On a Foundation Model for Operating Systems

Divyanshu Saxena, Nihal Sharma, Donghyun Kim, Rohit Dwivedula, Jiayi Chen, Chenxi Yang, Sriram Ravula, Zichao Hu, Aditya Akella, Sebastian Angel, Joydeep Biswas, Swarat Chaudhuri, Isil Dillig, Alex Dimakis, Daehyeok Kim, Chris Rossbach, Gang Wang

Abstract

This paper lays down the research agenda for a domain-specific foundation model for operating systems (OSes). Our case for a foundation model revolves around the observations that several OS components such as CPU, memory, and network subsystems are interrelated and that OS traces offer the ideal dataset for a foundation model to grasp the intricacies of diverse OS components and their behavior in varying environments and workloads. We discuss a wide range of possibilities that then arise, from employing foundation models as policy agents to utilizing them as generators and predictors to assist traditional OS control algorithms. Our hope is that this paper spurs further research into OS foundation models and creating the next generation of operating systems for the evolving computing landscape.

1 Introduction

The Operating System (OS) is the central pillar of modern computing systems, overseeing hardware and software resources and enabling applications ranging from assistive robotics to cloud services. OSes serve vital tasks such as scheduling processes; managing CPU, network, and memory resources, and interfacing with devices. To make good decisions, OS policies must account for complex system dynamics such as hardware variances and environment responses, which is challenging for two reasons. First, OSes can be deployed atop a variety of hardware, and amidst diverse workloads and environments. Second, the OS does not have full visibility of the environment (e.g., network performance) or the workload (e.g., application request patterns), making the state space uncertain.

Conventional OS policies, reliant on manual algorithms or heuristics, lack adaptability across hardware, environments, and workloads, and often require manual tuning. Recent proposals for using machine learning (ML) models in OS components such as the network manager [1, 23], memory manager [26, 28, 40, 53] and CPU scheduler [9], while being good starting points in bringing data-driven decisions, are still far from ideal as they only optimize for individual components. Furthermore, they neither integrate well together nor generalize well for diverse environments.

Inspired by the recent successes of large unsupervised “foundation” models in NLP and vision tasks, we argue that it is time for the OS to eschew such task-specific solutions in favor of foundation models. Our insight is that OS traces consisting of hardware metrics, system event logs, and application arrivals and requests, can capture all the information on the workings of various OS components and the impact of their decisions on each other. Further, OS traces collected on diverse hardware and application workloads can also capture the intricate relationship between OS decisions, hardware

Figure 1: FM4OS: a foundation model for operating systems.
features, and application workloads. We argue that a foundation model trained on such traces, FM4OS, is plausible and can be used for several downstream tasks (as shown in Figure 1).

2 Background

In this section, we first provide background on OS decision-making: what makes it difficult and why adaptive decisions are needed, and then we give a brief background on foundation models.

Desiderata for operating systems. Operating systems oversee hardware and software resources, including CPU, memory, storage, and network (Table 1 in appendix). In general, OS tasks can be considered sequential decision-making processes where past actions and states of the OS instruct the action at any time. However, these can be very complex because:

- OSes can be deployed on diverse hardware with differing performance profiles. Further, they can run different workloads (e.g., microservice [29] vs. ML workloads [50]) with varying objectives (e.g., prioritize power efficiency for robots vs. optimize performance for cloud servers).
- Access to fine-grained metrics (like the ones shown in Table 1: System State column) from hardware devices or the OS kernel, may be limited.
- System dynamics, i.e., the interplay of policies between OS components, also plays a role in decision-making because the actions of one component can impact the future states of other components. Capturing these intricate dynamics is difficult due to the myriad OS policy combinations. Thus, in the OS setting, there is an inherent uncertainty and partial observability in the state.

Existing methods: Prior research has proposed learned and data-driven approaches to address these challenges. Some have employed DNNs to learn policies for specific OS components [2, 9, 22, 30, 56, 57] while others have tackled state uncertainty by modeling OS tasks as MDPs [1, 31, 34, 48, 53]. Additionally, statistical and deep learning methods have been explored to generate realistic workloads [3, 20, 21, 27, 55] that can help inform conventional policies. However, these approaches remain point solutions that model individual OS components, leading to a diverse bag of policies, operating independently of others. Consequently, they fall short in optimizing end-to-end OS performance and decision-making.

