Operating System Policies in the Wild

Yushan Qin
yushan.qin@wisc.edu
University of Wisconsin–Madison
USA

Ananya Virmani
av664@cornell.edu
University of Wisconsin–Madison
USA

Sungkar Bolat
bolat@wisc.edu
University of Wisconsin–Madison
USA

Ayushi Chaudhary
adchaudhary@wisc.edu
University of Wisconsin–Madison
USA

Nida Tanveer Ul Hasan
ntanveer@wisc.edu
University of Wisconsin–Madison
USA

Mai Zheng
mai@iastate.edu
Iowa State University
USA

Sebastian Angel
sebastian.angel@cis.upenn.edu
University of Pennsylvania
USA

Christopher J. Rossbach
rossbach@cs.utexas.edu
The University of Texas at Austin & Microsoft
USA

Michael M. Swift
swift@cs.wisc.edu
University of Wisconsin–Madison
USA

Abstract

Operating system policies determine what should happen and when. They have a large impact on performance, and are the subject of much research. However, they are largely discussed in isolation: the single question of what thread to run next on a CPU, or what memory page to replace next. Decades of development in Linux, though, teaches us that OS policies can be both more complicated and simpler; there are also many more than commonly discussed.

This paper presents a study of Linux kernel policies across multiple subsystems with the goal of better understanding what they do, how they are implemented, and how readily they can be changed. Using manual code inspection, we identify over 330 policies across the Linux memory management, scheduling, I/O, file, and power subsystems. We identify common implementation styles, such as single and multi-function decisions, inline-code and configurable parameters; common features used for decisions such as local and global state, historical context, and hardware information; common decision types such as selecting objects, identifying attributes, and changing system modes. Taken as a whole, this study presents a comprehensive picture of the diversity and common structures of Linux policies, and serves as a basis for future work in developing new ML-driven policies.

CCS Concepts: • **Software and its engineering → Operating systems**.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857040

Keywords: policy, operating system, scheduling, memory management, file system

ACM Reference Format:
Yushan Qin, Ananya Virmani, Sungkar Bolat, Ayushi Chaudhary, Nida Tanveer Ul Hasan, Mai Zheng, Sebastian Angel, Christopher J. Rossbach, and Michael M. Swift. 2026. Operating System Policies in the Wild. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857040

1 Introduction

Operating systems are responsible for managing resources and providing isolation between processes. To achieve these goals, OS designers have long relied on a set of well-known, high-level policies. Many of these are policies taught in every undergraduate OS course: processor scheduling, I/O scheduling, page allocation and reclamation, NUMA balancing, file-system caching and allocation. These well-known policies are important drivers of system performance and have received decades of research attention. In many cases, improved policies can lead to significant performance gains through better resource utilization, with no change in the underlying mechanisms. [36, 38, 44, 48].

However, OSes also implement many policies that are rarely discussed in the general OS literature. These include decisions such as specific cache sizes, process limits and quotas, how to handle executable pages during page scanning, CPU frequency scaling, or when a file system should allocate, flush, merge, or reclaim internal state. These smaller policies may appear to have minor impact when considered in isolation, but they collectively define the behavior of the system and the performance observed by applications. They are also numerous: in just five Linux subsystems, we identify more than 330 policies, and over 200 of them cannot be changed without modifying or rebuilding the kernel. While

OSes include non-performance policies (e.g., security), this paper focuses on policies that impact performance.

We undertake a detailed study of OS policies with a specific goal: understanding what a production OS would have to expose, abstract, or restructure to support the pervasive use of data-driven or *instance-optimal* policies, where an *instance* is the combination of an OS, a workload mix, and an execution platform. It is a well known observation that no default policy is best for every instance [39], and prior work has demonstrated the benefits of workload-specific policies [27, 30], architectures that make policy customizable [2, 5, 17, 39, 54], extension mechanisms such as eBPF [3, 13, 62], and learned policies [1, 7, 14, 18, 19, 23, 63]. These efforts required substantial human effort, and each addresses a small slice of an OS's policy surface, leaving most of it, and its potential benefits, unexplored.

The cost of finding better policies is now falling though. Rapid progress in LLM-based code generation [8, 29, 40] has produced systems that pair LLMs with automated evaluation to discover better heuristics and algorithms [46, 51, 56], including systems policies [9, 16, 21, 65]. The ability to *synthesize* policies from existing code and runtime telemetry could dramatically reduce the effort and time needed to find instance-optimal policies. But these advances move the bottleneck from finding a better policy to deploying it: a synthesized policy is useful only if the OS exposes a way to swap the existing policy for the new policy.

Linux, meanwhile, allows only some of its policy to be updated easily. For example, it supports several mechanisms for customizing policy, including scheduler classes, tunable parameters, `Kconfig` options, eBPF extensions, and upcalls to userspace, but these are not a general policy customization substrate. Many policies are encoded as constants, distributed across helper functions, embedded in mechanism-heavy code, or exposed only through coarse tunables whose semantics are buried in implementation details. Learned systems that target these policies often must rewrite kernel code, restrict themselves to a small set of exported knobs, or approximate the desired behavior through indirect control [64]. Making policies visible, separable, and replaceable therefore starts with understanding them: what form each policy takes, what information it uses, what it decides, and what it trades off.

This paper provides that understanding for Linux. We report on a comprehensive effort to study the Linux 6.6.42 kernel to categorize policies in key subsystems: CPU scheduling, memory management, block I/O scheduling, file systems, and power management. For each policy we identify, we record how it is implemented (as a decision function, a multi-function decision, inline logic, a configuration parameter, a compile-time constant, or a system call that sets policy), which features it uses (global state, local object state, history and feedback, configuration and constants, or hardware information), what kind of decision it outputs, and which objectives it trades off. We do not propose or evaluate a technique for generating policies. Instead, we ask what OS support such techniques would need; our case studies use hand-written and LLM-synthesized policies to illustrate how the study informs interface design.

Summary of results. CPU scheduling and power management separate policy from mechanism most cleanly. Each scheduling class implements its decisions behind a few functions such as `pick_next_task()`, and most of the remaining scheduling policies are runtime parameters or compile-time constants. Power management is organized around replaceable `cpufreq` governors that choose CPU frequencies to balance performance against energy. I/O schedulers can also be replaced as a whole through the block layer's scheduler interface, but within each scheduler, policy is interleaved with mechanism. CPU and I/O scheduling also expose runtime parameters that adjust tradeoffs among latency, throughput, and fairness.

Memory management is different. It accounts for nearly 200 of the policies we identified, nearly half of which are constants or configuration parameters, and it has more policies interleaved with mechanism than any other subsystem (20 multi-function or inline). It also has the most decision functions (83), but unlike `pick_next_task()`, most of them make one step of a larger decision rather than the whole decision. Page reclamation, for example, emerges from a chain of smaller decisions (which LRU implementation to use, whether the inactive list is too small, how much of each list to scan, and which folios to evict) that are spread across helper functions and interleaved with mechanism.

File systems add a different kind of complexity. Their only runtime policies implemented as code are the block and inode allocators; 51 of their 63 policies are configuration settings, set at one of three points in the file system's lifetime. