Machine learning in the Lower Realms of the Operating System – A Review

1st Vahid Mohammadi Safarzadeh
Research and Development Section
Takin Pardazesh Khayyam Company
Ahvaz, Iran
vahid.msafarzadeh@tpkai.com

***Abstract*—Artificial Intelligence has been integrated into the operating system with applications such as voice or image recognition. However, this integration is only in user space. Here, we seek methods and efforts that exploit AI approaches, specifically machine learning, in the OSes' primary responsibilities. We provide the improvements that ML can bring to OS to make them more trustworthy. In other words, the main question to be answered is how AI has played/can play a role directly in improving the traditional OS kernel main tasks. Also, the challenges and limitations in the way of this combination are provided.**

***Index Terms*—Machine Learning, Operating Systems, Kernel Space, User Space**

I. Introduction

Artificial Intelligence and its famous subcategory, Machine Learning, is the current trend in computer science society. It has affected many scientific and technological applications such as medicine, science, and industry. However, it is interesting that OSes' low-level operations as the basics for all computing jobs have remained almost untouched by this revolutionary science. OS algorithms seem very similar to that of twenty years ago with no sign of intelligence in its current meaning; learning and adapting. Although AI does not guarantee an improvement in every application and also the idea of applying it to any traditional procedure and expecting miracle seems naive, the origins of uncertainty and unpredictability that OSes are facing now can be reasonable for AI practitioners to see them as problems to be solved via the available tools of AI. At first sight, mechanisms such as memory management and process scheduling are the playgrounds that AI can help OS with its current procedures. However, several challenges need to be addressed. OSes are programs that resulted from many years of development. They must remain as light-weight as possible while handling the most critical job in a system in the most deterministic and flawless manner. Its actions also have many parameters, consequences, and players. Therefore, in OSes, as an automated and deterministic mechanism, AI can have a role in ways that do not compromise the correctness of the system [1]. Here, our focus is on the kernel space applications of AI in OS. Algorithms used in OS design are either deterministic or heuristic. The heuristic ones are mainly more monitored trial and error methods. For example, in resource scheduling, they contain monitoring the performance changes of the system while increasing/decreasing specific resources for a process [2]. The overwhelming wave of ML computational methods is affecting the lower level of computers. Several concepts such as learned data structures (Indexes) [3], learned database systems (for optimizing queries) [4], learned indexes [5]–[7] , and learning memory hits pattern [8] are introduced, recently, which shows the applications of AI into the low-level world of computation. The main feature of ML is the ability to exploit experiences of the past for future decisions. Searching in configuration spaces and predicting the system's future states, timings, or sizes are challenges that OSes are performing continuously. These operations become even more crucial in applications such as storing systems, web servers, High-Performance Computing systems, clouds, and real-time applications. In this paper, along with analyzing some of the challenges of using AI to enhance OS operations, previous efforts in this area are analyzed. Linux's open-source nature makes it a suitable platform to apply AI methods on an OS's different tasks, although it can not be easily implemented. There also have been efforts to make hardware architecture more intelligent using ML. For example, in [8], the prefetching operation (in which hardware predicts memory accesses before the request by software based on past history) is modeled as a sequence prediction problem. They reported a solution to this challenge using Long Short-Term Memory (LSTM) Recurrent Neural Networks. However, we do not encounter such integrations in this paper.

The kernel, services, and Shell constitute an OS. The kernel is the central part, and other parts can be seen as tools to interact with the kernel and sending requests to it. The primary responsibilities of OSes are managing software and hardware as well as providing system services. Some of these services are fulfilling users' (processes) requests such as I/O requests, file-system manipulation, or inter-process communications. Others are for increasing the system's efficiency. These services are mainly related to sharing resources among processes or threads using resource allocation algorithms. With resource, we mean CPU cycles, main memory, file storage, or I/O devices [9].

Another important job of OS (and hardware) is caching. With the increasing popularity of cloud computing, some approaches towards programming have been changed. One of the most important changes was using micro-services to write

applications instead of the traditional monolithic manner in which all parts of the program were in one piece. Breaking programs into micro-services have provided new challenges to OSes in handling the requests of these services. Dealing with the massive amount of micro-services demands can not be done optimally with traditional methods [2].