Ideally, if we could learn how an OS task is impacted by other OS components, application workloads, and hardware specifications, we can devise methods to optimize OS decisions for desired objectives. These existing approaches also struggle with generalization beyond their training distribution, as shown in prior research [17, 41]. Therefore, we need techniques that generalize well to unseen inputs.

Foundation models. This is a catch-all term for ML models trained on a large and diverse dataset to understand the general structure of the data and then fine-tuned (with much less data) for specific tasks. While they have been touted as useful in myriad settings [4, 54], these models have shown tremendous empirical success in sequence modeling problems in natural language [5, 10, 15], finance [51], computer vision [38], biomedical imaging [44] and climate modeling [32] to name a few. At the core of these successes is efficient use of the transformer architecture [45], which learns long-term (spatial and/or) temporal correlations between input sequences, and the principles of transfer learning [42], that enable learning for different tasks, domains, and modalities.

Several OS tasks also fall into this broad category of sequential modeling with the important caveat that the cadence with which decisions are made and the amount (time) and explicit form (states) of past observations vary widely between tasks (see Table 1 in Appendix B).

3 FM4OS: A Foundation Model for the OS

We propose the development of FM4OS, a foundation model that understands the “natural behavior” of the OS and can be fine-tuned for several classes of downstream tasks - all of which either replace or aid the existing policies in the OS. We begin by describing the data sources available that can be used to train such a model.

Data Sources. Today’s OSes, along with associated monitoring and data collection infrastructures, provide data in several forms (as shown in Figure 1), including logs from OS components, hardware metrics, and application workloads. We elaborate on these sources in Appendix A.

We will use the term “OS trace” to refer to the union of the data corresponding to a single machine drawn from the sources above, represented as a single (time-annotated) sequence. Such traces can be collected from systems with varying hardware specs (CPU, Cache, RAM, NIC, file system, etc.) and under various deployments (cloud, robots, and edge). Below, we describe two OS tasks that operate on different parts of the system, both of which can be trained from the OS traces.
An example use case. Consider the OS scheduling task \textit{SCH} and cache replacement task \textit{CACHE} (descriptions as given in Table 1). As shown in the table, optimal decision-making for the \textit{SCH} task requires process states, process completion times, hardware state, and process arrival workloads while the \textit{CACHE} task requires cache size, state, and cache access workloads. All of these are captured in the OS traces. For \textit{SCH}, the process states and completion times are captured in the logs, process arrival workloads and hardware aspects are captured by the application workloads and environment metadata, respectively. Similarly, for \textit{CACHE}, the cache state is captured by the resource metrics, cache size by environment metadata, and cache access patterns by application workloads.

The OS traces also capture the relationships between the two tasks. For example, the process completion times would depend on the hardware specs of resources other than CPUs, such as the cache. This is because processes may access resources other than CPUs during their execution. For the same reason, OS decisions relating to \textit{CACHE} would also impact the process completion times. Since our OS traces record features that cover the input space of both tasks \textit{SCH} and \textit{CACHE}, they can be used to train one model that can orchestrate both. This model can then be used for several downstream tasks, including: (i) directly making good-quality decisions for the \textit{SCH} and \textit{CACHE} tasks, (ii) predicting the completion time of a newly arrived process, or (iii) generating traces for \textit{CACHE} tasks that can be used to improve conventional data-driven or ML-based algorithms.

When trained on diverse OS traces, the model learns how scheduler and cache behaviors relate to hardware and workloads, enabling generalization to predict program performance on new CPU specifications and cache sizes.

