check properties on more occasions or more frequently; or perform additional corrective actions. In what follows, we provide our Interface, discussing the key design decisions and how the interface simplifies guardrail specification and enables automatic synthesis.

4 Interface

We propose a *guardrail* interface built on a formal foundation that enables rigorous reasoning about properties and violations while supporting a comprehensive set of corrective actions required to address violations caused by learned OS policies. Our interface is inspired by Monitor-Oriented Programming (MOP) [6], a runtime verification framework that enables developers to specify properties and associate them with callback handlers for corrective actions. However, while MOP is primarily designed for writing specifications for user-level software programs, it does not address the more intricate requirements of a learning-aided OS which must carefully manage potential issues from system behaviors and diverse learned components.

Listing 1 illustrates the syntax of our guardrail specifications, which are organized around three core concepts: *triggers*, *rules*, and *actions*. Below, we will discuss these concepts - explaining how the interface enables the goals of intuitive, declarative specification that enables automated compilation.

```
⟨Guardrail⟩ ::= ⟨Property⟩ ((⟨Action⟩))+
⟨Property⟩  ::= ((⟨Trigger⟩))+   ((⟨Rule⟩))+
⟨Trigger⟩   ::= TIMER | FUNCTION
⟨Rule⟩      ::= ⟨Expression⟩
⟨Action⟩    ::= REPORT | REPLACE | RETRAIN | DEPRIORITIZE
```

Listing 1: Syntax of Guardrail specifications

4.1 Triggers and Rules

Triggers determine *when* to evaluate the rules. We currently support:

- `TIMER(start_time, interval, stop_time)`: Periodically checks the rules at fixed intervals. Useful for tracking performance distributions and model accuracy over time.
- `FUNCTION(function_ptr)`: Invokes rule checks whenever a specific function (e.g., a learned scheduler routine) is called.

Rules define *what* property should hold. These may be simple predicates (e.g., `latency <= 20ms`) or more sophisticated conditions (e.g., bounding the average error rate of a learned model).

Crucially, our interface *decouples rules from triggers*. This allows greater flexibility in the OS context, where the property to check is specified independent of when and how frequently to check it. For example, our `TIMER` trigger allows systematic sampling in order to regulate the overhead of checking for the property.

4.2 Defining Corrective Actions

While in theory, MOP provides the facility for arbitrary code to run on either the satisfaction or violation of a property, we are not aware of significant work that has studied *corrective actions*. One could argue that this is sensible, as runtime verification tools are meant to be a kind of lightweight verification technique. In this view, violations should simply be reported to the developers. On the contrary, Guardrails help drive the OS. We expect property violations for learned policies to be relatively commonplace (e.g. out of distribution inputs), and as a result, we need to specify how the system should recover.

Importantly, this means that our guardrails may need to interface directly with system resources. For instance, replacing a misbehaving learned policy with a fallback. To this end, we introduce simple API for the actions specified in Table 1 (shown in the leftmost column of the second table).

We expect the library of callbacks to evolve with our experience of guardrails, but we have found these useful in our initial experience and reusable in correcting the property violations described in 3.1. Further, the limited types of actions allows us to define the semantics for each type that simplifies compilation into guardrail monitors, and helps reason about their correctness and crash-free semantics.

4.3 Managing State

A crucial aspect of any runtime monitoring framework is the management of state used to evaluate properties. As mentioned previously, guardrails need to interface closely with system-wide metrics (e.g., average latency, throughput, or error rates) that must be aggregated over time or across many function invocations.

We currently provide a lightweight, global *feature store* accessed via `SAVE(key, value)` and `LOAD(key)` calls. This allows guardrails to maintain counters, metrics, and other persistent data without introducing ad-hoc kernel data structures. For instance, to state "the average page fault latency over every 10 seconds is below 2ms," guardrails must aggregate data from multiple locations in the kernel that can cause a page fault. Relying on local variables, only accessible within specific kernel functions, would force us to intercept every function call and replicate logic across many guardrail instances. Instead, using a shared feature store simplifies data collection and ensures consistent updates.

5 A Guardrail Example

We demonstrate the benefits of guardrails using the example of LinnOS [12]. LinnOS uses a learned model to predict I/O latencies based on the history of I/O accesses. LinnOS helps storage clusters with built-in failover logic such as flash RAID by revoking slow I/O and re-issuing to a replica. A model misprediction could cause an I/O to be submitted to a slow

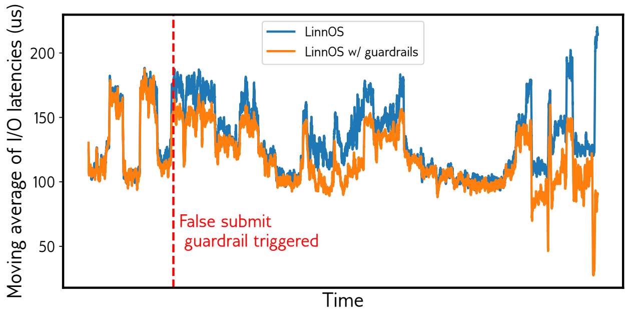

Figure 2: I/O latency moving average. False submit guardrail is triggered mid-way and mitigation is applied. Thereafter, average latency reduces (orange) compared to LinnOS without guardrails (blue).

disk (*false submit*). A high rate of false submits can erase the benefits of a learned model and degrade performance.

To prevent performance regression, we add a guardrail (Listing 2) to detect high rate of false submits (property). When the rate exceeds a threshold, we disable the model and fallback to default behavior (action). As shown in Figure 2, the moving average of I/O latencies improves after the guardrail is triggered and the mitigation is applied.

```
guardrail low-false-submit {
  trigger: {
    TIMER(start_time, 1e9)  // Periodically check every 1s.
  },
  rule: {
    LOAD(false_submit_rate) <= 0.05
  },
  action: {
    SAVE(ml_enabled, false)
  }
}
```

Listing 2: Example guardrail rule enforcing a low false-submit rate in LinnOS.

6 Summary and Discussion

We introduced a framework for safely integrating and utilizing learned policies for decision-making in the operating system, while adhering to constraints defined by kernel developers. These constraints are encapsulated within the abstraction of **OS Guardrails**. We detailed this abstraction, providing examples to illustrate its use cases, and proposed an interface that enables the intuitive specification of guardrails. Our framework represents a significant advancement over existing state-of-the-art frameworks for safely applying machine learning in systems, specifically tailored for OS practitioners rather than general ML safety approaches [8, 24].

Our proposal raises several compelling questions, such as how to support a diverse range of guardrails for learned kernel policies, how to synthesize efficient guardrail monitors, update guardrails at runtime without requiring a kernel reboot, and how to extend guardrails to other learned infrastructure components. There are also several avenues to improve the proposed Guardrails framework. For example, deploying multiple guardrails in the kernel—each monitoring a different property—can create feedback loops, where preventing one violation triggers another, causing the system to oscillate between violation states. Another interesting area for exploration is the potential to improve over trigger-based periodic checking by tracking a minimal set of data dependencies, enabling such properties to be automatically checked *only* when relevant system state changes.

Acknowledgements

We thank the anonymous reviewers for their suggestions that improved this paper. This work is supported by a U.S. National Science Foundation (NSF) CISE "Expedition" Grant Number 2326576.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 632–647, 2020. URL https://doi.org/10.1145/3387514.3405892.

[2] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 94–102, 2021. URL https://dl.acm.org/doi/10.1145/3465332.3470875.

[3] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6172–6183. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/448d5eda79895153938a8431919f4c9f-Paper.pdf.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 177–192, 2022. URL https://doi.org/10.1145/3544216.3544223.