Machine Learning is producing programs to optimize a performance criterion using data or past experiences [9]. Each learning application is divided into two parts: Training to produce a model and Inference based on that model. The first part is more resource-demanding than the second one, and it also takes more time. There are a considerable amount of publications about utilizing ML methods in different fields. Besides, a subcategory of ML, Deep Learning, has gained even more attention in recent years. In this paper, we investigate some aspects of OSes from an ML point of view. Then, we analyze some efforts and reported efforts in the literature in integrating ML with OS. Finally, we encounter some challenges and possible solutions. The conclusion and future of the idea of embedding ML into OS come in the last section.

II. Origins of Uncertainty in Operating System

OS designers try to make their systems as predictable as possible. By predictability, we mean the system's ability to evaluate each task's timing and resource properties. One smart idea is for OS to provide the exact amount of resources that every algorithm requires when it is called to perform. This behavior will optimize the load on the OS (server) and reduces the amount of energy that it needs. Such predictability, on the other hand, also may have security drawbacks because it simplifies the way that malware designers can communicate with OS [11] [12]. Therefore, any approach (including ML approaches) should consider such a threat to the final system.

There are some sources of uncertainty in OSes like the uncertainty in reading timestamps resulted from Linux Kernel overhead on which the load of the system and CPU clock speed has direct impact [13]. Although such uncertainties may not be harmful in desktop or even traditional server usages of OSes, despite the energy deficiency that they cause, they are crucial in real-time and embedded OSes.

The procedure of sharing resources in OSes during which so much interference (allocation and deallocation of resources) occurs is another reason for unpredictability in OSes. Predictable latency is an important issue in resource management in cloud computing [2], [14]– [17].

The nature of multiprocessor architectures is associated with unpredictability. Utilizing a particular type of memory access is an example of resource sharing that increases unpredictability in OS. There are two types of shared-memory multiprocessors: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) [18]. In NUMA systems, each processor has special (efficient) access to a memory section, called a NUMA node. For each process, memory can be allocated locally (from the local node) to the corresponding CPU [19]. This allocation process is called NUMA placement. If the local node can not fulfill the memory request, kernel (Linux) will seek the remote options [20]. NUMA placement is a source of uncertainty, and some works such as [21] has proposed ML models to predict its influence on applications performance.

Other causes of unpredictability, as described in [18], are pipeline optimization, cache interference, which increase memory traffic, and the NP-hard problem of scheduling in assigning tasks to processors.

Also, as mentioned before, many OS algorithms consist of heuristics. For example, when all the page frames in memory are allocated, the kernel must decide which old pages can be freed and prevent them from being used by new processes. To reach this goal, heuristics like Least Recently Used (LRU) are being used [22]. However, still designing an acceptable page frame reclaiming method is a trial and error job. Thus it becomes a situation where the system becomes unpredictable; therefore, a perfect candidate for getting help from AI techniques such as "Reinforcement Learning" [23]. In the next section, we investigate how ML can be beneficial to improve OS tasks.

III. How OSes Can Benefit from ML

As mentioned above, several aspects of OSes are the candidates to apply AI, particularly ML, to enhance performance. Transforming OS's deterministic nature into a more flexible and dynamic form can maximize the exploiting of hardware and energy, for example, by reducing the idle times of the CPU. One of the features of ML is the ability to use previous experiences for future predictions. In this case, OSes with a massive amount of processes (threads and services) that continuously start and stop running can benefit from the previous execution behavior of the processes to manage them more optimally, for example, via allocating resources to them more intelligently according to their patterns of usage.