\textbf{Foundation Model for the OS}. The OS traces used above not only encode information for \textit{SCH} and \textit{CACHE} tasks but also that corresponding to the decision-making in several other OS components, e.g., I/O prefetching, packet scheduling, congestion control policies, etc. Referring to Table 1, we make two observations to support this. Firstly, several of the OS tasks have shared state space components. For example, both \textit{PREFETCH} and \textit{CACHE} tasks need the cache state, both \textit{PREFETCH} and \textit{PAGE} tasks require process instructions, etc. Secondly, these tasks are not entirely independent, as shown in the above \textit{SCH} task example, where the process completion times (needed for \textit{SCH}) depend on the policies adopted in the \textit{CACHE} task. This inter-dependence of one component on others is a widely seen and natural phenomenon in the operating system. Using OS traces collected across many machines, one can therefore build a foundation model – FM4OS, that knows the ‘natural behavior’ of the OS. A prospective pretraining regime for FM4OS is discussed in Appendix C.

\section{4 Downstream Tasks for FM4OS}

We are now ready to discuss the fine-tuning of FM4OS. We present key downstream tasks and categorize them into three broad use cases: as a policy agent, a generative model, and a predictive model. We discuss these individually below and highlight challenges unique to the OS setting that require novel research on training and using foundation models, in Appendix D.

\subsection{4.1 FM4OS as a Policy Agent}

As discussed in §2, several OS tasks can be modeled as a sequential decision-making process where the state of the OS evolves according to the actions a policy makes. Prior works \cite{1, 9, 22, 34} have used handcrafted features based on heuristics in order to model complex system dynamics.

The key challenge for any solution addressing multiple tasks in the OS is the diversity in state and action spaces of tasks and the different lengths of temporal history deemed relevant for each task (see Table 1). Foundation models have been shown to solve precisely this issue of varying lengths of temporal history due to their ability to summarize inputs of arbitrary lengths in a common representation space. Further, they have also shown evidence of being capable of handling multi-modal input data \cite{43}, which suits the various forms of information captured in OS traces (see §3). By engineering the size of these representations for pre-training and specifying the objective during fine-tuning, we expect that FM4OS can be used to suggest optimal actions.

\textbf{Making low-level decisions:} By pre-training FM4OS over OS traces, we expect it to understand the semantic space for OS decision-making. Then, we can use FM4OS to take low-level actions for OS tasks, such as setting the congestion window for the \textit{CC} task and choosing processes for the \textit{SCH} task. Fine-tuning to make these decisions requires historical traces labeled with optimal actions.

\textbf{Policy selection:} Current inference times for transformer-based models do not match the pace at which some OS tasks require actions (every few n.s). Accelerating inference \cite{13, 52}, especially
for operation in the OS [19] is an ongoing research area. In the meantime, FM4OS can address the relatively simpler task of selecting from existing policies (over longer time frames) instead of specifying actions explicitly. For each task, there exist policies optimized for specific environments and workloads. For instance, for the \texttt{CACHE} task, Least Recently Used (LRU) policy is favored when access patterns follow locality trends, while Least Frequently Used (LFU) policy is more suitable for random accesses with consistent popular requests [37, 49].

\section*{4.2 FM4OS as a Generative Model}

Content generated by ML models offers new opportunities for OSes, similar to benefits observed from using generative models in other domains [3, 38]. Synthetically generated data can help add diversity to existing training data used by data-driven solutions, help with the availability and sharing of proprietary or confidential data, and testing models under settings that occur infrequently in practice.

**Generating traces:** The lack of (diverse) training data is a major hurdle in most data-driven and learned approaches for OS tasks. Even if such data were available, storing and maintaining such a large corpus of data collected under different hardware configurations and workloads be challenging. For example, for the \texttt{CACHE} task, traces are needed for different memory specifications and for different types of workloads (small objects, large objects, mixed sizes, etc.). By training FM4OS using auto-regressive tasks like Next Token Prediction, we could train the model to learn to generate OS traces that can be used in a variety of ways.

Fine-tuning it with specially designed prompts could lead to traces that adhere to specific constraints (e.g., setting hardware configurations, limiting network bandwidth, etc.). These can be used to supplement the training data collected on specific configurations. Further, the foundation model can also be fine-tuned to obfuscate confidential information from the traces while keeping the important relationships of the traces intact (prior works [27, 55] show feasibility of such obfuscation in network traces). Another opportunity that we identify here is that FM4OS can also be used to generate pathological corner cases. Specifically, we posit that by appropriately querying the foundation model, we can use it to generate pathological corner cases that would have been otherwise difficult to get.

