"Learned" Operating Systems

Yiying Zhang, Yutong Huang
Purdue University

Abstract

With operating systems being at the core of computer systems, decades of research and engineering efforts have been put into the development of OSes. To keep pace with the speed of modern hardware and application evolvement, we argue that a different approach should be taken in future OS development. Instead of relying solely on human wisdom, we should also leverage AI and machine learning techniques to automatically "learn" how to build and tune an OS. This paper explores the opportunities and challenges of the "learned" OS approach and makes recommendation for future researchers and practitioners on building such an OS.

1 Introduction

Among all types of software, operating systems are probably the most complex and intricate type. OSes are at the core of almost every computer system. They manage hardware resources and provide a protected environment for application execution. The design of an OS can thus affect all applications running on it. Traditionally, OSes are built by experts with long and recurring engineering efforts. Most OSes like Linux and Windows adopt general-purpose designs and leave various tuning options at or after installation time. The common practice is to install OSes with their default configurations and change specific configurations when needed.

There are four limitations with this long-standing practice of building and tuning OSes. First, OSes have evolved slowly and changing a mature OS is hard. However, today's hardware and applications change fast. Although new hardware and applications may not require building a whole new OS from scratch, they can largely benefit from rewriting or adding certain OS functionalities [6, 14]. Unfortunately, the traditional manual OS development approach cannot keep up with the pace of evolving hardware and applications.

Second, it is hard to properly tune an OS. Modern OSes have many configurations that can affect application performance. For example, Linux v5.1 (the latest version) has more than 17K kernel configurations in total. The process of hand-tuning the large number of OS configurations is lengthy and ad hoc. Moreover, doing so cannot achieve the best results.

Third, OSes do not "change" at runtime. After building, installing, and configuring an OS, its functionalities, policies, and parameters all stay the same. As a result, today's OSes cannot dynamically adapt to applications' changing behavior and needs.

Finally, general-purpose OSes cannot support various types of applications or hardware well. Most functionalities in today's mainstream OSes are designed for general purposes. Configurable parameters only allow limited specialization in certain OS functionalities. For example, the Linux swap system offers a parameter called *swappiness* to control the frequency of swapping out memory pages, but it always uses the LRU policy when choosing pages to swap out. Moreover, swappiness is a global parameter and all applications running on a Linux OS have to use the same swappiness value. With growing heterogeneity in future hardware and applications [2, 4, 5, 8, 10, 18], OSes should allow more specializations.

We believe that these limitations call for a rethink of traditional OS architectures. Our answer is to leverage machine learning (ML) techniques to build and configure OSes, an approach we call *"learned" OSes*. For example, we can use ML to predict the best configurations of an OS and such configurations can keep adapting to application changes at runtime. ML can also be used to generate *policies* and *mechanisms* of certain OS functionalities. By designing and building the framework to train and use ML models for OSes, we can avoid the huge engineering efforts of OS development. Moreover, ML-based approaches can potentially yield better results that are more adaptive and more fine-tuned to different applications.

However, "learning" OSes is not easy. There have been a handful of prior attempts in using ML techniques to generate or improve (simple) OS policies [16, 19], but none of them have been adopted at production scale. Nevertheless, with the recent success of applying ML techniques to low-level software like databases [11, 12], we believe that learned OSes are not only feasible, but also more efficient than current OSes which are built with heuristics and human experiences.

2 Opportunities for ML in OSes

ML can assist or replace at least three types of traditional OS components. First, ML can be used to (dynamically) set many *configurations* in an OS. Second, ML techniques can be applied to generate *policies* in an OS based on application behavior and hardware properties. The final and the most aggressive approach is to use ML to build certain OS *mechanisms*.

2.1 Learning Configurations

Modern OSes incorporate thousands of configurations that can be set by privileged users at or after OS installation time. For example, there are 89, 351, and 729 configurations for the memory system ("*mm*"), file systems ("*fs*"), and networking system ("*net*") in Linux-5.1. Among all types of Linux configurations, there are at least two broad categories that can directly affect applications' performances and can largely benefit from ML approaches.