OSes, along with other systems like compilers, are full of heuristics that are hard-coded into their programs. Such heuristics are written in a concrete way and have not the capability to adapt to changes in usage [10]. This adaptation requires mechanisms to use the previous knowledge and patterns of users' behaviors to find the best configurations for timing operations, such as thread scheduling, memory paging, and the frequency of flushing buffer cache. Additionally, determining the optimal sizes for different containers such as buffer cache in storage caching is another operation that can be improved utilizing ML. In addition to learning the optimal configurations for timing and sizing activities, an OS can learn how to allocate memory spaces to applications or hard disk space to files in different situations, which is traditionally done according to predefined policies. Scheduling is one of the essential tasks of an OS, and it also has been one of the well-researched and implemented areas in Artificial Intelligence literature. OS schedules the accessibility of CPU to process threads, and its goal is to distribute the CPU time fairly and optimally among all threads running in the system. Other tasks, such as scheduling different network and storage queues, are of the same type. There are deterministic algorithms defined in OSes

to manage scheduling [1]. Applying AI methods to learn the CPU usage pattern by different processes and applications, we may achieve better and more adaptive scheduling mechanisms.

Also, Since working with OSes is a continuous activity and is in direct contact with humans to fulfill their tremendous amount of requests, online approaches seem beneficial. For example, in space allocation tasks, exploiting Reinforcement Learning is a proper method of ML. This method can be utilized in a way to improve the time, energy consumption, and performance penalties that the system suffers in case of wrong decisions [1]. In the next sections, we provide some efforts reported in the literature to apply ML methods to enhance OS tasks.

A. ML in I/O Scheduling and Latency Management

For making I/O devices (e.g., disks) more compatible with modern processors, I/O schedulers must be in their most optimal form to eliminate the processing requests' overhead. Therefore, I/O scheduling is an important task of OSes in which AI can be used. In [24], a self-learning I/O scheduling scheme is proposed to classify different types of requests or workloads in run-time. To make such scheduling decisions, in the data collection phase, they gathered request features, including types and sizes of the requests, number of processes, and inter-request distances between current and previous requests. Also, workload features, like the number of reads/writes, average request size, and the average number of processes, are collected. They found that Support Vector Machines (SVM) classification results produces the lightest overhead. The whole ML system was implemented in Linux kernel 2.6.13 by modifying kernel I/O schedulers. However, the details of implementation are not provided. They found online learning more adaptive than offline learning.

Using their own in-kernel ML library, KMLIB, researchers in [25] used a regression model to early-reject I/O requests that have a low chance of not meet the deadline. Learning data collected by producing random read and write operations with four threads on a 1GB data set by running an I/O tester called FIO [26]. They modified the "mq-deadline" I/O scheduler in Linux Kernel 4.19.51 and adopted their ML library. The thresholded regression model's accuracy indicating whether an I/O request misses the deadline or not was 74.62%, which reduced the overall latency by 8%.

In a continuation of previous works like [27]– [29] which were based on some heuristics, the LinnOS introduced in [15] was an effort to deal with the problem of high latency in flash storage and SSDs by possessing a model that can learn the behavior of the storage device. It can inform client applications about the anticipated per-I/O speed that the storage can provide for their current requests. LinnOS contains a computationally light-weight neural network to reduce the overheads of the learning and inferring phases. Each incoming request is classified either as a slow-speed or fast-speed. In their online approach, if the latency is fast-speed, it will be passed to the storage, otherwise, with a slow-speed latency, the application will be informed, and the request will be denied. In this case, the application tries another storage node. The NN's input features that are extracted from the current and recent I/O requests are:the number of I/Os in the queue when the new request arrives, the latency of the four most recently accomplished requests, and the number of pending I/Os in the time of arrival of that 4 I/Os. The output is the inference of the speed of the I/Os (slow or fast). Because of using SSDs, data gathering was not an issue. They collected data in busy-hours to achieve more general and richer training data. The NN had three layers with linear neurons. In LinnOS, the training phase is done in the user space using TensorFlow [30] then the weights are sent to NN running in kernel space after transforming them to integer values to compensate for the lack of support for floating-points in the kernel space. The NN is implemented in the block layer part of the Linux Kernel and needs 68 KB of kernel memory.