\section*{4.3 FM4OS as a Predictive Model}

Foundation models have been shown to exhibit good performance on downstream prediction tasks [5, 32]. In the OS setting, we can use FM4OS as an encoder of the state, and then use linear probing to predict various things about the system’s response, future utilization. This can lead to efficient placement, scheduling, performance, and anomaly detection.

**System response prediction:** Understanding how the environment of the OS evolves with application-level decisions made by the OS are crucial to improve decision quality. For example, predicting the time to completion of a process would allow the kernel to reorder its CPU work queue based on completion times leading to an optimal schedule for minimum waiting time of jobs. Since FM4OS is pre-trained to understand precisely the needed semantic relationships between OS subsystems, it can be used to closely predict system responses.

**Application behavior prediction:** Predicting the behavior of an application can help the OS prepare in advance for additional resources the application might need and minimize competition for shared hardware. For example, if the OS can predict that an application’s execution will be memory-intensive in the near future based on its recent memory allocation calls and nature of inputs received, it can both provision more memory for the application and avoid scheduling another memory-intensive application on the same node.

**Anomaly Detection:** Using the state encoding of the OS or any of its components, and given a trace, one can ask if the current state is normal, or if there is some anomaly or failure issue. Such predictions can be used to identify and kill anomalous applications, thereby improving the security of the OS kernel.

\section*{5 Summary}

In conclusion, we argue that the OS decision-making tasks provide a rich arena for a domain-specific foundation model to be built for the OS. We discuss the shortcomings of existing methods of data-driven decision-making and posit that rich OS traces can provide the necessary data to train such a foundation model, FM4OS, which can understand the ‘natural behavior’ of the OS. We then provide a systematic analysis of the various ways in which FM4OS can be used and the various key challenges that remain open research questions.
References


[39] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at


A Data Sources for OS Traces

Below we list the various data sources that can be used to train FM4OS:

- **Action logs from OS components:** Kernel logs such as `dmesg` in Linux/MacOS and event logs in Windows, capture kernel debugging data, hardware events (e.g., network link status), and system events like interrupts, process restarts. These logs capture the actions taken by the OS components and the system state used by OS tasks (System state and Actions columns in Table 1).

- **Resource metrics and hardware counters:** Hardware drivers record several quantities relating to the resource’s state at a pre-configured frequency. These include CPU, memory, and disk bandwidth utilization, NIC queue length, and hardware counters.

- **Application workloads:** Workload traces from productions [29, 39, 47], public infrastructures [18] and synthetically generated ones offer detailed application-level information, such as application type, request arrival rates, statistics of resource usage during execution.

B Decision Making Tasks in Operating Systems

Table 1 shows the various components in the OS and a representative subset of the tasks for these components. It also shows the relevant system and environment states, action spaces, and the objectives of these tasks. Each task description is also accompanied by an acronym that we use in the paper to refer to the particular task, e.g. SCH for the CPU scheduling task.

C Pretraining Methodology for FM4OS

We envision FM4OS to be **pre-trained** using self-supervised methods on a large corpus of OS traces. This pre-trained model would capture temporal relationships in the sequence of inputs it accepts and build an understanding of the system dynamics of the OS. Existing literature (particularly in natural language) has proposed several pre-training tasks that can be used to develop this basic understanding. Notable among these are the Next Token Prediction [35], Masked Language Modeling, Next Sentence Prediction [14]; each with their own pros and cons. While it seems intuitive to employ an auto-regressive model, pre-trained with next token prediction to build FM4OS, the optimal pre-training task is an open and interesting question in itself.
**Table 1: Various decision-making components in the OS.**