First, there are many *timing*-related configurations in Linux, such as the frequency of interrupting a CPU core (for thread scheduling), the frequency of invoking background swapping (for memory paging), the frequency of flushing buffer cache (for storage), and the sampling rate of CPU clock frequency (for energy and performance). Setting these timing-related configurations is hard, since there are various tradeoffs associated with them. For example, frequent CPU interruption offers the opportunity to improve CPU utilization (with more aggressive thread scheduling) but can cause performance overhead (by preempting and context switching threads), which in turn, reduces effective CPU utilization.

Second, there are many configurations on various types of *size*s, such as the buffer cache size (for storage caching), disk prefetching amount (for storage access), and swap prefetching amount (for memory paging). Setting these size-related configurations is hard, especially when there are tradeoffs among different sizes. For example, a larger buffer cache improves the performance of the storage system but reduces available memory for user applications.

Many of the above two types of configurations can considerably affect application performance and other important metrics like energy cost. However, the practice of setting them has long been one that involves heavy engineering and human efforts: with heuristics, by trial and error, or with offline experiments. Moreover, once set, they are seldom changed.

ML is a better fit for setting these OS configurations. A good ML model trained with past workloads and OS/hardware environments can potentially outperform human-set configurations. The model can continue to *dynamically* generate new configurations to adapt to workload and environment changes. One promising ML technique to generate OS configurations is reinforcement learning. Although reinforcement learning has a more costly inference process, it fits our need well, since OS configurations only need to be re-generated infrequently.

2.2 Learning Policies

There are many decisions that an OS needs to make. These decisions often affect application performance and resource utilization, but rarely affect correctness. ML techniques that can be adapted to different application behaviors dynamically have the potential to outperform current OSes' global, static policies that are based on heuristics. Below, we discuss several types of OS policies that could be generated with ML.

Space allocation. A key task in OSes' management of hardware resources is space allocation. When an application requests for memory or storage spaces, an OS needs to decide which free space to give to the application. A lot of these allocation policies are based on heuristics and simple algorithms. For example, Linux allocates virtual memory spaces for *mmap* system calls using a best-fit policy (*i.e.*, choose the smallest virtual address hole that fits the requested *mmap* size). The Linux *ext* family of file systems allocates close-by spaces for files under the same directory. Although these policies work for many workloads and usages, they are not the best choice for all types of applications. For example, files under the same directory are likely to be accessed together, and placing them close to each other can save disk seeks. However, when users access files under different directories, the current ext file systems' allocation policy does not work well.

To better make space allocation decisions, OSes can use ML models to predict candidate locations for allocation. Building good ML models is crucial to the success of ML-based space allocation. A viable approach is to start building models (global, per-user, or per-application) by analyzing historical traces of how much space users requested, what space the OS allocated, how efficient spaces are utilized (*i.e.*, how much fragmentation there is), and how users access the allocated space. However, static models are not enough, since application behaviors can change and new types of applications can appear. We expect some online learning techniques will be needed to keep update space-allocation ML models.

Scheduling. An OS makes several types of scheduling decisions. For CPU scheduling, OSes decide which threads to run on each CPU core. When the time slice of a running process expires or when a process relinquishes its running core, Linux (since v2.6.23) uses the CFS (Completely Fair Scheduling) policy by default to decide which thread to run next. OSes also manage various queues such as network and storage queues. They schedule requests/operations on these queues for better performance, fairness, and load balancing. Setting good scheduling policy is hard and the criteria of a good policy can change from time to time. For example, before CFS, older versions in Linux use an *O(1)* CPU scheduler.

Rather than hand-tuned scheduling policies, dynamically generated scheduling decisions using ML can greatly benefit OSes. Additionally, compared to traditional scheduling policies, ML models have the potential to run faster and save metadata memory space. For example, the Linux CFS scheduler uses a red-black tree to store available virtual memory address ranges and requires O(logN) time to make a decision.

Cache management. Caching is a widely used technique in OSes. OS virtual memory systems store hot data in physical memory and leave the rest in a slower storage device. File systems use in-memory buffer cache to store hot file data and metadata. When managing a cache, OSes need to decide when and which data to evict. Currently, OSes use a set of fixed cache eviction policies, which usually are the result of years

of research and engineering efforts. For example, most OS virtual memory systems try to swap out memory pages that are least recently used with some form of approximate LRU. Such policies work well for workloads that display good temporal locality but not for those that have poor locality or for those whose locality cannot be captured by the approximate LRU policy.

Instead of a fixed policy like LRU, OSes can use ML to decide candidates of cache eviction. Such ML models can be learned using past memory/storage access patterns together with learning cache sizes (Section 2.1). We will discuss the challenges of learning correlated objectives in Section 3.1.

2.3 Learning Mechanisms

Most configurations and policies do not affect the correctness of an OS. ML is thus a good candidate for them. A more challenging task is to use ML for OS tasks that need to be precise. There are many such tasks in an OS and they are usually *mechanisms* to achieve some functionalities.

Inspired by learned index [12], we identified two "mechanisms" in an OS that can be replaced with ML models, both performing the functionality of mapping one abstraction to another abstraction. The first is the mapping from virtual memory addresses to physical memory addresses, which is currently performed by page table. The second is the mapping from a file name and offset to disk logical block address, currently done by file system multi-level index structure. These two types of "mapping tables" are crucial to the performance of all memory and storage systems, and extensive research and engineering efforts have been put into improving them.

Both the memory and the file mappings can benefit from an ML approach. ML models have the potential to reduce both the performance and space costs of memory and file mappings [12]. Moreover, ML models are flexible and can be customized to any types of workloads. Unlike fix-sized memory pages in today's memory systems, an ML-based mapping can inference any size and offset of memory space. Moreover, an ML model can potentially be smaller and run faster than a multi-level page table. We can further improve its performance by storing model parameters in a contiguous memory space to improve spatial locality and CPU cache hit rate.

3 Challenges and Potential Solutions

ML offers many benefits in building OSes. However, although the future of ML-based OS development is promising, many challenges remain to be solved before ML can actually be used to build real OSes.

3.1 Model Building

Section 2 presented our analysis of which part of an OS can benefit from ML, *i.e.*, what can be learned. The subsequent step is to design a learning approach. ML model selection is a hard problem in many ML applications. While systems like AutoML [17] can largely automate ML model selection, they only work for well-studied problems like image recognition and NLP. There are many unique challenges in selecting the right models for OSes.

First of all, how can we tell how good a model is? A seemingly straightforward way is to evaluate end applications' performance changes after applying a model. However, application performance can be affected by many factors like workload changes, other workloads running on the same OS, and other parts of the OS. To better pinpoint the effect of a certain model, we should seek better and more localized evaluation objectives. For example, instead of application performance, we can use buffer cache miss rate to determine the benefits of a model that predicts buffer cache replacement policy.

Next, shall we build global ML models, per-user models, or per-application models? Finer-grained models can achieve more accurate prediction with more customization and specialization, but require more resources (*e.g.*, memory space, CPU time for training).

Another potential optimization during prediction is to generate multiple candidates or multiple steps into the future. For example, the paging eviction model can return top K candidates for eviction, and the OS only needs to perform inference once in K page evictions. Doing so can reduce the performance overhead of generating a single candidate at each step.

A final major challenge is about learning multiple correlated tasks in an OS. Different configurations, policies, and mechanisms can all affect how well an OS sub-system performs, and sometimes they can even correlate with each other. For example, buffer cache size, flushing frequency, and eviction policy can all affect the performance of buffer cache (and in turn the performance of the storage system). The size of the buffer cache can also affect the performance of the memory system. We thus should jointly optimize related tasks for the best results. We envision multi-task learning to be helpful in such situations.