B. ML in Scheduling

Scheduling algorithms are the heart of any OS. Their purpose is the fairness of time and memory allocation among all processes while the system operates optimally, and the processes finish their jobs as soon as possible. For example, this fair distribution of time among processes is done by Linux's Completely Fair Scheduler (CFS) in the Linux kernel. However, CFS has limitations in multicore environments where it becomes so complicated [31] . In [32], an ML method is used to balance the Linux kernel load on a multi-processor computer. They modified a kernel function and embedded their Multi-Layer Perceptron (MLP) model with three layers for load balancing decisions. The forward phase of the MLP was implemented in C and contained floating-point computations. However, the data collection phase was implemented in a two-way fashion between kernel and user space using some tools explained in their paper. The training set contains 500,000 records resulted from calling load-balancer in different levels of workloads. The same amount of data also was collected in different CPU load averages. Fifteen input features, including the combination of Idle time of the target CPU, NUMA node numbers, and running time of the process per core, were collected. However, in addition to the extra load of the MLP, they did not see any noticeable difference between the original Linux scheduler and the ML-based one.

The increasing usage of Deep Learning methods raised concerns about the Quality of Services of DL-based applications. For example, the CFS can not effectively handle the resulting excessive memory traffic cased by requests for DL-based services [33]. New strategies based on ML should be considered for this situation. This is interesting because, in this approach, ML can be used to enhance ML services. Many deep learning methods and platforms are using server-less computation, where micro-services play the computation role. With a considerable inclination towards micro-services on cloud in AI-based applications, particularly Deep Learning methods, resource scheduling among many micro-services, therefore the Quality of Service on a server is becoming more challenging for traditional schedulers designed for OSes [2]. Here the

challenge is to find the most optimal allocation of resources to different micro-services so that the QoS maximizes. This problem is a searching problem. The resources to be allocated among services include CPU cores, main memories, cache, and bandwidth. The number of micro-services fluctuates, and also, these services are computationally heavy and sensitive to the reduction of resources during their run-time. The traditional forms of resource management and scheduling of OSes are not adapted to the heavy computation needs of deep learning applications.

The OSML scheduling mechanism, introduced in [2], attempts to reach the best QoS for micro-services. It uses ML to find the optimal amount of resources (cores and cache) that any micro-service requires (OAA: Optimal Allocation Area) with which OS understands that extra resource is not needed by the service. It also prevents micro-services from facing a sharp reduction in their QoS due to loss of resources preempted by CPU for serving other micro-services (RCliffs: Resource Cliffs). The models are designed on TensorFlow, and the whole mechanism runs in the user space. They used four three-layered Multi-Layer Perceptrons (called Model-A, Model-B, and their shadows) and a modified version of Deep Q-Network [35], [36] (called Model-C). The Reinforcement Learning approach in Model-C was used to correct the wrong decisions in what Model-A and B propose for resource management. OSML acts as an alerting mechanism to the kernel scheduler.

In [34], the authors reported an experiment in which they exploited ML methods to predict CPU burst times for a process. Several CPU-bound programs, including matrix multiplication, sorting programs, recursive Fibonacci number generating programs, and random number generator programs, were analyzed. They added two system calls to the Linux kernel for providing interoperability between the kernel space scheduler and the Decision Tree inference mechanism in the user space. One of the system calls was responsible for taking the Special Time Slice (STS) classified by the C4.5 decision tree and set the time_slice variable of the process descriptor in the modified scheduler at the kernel level. Another system call also was used to take the "time-rewarded" process back to the normal condition to be allocated the time slice according to the system default task_timeslice() procedure.

C. ML in Cache Management

There are several caches in a computer system, such as CPU cache, web cache, file system cache. Each cache reduces the waiting time of the faster device for receiving data from the slower device. Another challenging job of an OS is to handle the caches (in virtual memory or file systems) in order to know what data should be cached, how much time the data should remain in the cache always to maintain only data on immediate demand in the cache and remove others [37] [1]. Caches always try to maintain more data to help processes run faster [9]. However, the size of caches and the cache hit rates are essential criteria in the system's efficiency. Therefore, keeping the cache's size as minimal as possible while reducing the amount of removing operations and maximizing the cache hit rates are trade-offs in caching procedures. Learning the patterns of data usages by different processes or users can help the OSes to pick the data better to be evicted from the cache.

How a caching procedure operates at the OS level is currently hard-coded in the kernel, based on predefined assumptions about data and users' behavior, while the workloads in the caching are not constant and face many fluctuations. Since caching occurs in different computer system levels, it is regarded as a "caching problem" and is worked on in other researches as a general problem to be solved.

For example, in [9], the authors investigated the improvement in online cache management by applying reinforcement learning based on the reward and impact that a caching decision has brought to the system's performance. As authors reported, along with a better adaptation to workloads, the approach can provide a higher hit rate while minimizing the memory space needed. The method was implemented using TensorFlow. Although its results may be convincing for networking, it is mainly designed for user space applications, and the challenges for implementing in kernel space must be considered.

Other works such as [38] also looked at this as a general problem and only provided ML-based approaches from this point of view. On the other hand, some researchers tried to solve this problem at the hardware level [39] [40]. For example, in [40] researchers used an LSTM-RNN-based cache replacement mechanism, and using the insights from that, they could create a lighter SVM-based architecture at the hardware level to solve the problem of cache replacement. Their method gained better performance than previous heuristic methods such as LRU. Consequently, it seems that applying ML algorithms in OS level cache management still needs more efforts from the researchers.

D. ML in Malware Detection

There are several works in which ML is used to identify malware by monitoring the system calls that they use. For example, in [41], for Android systems, they used system calls of and executable file as features since they demonstrate how program communicate with the kernel. During generating their data set, they also weighted the system calls mostly invoked by each application to boost the discrimination method. Several classification methods were utilized to investigate whether an executable file is a malware or not. Although the approach showed a noticeable performance in identifying malware apps, the vulnerability against malicious learning data remains a severe challenge in this area. We will explain this issue in the following sections.

Also, increasing the security of devices that are not connected to the Internet or have no definitive way to remain up-to-date through getting security patches is a serious issue, particularly in embedded OS. Using ML as a dynamic way of controlling the behavior of processes and requests by distinguishing the system's expected behavior from the anomaly

may help them stay secure on their own. In [42], a host-based, run-time anomaly detection mechanism for Linux OSes is proposed to increase the firmware's security in embedded systems. Their mechanism consists of three parts in both kernel and user spaces. Deep Learning models, in their Exein ML Engine (MLE), were trained to learn the normal behavior of the systems' processes.

They categorized processes into several classes and put a tag identifying that class into the corresponding executable files during the firmware build. Then, for each class, an ML model was designed. MLE is a user space procedure with Convolutional Neural Networks trained based on the processes' behavior at the kernel level.

"Anomaly score" as the model's output is the value used to indicate how unusual a process is acting. If a process is labeled as malicious during execution time, another module, called Exein Linux Security Module (LSM) that works in direct contact with the kernel (kernel level), will be informed. Every system calls of a monitored process during its run-time is hooked and sent to LSM via a kernel module: The Exein Linux Kernel Module (LKM). Then LSM will extract each call's data such as file descriptors, name, and paths, inode attributes, memory information, and permission attributes. This information will be sent to the MLE to label the behavior of the process again via LKM. LSM will be informed of the MLE decision and performs accordingly to trust or distrust the process. Therefore, the LKM acts as an interface between the MLE in user space and the LKM in kernel space.

Such a malware detection procedure adds a significant computational load on the OS, as the authors admitted. However, this load is mostly coming from the process monitoring part and the information sent and received by the three modules, which was alleviated by shrinking the whole process data to limited "snapshots" or only monitoring processes that communicate with the network as the main entrance for attacks. The learning phase of ML is done offline (before building the embedded firmware); however, CNN's inference load added to the OS also should be considered.

IV. Complexities in Using ML in OS

In this section, we encounter some challenges in using ML models in the OS environment. Possible or tested solutions in the literature also are analyzed. Here, we tried to add different aspects to those stated in [1].

A. Implementation Constraints (in Kernel Space)

The particular challenge in manipulating the kernel space mechanisms is limited access to libraries and programming languages in this space. One solution is to make a two-way path between the kernel and user space to benefit from the user space's resources, which is not an optimal approach. For example, The main problem with the two system calls used in [34] was extra transitions from user mode to kernel mode, and this is even more considerable if the learning method is online (adaptive) and needs to be trained regularly based on the new events in the system. The first system call was created due to the lack of enough flexibility in the kernel space, and it is vital to consider the overload of such system calls in the overall performance evaluation of the system benefiting AI. Another solution is to implement the required tools in the kernel space.

KMLib, the ML library introduced in [25], contains the math functions implemented from scratch to be used in the kernel space, mainly to deal with the inaccessibility to floating-point math functions at this level. Based on the common preference of tensor computations in major deep-learning libraries such as TensorFlow or PyTorch [43], KMLib also used such approach but obviously in a more constrained and less resource consuming manner. KMLib can operate in two modes: kernel mode and kernel-user memory-mapped shared mode. They performed the floating-point calculations in a code block started by kernel_fpu_begin and ended by _fpu_end macros in the kernel mode. Nevertheless, the code in this block must be small because of the overhead that it adds to the computations.

In contrast to the work in [25], researchers in [32] took a fixed-point approach in which a fixed amount of bits are for storing integer part, and the remaining bits are to store the decimal part of the number. Computation with such numbers is similar to that of integers.

B. Data Shortage and Data Gathering

Another issue in ML-based modifications in OS's low-level activities is related to data. The question is how the data should be gathered to be general enough for multi-purpose OSes' jobs. In [34], since the goal was to predict the best STS for each program, 84 data instances were generated by setting different STSs for five programs (with CPU-bound processes) and finding the best STS the one that minimizes the TaT (Turn-around-Time) of the processes of each program. Each data's features are several static and dynamic attributes of a process, mostly the sizes of different tables such as the hash table size and the program's size. A Genetic and an Exhaustive search are used to find the best set of features and the corresponding STS with which the process has the most efficient execution behavior.

Also, in online approaches collecting data is done by watching the system's state, then train a model regularly, which adds up more load on the system as we saw in [42].

C. Many Parameters

Each OS contains hundreds to thousands of processes running at the moment and many resources managed by the kernel. Every modification in one part of the kernel must take into account other parts. For example, as in [34], the only evaluation criteria was the reduction of the TaT of one program being tested. However, the consequences of such a change in other parts of the OS are not evaluated.

D. Computational Needs of ML Inference and Learning

Every AI algorithm also takes several computing resources. In the case of learning algorithms, if we assume the training

phase as a separate procedure from the inference phase (which is not in online learning methods), still the inference phase takes some time to produce the results. Generally, to achieve a more reliable and more data-oriented application, we need a considerable amount of data and, consequently, more complicated learning models like those introduced in Deep Learning literature, which also takes more time to infer. However, other DL approaches can be tried, such as few-shot learning, [25] in which the training and inference time and the amount of data can be reduced.

E. Security Threats

Penetrating the isolated and tightly controlled environment of the kernel by outside data motivates malicious users to feed data that can mislead the learning models and, consequently, jeopardize the OS's logical operation. There are recommended solutions, such as using separate ML models for different applications. However, concerns remain for side-channel attacks [1].

In [34] as authors admitted, a simple threat was that their method did not take the security holes that are embedded into the kernel by their method since there were not any precautions to prevent a malicious user from creating a process that requests the maximum amount of STS (special_time_slice) for itself that can have excessive access to CPU.

Works such as [25] via implementing the whole ML into the kernel can eliminate such threat from the outside world with the penalty of more limited capabilities for making more powerful models.

Additionally, every combination of basic kernel tasks with AI methods must consider the overloads that extra security precautions can cause.

F. Interpretability (Explainability)

The ML methods' black-box nature makes their results unexplainable not only for the end-user but also for the experts and designers. Explainable AI and recently with larger and more complex learning models, explainable DL, have become an important topic in the literature [44] [45]. Therefore, in OS designed to have high predictability (in its traditional meaning and despite the potential vulnerability) and every response is based on a definitive algorithm, probing ML methods with little or no interpretability requires a change of perspective.

G. Conclusion and Future

In this paper, we reviewed some recent works on integrating ML methods into the operating systems. According to these works, we also mentioned some challenges and possible solutions in reaching the idea of "Intelligent OS." As we listed, researches are done in utilizing ML for enhancing the main tasks of an OS, such as process scheduling, memory allocation, I/O and cache management, and malware detection. These methods were implemented in user or kernel spaces or both. In user space, designers have access to tools such as TensorFlow to train powerful models, while the overhead was tremendous. On the other hand, in-kernel methods must cope with the computational limitations.

Security was another issue in methods that use both spaces. Security was also a problem in using user' (processes') data to create a data-driven operating system because of the threat of malicious users.

Now, the question is: is it better to transform the operating systems to become more acknowledging toward AI methods, for example, by providing richer programming libraries in the kernel? or try to solve the above challenges. Is it worth to change OSes in this way? Or we should stick to the deterministic nature of them and let the under-world of the kernel space remain as explainable as possible? Learning methods produce results that can not be explained, and there are efforts to make them as interpretable as possible to become more reliable. Although predictability is desirable in operating system design, explainability and interpretability of OS actions are also important. Making operating systems more thoughtful also will end in more computation for tasks that are already done with trial and error but faster. Dealing with these trade-offs requires lots of research and implementations.

REFERENCES

[1] Y. Zhang and Y. Huang, "'Learned' operating systems," Oper. Syst. Rev., vol. 53, no. 1, pp. 40–45, 2019.

[2] S. T. Report and L. Liu, "QoS-Aware Resource Scheduling for Microservices: A Multi-Model Collaborative Learning-based Approach," arXiv, 2020.

[3] P. Ferragina and G. Vinciguerra, "Learned Data Structures," in Studies in Computational Intelligence, vol. 896, 2020, pp. 5–41.

[4] T. Kraska et al., "SageDB: A learned database system," 9th Biennial Conference on Innovative Data Systems Research, 2019.

[5] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proceedings of the ACM SIGMOD International Conference on Management of Data, May 2018, pp. 489–504.

[6] Y. Wang, C. Tang, Z. Wang, and H. Chen, "SIndex: A scalable learned index for string keys," in APSys 2020 - Proceedings of the 2020 ACM SIGOPS Asia-Pacific Workshop on Systems, Aug. 2020, pp. 17–24.

[7] J. Ding et al., "ALEX: An Updatable Adaptive Learned Index," in Proceedings of the ACM SIGMOD International Conference on Management of Data, Jun. 2020, pp. 969–984.

[8] M. Hashemi et al., "Learning memory access patterns," in 35th International Conference on Machine Learning, ICML 2018, 2018, vol. 5, pp. 3062–3076.

[9] A. Silberschatz, G. Gagne, and P. B. Galvin, Operating System Concepts, 8th ed. Wiley Publishing, 2011.

[10] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, G. Zlokapa, and I. Science, "Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel," Oper. Syst. Rev., no. May, pp. 1–7, 1990.

[11] R. S. D. E. Porter and D. O. M. Bishop, "The case for less predictable operating system behavior," 15th Workshop on Hot Topics in Operating Systems, 2015.

[12] A. Alanwar, F. M. Anwar, J. P. Hespanha, and M. B. Srivastava, "Realizing uncertainty-aware timing stack in embedded operating system," in CEUR Workshop Proceedings, 2016, vol. 1697.

[13] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, 2016, pp. 363–378, Accessed: Dec. 10, 2020.

[14] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network." 14th USENIX Symposium on Operating Systems Design and Implementation, 2020.

[15] E. Cortez, M. Russinovich, A. Bonde, M. Fontoura, A. Muzio, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms?," in SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles, 2017, pp. 153–167.