<table>
<thead>
<tr>
<th>Components</th>
<th>Description and Acronym</th>
<th>System State</th>
<th>Environment Info</th>
<th>Actions</th>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Scheduling</td>
<td>Process state (niceness, priority, execution time); Hardware state (CPU, RAM spec, etc.)</td>
<td>Arrival pattern and type of processes (computational, heavy vs. I/O-heavy)</td>
<td>Process to core assignment</td>
<td>Job completion time, fairness</td>
</tr>
<tr>
<td></td>
<td>Voltage and frequency scaling</td>
<td>CPU frequency buckets, Hardware spec of the CPU</td>
<td>Process workloads, and process instructions</td>
<td>Choose CPU frequency</td>
<td>CPU performance, power, temperature</td>
</tr>
<tr>
<td>Memory Subsystem</td>
<td>Page Allocation</td>
<td>Page table size, Hardware spec (amount of memory, type, etc.)</td>
<td>Memory access patterns of running processes</td>
<td>Page Size, Allocation mechanism</td>
<td>Latency of memory accesses</td>
</tr>
<tr>
<td></td>
<td>Cache</td>
<td>Cache hit ratio</td>
<td>Throughput and delay</td>
<td>Throughput and delay</td>
<td>Throughput and delay</td>
</tr>
<tr>
<td></td>
<td>I/O scheduling</td>
<td>I/O metadata (block offset, size, queue state, historical I/O latencies)</td>
<td>Application type (video streaming, analytics, etc.)</td>
<td>Order of I/O requests</td>
<td>I/O latencies</td>
</tr>
<tr>
<td>Storage Subsystem</td>
<td>Prefetching</td>
<td>Cache size and state, Cache and PCIe spec</td>
<td>Process workloads and process instructions</td>
<td>Choose segment to prefetch</td>
<td>Throughput of future reads</td>
</tr>
<tr>
<td></td>
<td>Cache replacement</td>
<td>Cache size and state (occupied, address, last access)</td>
<td>Cache workloads (object sizes, frequency of access, etc.)</td>
<td>Choose a set of objects to evict/submit</td>
<td>Cache hit ratio</td>
</tr>
</tbody>
</table>

### D Open Challenges for FM4OS

#### D.1 Challenges in using FM4OS as a Policy Agent

End-to-end application performance depends on collective decisions made by OS components. Using foundation models as policy agents brings two unique challenges: *composability* of actions from various policy agents and end-to-end *explainability* of their decisions. The former arises because decisions of one policy can affect the future states of other agents (as with the CACHE and SCH example discussed in §2). Independently fine-tuned components in the OS may result in suboptimal OS-wide decisions, that may affect both individual application and system-wide guarantees (e.g. fairness and starvation-freedom). One possible approach here is to jointly fine-tune components (to ensure concerted decisions) as well as to develop techniques that provide component-wise guarantees (on performance, e.g., bounds on tail request completion times, or correctness, e.g., safety and liveness properties [17]), and *formally guaranteed composability* of these actions to provide global invariants for the entire OS.

Regarding the latter, ideally, we desire human users to understand the OS at some level to audit or debug it. However, learned decisions from a black-box model may easily obscure the understanding of overall behavior. We envision the use of approaches that describe what each learned policy did in a given execution (similar to LIME [36]), what could have happened had a learned policy made a different decision, and also produce human-comprehensible ‘summaries’ in the form of rules [11, 12], or programs [46] of what the module will do ahead of time.

#### D.2 Challenges in using FM4OS as a Generative Model

As with any generative model, quantifying the quality of synthetic samples is a key challenge. At the very least, these traces should maintain certain relationships between variables (e.g., total network transmissions should be less than network bandwidth). They must also capture desired properties that are difficult to obtain otherwise, such as ‘realism’, i.e., a specific sequence of requests in a generated trace can actually arise in practice — this is an avenue for future research. Further, the generated traces should also be diverse to be useful. For example, for a cache replacement algorithm, we would want traces with diverse and realistic combinations of small and large object arrivals to effectively stress-test the algorithm [8]. Another challenge is with leakage and memorization of sensitive data. As shown in previous works [7, 33], carefully designed prompts can extract memorized training data with sensitive information. Thus, integrating techniques such as filtering the memorized data [6] and ideas from say, differential privacy [16], into FM4OS are necessary.