[5] Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. Idt: Intelligent data placement for multi-tiered main memory with reinforcement learning. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–82, 2024. URL https://dl.acm.org/doi/10.1145/3625549.3658659.

[6] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 569–588, 2007. URL https://doi.org/10.1145/1297105.1297069.

[7] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171, November 2020. URL https://www.usenix.org/conference/osdi20/presentation/dai.

[8] David "davidad" Dalrymple, Joar Skalse, Yoshua Bengio, Stuart Russell, Max Tegmark, Sanjit Seshia, Steve Omohundro, Christian Szegedy,

Ben Goldhaber, Nora Ammann, Alessandro Abate, Joe Halpern, Clark Barrett, Ding Zhao, Tan Zhi-Xuan, Jeannette Wing, and Joshua Tenenbaum. Towards guaranteed safe ai: A framework for ensuring robust and reliable ai systems, 2024. URL https://arxiv.org/abs/2405.06624.

[9] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 37–48, 2019. URL https://dl.acm.org/doi/10.1145/3307681.3325398.

[10] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 305–318, 2021. URL https://doi.org/10.1145/3452296.3472936.

[11] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 846–861, 2023. URL https://doi.org/10.1145/3575693.3575697.

[12] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190, 2020. URL https://www.usenix.org/conference/osdi20/presentation/hao.

[13] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, and Madian Khabsa. Llama guard: Llm-based input-output safeguard for human-ai conversations, 2023. URL https://arxiv.org/abs/2312.06674.

[14] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, April 2023. URL https://www.usenix.org/conference/nsdi23/presentation/karthikeyan.

[15] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units, 2021. URL https://arxiv.org/abs/2008.01040.

[16] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, 2018. URL https://doi.org/10.1145/3183713.3196909.

[17] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. Mlos in action: Bridging the gap between experimentation and auto-tuning in the cloud. *Proc. VLDB Endow.*, 17(12):4269–4272, November 2024. ISSN 2150-8097. URL https://doi.org/10.14778/3685800.3685852.

[18] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016. URL https://doi.org/10.1145/2901318.2901326.

[19] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. CBMM: Financial advice for kernel memory managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 593–608, July 2022. URL https://www.usenix.org/conference/atc22/presentation/mansi.

[20] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, ravichandra addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Dr.Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/f69e505b08403ad2298b9f262659929a-Paper.pdf.

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: a learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019. ISSN 2150-8097. URL https://doi.org/10.14778/3342263.3342644.

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *SIGMOD Rec.*, 51(1):6–13, June 2022. ISSN 0163-5808. URL https://doi.org/10.1145/3542700.3542703.

[23] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. DOTE: Rethinking (predictive) WAN traffic engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, April 2023. URL https://www.usenix.org/conference/nsdi23/presentation/perry.

[24] Traian Rebedea, Razvan Dinu, Makesh Sreedhar, Christopher Parisien, and Jonathan Cohen. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails, 2023. URL https://arxiv.org/abs/2310.10501.

[25] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. Sibyl: Adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 320–336, 2022. URL https://doi.org/10.1145/3470496.3527442.

[26] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015. URL https://dl.acm.org/doi/10.1145/2741948.2741964.

[27] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. Sol: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–634, 2022. URL https://doi.org/10.1145/3503222.3507704.

[28] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. Teal: Learning-accelerated optimization of wan traffic engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 378–393, 2023. URL https://doi.org/10.1145/3603269.3604857.

[29] Chenxi Yang, Divyanshu Saxena, Rohit Dwivedula, Kshiteej Mahajan, Swarat Chaudhuri, and Aditya Akella. C3: Learning congestion controllers with formal certificates. 2024. URL https://arxiv.org/abs/2412.10915.

[30] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 414–427, 2022. URL https://dl.acm.org/doi/abs/10.1145/3544216.3544229.

[31] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 686–701, 2019. URL https://doi.org/10.1145/3314221.3314638.