3.2 Training

Training ML models for OSes present unique challenges. First, collecting training data should impose minimal overheads to foreground applications. For example, it is not feasible to trace each memory access to build models for predicting memory eviction candidates. A large amount of training data also causes prohibitively high space overhead. On the other hand, not having enough training data can reduce the accuracy of ML models. One potential solution is to train models offline using pre-collected, well-represented training data and then use online training to (infrequently) update the built models. The offline training can use more fine-grained data, while for online training, we may only be able to collect coarse-grained data so as not to disturb foreground application performance.

Second, how do we build validation sets? For certain problems, there is ground truth or a theoretical optimum. For example, for CPU scheduling, theoretically it is best to first run the job with the shortest remaining time (for fastest turn-around time); for page replacement, the theoretical best is to evict the page that will not be used farthest into the future. These theoretically best solution can directly be used as the validation sets during training. However, for other problems, there is no clear best solution. One possible approach is to let users to define their applications' requirements or objectives by providing *reward functions* instead of a validation set. OSes can then use reinforcement learning techniques to find the best solution for these requirements/objectives.

3.3 Inference

Different OS learning objectives have different criteria for the speed of inference. The first category of learning only needs to run once in a while or when workload changes, *i.e.*, foreground application operations do not wait for any of these inference results. Configurations and policies all fall into this category. For this type, an inference can run a bit slower, allowing us to explore more costly ML techniques like reinforcement learning.

The second category of OS "decisions" must be made very fast, since they are on the application performance critical path. For example, OSes need to decide which thread to schedule on a core before the thread can start execution, implying a tight bound of decision making to be within or at most around the time of a context switch. With fast storage and networking devices [9, 20], decisions in OS file/storage and network systems also need to be made fast (around or within 1μs). GPUs and other specialized processors like TPU can make fast prediction with complex models, but invoking them still takes long (1-2μs with today's PCIe). To be able to make inference fast enough for these OS usages, we either need to reduce this invocation cost or reduce model complexity and run it on a local CPU.

Along with the performance overhead of performing inference, there is also the memory space overhead to store ML models for inference. Big models can easily take hundreds MBs of memory. When there are hundreds to thousands of configurations, policies, and mechanisms to learn in an OS, the model space costs can be prohibitively high. A promising approach to reduce memory consumption is leveraging model memory-reuse techniques [1] in recurrent neural networks(RNNs).

3.4 Integrating ML in OSes

ML alone cannot make a whole OS. We foresee several challenges in integrating ML models and their predictions into existing OSes.

First, while most OS policies are used just for performance improvement and can be sub-optimal or "wrong", some OS functionalities such as file system and virtual memory mappings need to be precise. How to use machine learning techniques that are probabilistic in nature to achieve OS deterministic tasks is an interesting yet challenging problem. One viable approach is to let ML model first make a range of probabilistic predictions and then use traditional algorithms to compute the final exact answer within this range. The more precise the ML predicted range is, the faster the second step of the exact search can be.

Besides correctness guarantee, how to run ML models in kernel space is also a new challenge. Unlike the user space, the kernel space lacks the support of ML libraries. In order for a kernel to use ML techniques, potential options lie between building new kernel-space ML libraries and running ML in user space and then passing results back to the kernel space. The first option requires significant engineering effort while the latter option may suffer from performance loss in context switches.

Finally, modern OSes like Linux take a monolithic kernel approach and have become very complex over decades of development efforts. To integrate ML in existing OSes will require careful engineering to minimize the disturbance of the rest of the OSes.

3.5 Security

OSes should offer applications protected accesses to hardware resources. With the learned OS approach, is it safe to use ML models that are learned using application data in OSes? Although we envision all security- and protection-related tasks in an OS to still be implemented in the traditional manner, using ML for the rest of the OS kernel can introduce unique security threats and implications.