[16] K. Ousterhout, S. Ratnasamy, C. Canel, and S. Shenker, "Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks," in SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles, 2017, pp. 184–200.
[17] A. Nógueira and M. Calha, "Predictability and efficiency in contemporary hard RTOS for multiprocessor systems," in Proceedings - 1st International Workshop on Cyber-Physical Systems, Networks, and Applications, CPSNA 2011, Workshop Held During RTCSA 2011, 2011, vol. 2, pp. 3–8.
[18] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," Queue, vol. 11, no. 7, pp. 40–51, 2013.
[19] "What is NUMA? — The Linux Kernel documentation." www.kernel.org.
[20] F. Arapidis, V. Karakostas, N. Papadopoulou, K. Nikas, G. Goumas, and N. Koziris, "Performance prediction of NUMA placement: A machine-learning approach," in Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, Dec. 2018, vol. 2018-December, pp. 296–301.
[21] "Page Frame Reclamation." www.kernel.org.
[22] "Better caching using reinforcement learning - UBC Wiki."wiki.ubc.ca/".
[23] J. Dean, "Machine Learning for Systems and Systems for Machine Learning," Learning, pp. 416–423, 2017.
[24] Z. Yu and B. Bhargava, "Self-learning disk scheduling," IEEE Trans. Knowl. Data Eng., vol. 21, no. 1, pp. 50–65, 2009.
[25] S. Grünewälder, "KMLIB: Towards Machine Learning For Operating Systems," arXiv, no. 113, pp. 13–21, 2020.
[26] J. Axboe, "Fio - Flexible I/O tester," freecode.com/projects/fio, 2014.
[27] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," Commun. ACM, vol. 15, no. 3, pp. 177–184, 1972.
[28] I. Ahmad, "Easy and efficient disk I/O workload characterization in VMware ESX server," in Proceedings of the 2007 IEEE International Symposium on Workload Characterization, IISWC, 2007, pp. 149–158.
[29] E. Varki, A. Merchant, J. Xu, and X. Qiu, "Issues and challenges in the performance analysis of real disk arrays," IEEE Trans. Parallel Distrib. Syst., vol. 15, no. 6, pp. 559–574, 2004.
[30] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, 2016, pp. 265–283.
[31] J. P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler: A decade of wasted cores," Proceedings of the 11th European Conference on Computer Systems, EuroSys, 2016.
[32] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Machine learning for load balancing in the Linux kernel," in APSys 2020 - Proceedings of the 2020 ACM SIGOPS Asia-Pacific Workshop on Systems, Aug. 2020, pp. 67–74.
[33] J. Kim, P. Shin, M. Kim, and S. Hong, "Memory-Aware Fair-Share Scheduling for Improved Performance Isolation in the Linux Kernel," IEEE Access, vol. 8, pp. 98874–98886, 2020.
[34] A. Negi, S. Member, K. K. P, and P. Kishore Kumar, "Applying Machine Learning Techniques to improve Linux Process Scheduling," IEEE Reg. 10 Annu. Int. Conf. Proceedings/TENCON, vol. 2007, pp. 1–6, 2005.
[35] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," arxiv, 2013.
[36] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015.
[37] S. Alabed, "RLCache: Automated cache management using reinforcement learning," arXiv. 2019.
[38] G. Vietri et al., "Driving cache replacement with ML-based LeCaR,", 10th USENIX Workshop, 2018.
[39] S. Charles, A. Ahmed, U. Y. Ogras, and P. Mishra, "Efficient cache reconfiguration using machine learning in NoC-based many-core CMPs," ACM Trans. Des. Autom. Electron. Syst., vol. 24, no. 6, 2019.
[40] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2019, pp. 413–425.
[41] P. Vinod, A. Zemmari, and M. Conti, "A machine learning based approach to detect malicious android apps using discriminant system calls," Futur. Gener. Comput. Syst., vol. 94, pp. 333–350, 2019.
[42] A. Vivona, A. Carminati, G. Cuozzo, G. Spagnuolo, and G. Alberto Falcione, "EXEIN CORE A host-based, run-time anomaly detection mechanism for Linux-based embedded systems." EXEIN, 2020.
[43] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," 2019.
[44] A. Adadi and M. Berrada, "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)," IEEE Access, vol. 6, pp. 52138–52160, 2018.
[45] N. Xie, G. Ras, M. van Gerven, and D. Doran, "Explainable deep learning: A field guide for the uninitiated," arXiv. 2020.