When applying ML technique in OSes, user data will be involved indirectly in the control plane of OSes. Without proper protection, malicious users will be able to manipulate both the training and the inference process by feeding carefully crafted data. Doing so can cause OSes to use wrong ML models that work in the attackers' favor. For example, an attacker can train an ML model to always evict other applications' memory and launch a denial-of-service attack. It can also generate a bad ML model that causes an OS to constantly miss predict and suffer from deteriorated performance. Using separate ML models for different applications (*i.e.*, isolation of ML) can largely improve the learned OS' security. However, it is still challenging to protect from side-channel attacks [3] and to prevent information leakage.

4 Related Work

ML for other low-level systems. The recent work of learned index [12] proposes to replace traditional tree-based index structures by neural network models that predict the location of data. Learned index has inspired many research works afterwards. In fact, this paper was also inspired by it. A subsequent

work of learned index is SageDB [11], a learned database system. It extends learned index with learned Apart from index, it uses ML for merge and hash-join operations. Another extension of learned index is learned bloom filters [15]. There are also many proposals to use ML for various hardware problems. More relevant is a recent work that predicts memory access patterns and performs memory prefetching using recurrent neural network (RNN) [7].

ML for OS. Although ML has been used in many domains and recently more in low-level systems, OSes have rarely adopted any ML techniques and most research proposals dated decades back. For example, there are several proposals of using ML techniques (e.g. C4.5 decision tree, linear regression) to improve application job average turn-around time, for example, by tuning kernel preemption time [16] and by predicting job run time [19]. Lynx [13] is a system that use ML to better perform prefetching from SSDs. It leverages Markov Chains to detect I/O workload patterns and compute the transition probabilities between file pages.

5 Conclusion

Advances in ML techniques and the availability of "big data" and computing resources have made it possible to apply ML in many domains that previously rely on human efforts. We believe that OS is also such a domain. This paper systematically explores the opportunities and challenges in using ML for OSes. It is of course just a starting point of building real learned OS solutions. We expect more design, development, and deployment challenges to appear. Nevertheless, we believe learned OS to be a direction that is worth exploring and hope this paper to inspire and help future researchers and practitioners in this area.

Acknowledgments

We would like to thank Jian Huang from Pinterest for his input in the early discussion of this work, which has substantially improved the content of this paper. We would also like to thank the SIGOPS Operating Systems Review editors, Kishore Kumar, Chris Rossbach, and Robbert Van Renesse for their feedback and comments of this paper.

References

[1] Audrūnas Gruslys and Remi Munos and Ivo Danihelka and Marc Lanctot and Alex Graves. Memory-efficient backpropagation through time. In *Proceedings of Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[3] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018.

[4] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[5] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[6] L. Guo, S. Zhai, Y. Qiao, and F. X. Lin. Transkernel: An executor for commodity kernels on peripheral cores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[7] M. Hashemi, K. J. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. *international conference on machine learning*, pages 1919–1928, 2018.

[8] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. K. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[9] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *USENIX ATC '16 Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, 2016.

[10] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[11] T. Kraska, M. Alizadeh, A. Beutel, E. H. hsin Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[12] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[13] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff. Lynx: a learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016.

[14] C. Lee, D. Sim, J. Y. Hwang, and S. Cho. F2fs: a new file system for flash storage. In *FAST'15 Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[15] M. Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. *neural information processing systems*, pages 464–473, 2018.

[16] A. Negi and K. Kishore. Applying machine learning techniques to improve linux process scheduling. In *TENCON 2005 - 2005 IEEE Region 10 Conference*, 2005.

[17] Quoc Le and Barret Zoph. Using machine learning to explore neural network architecture. https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html, 2017.

[18] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[19] W. Smith, I. T. Foster, and V. E. Taylor. Predicting application run times using historical information. *job scheduling strategies for parallel processing*, pages 122–142, 1998.

[20] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, 2015.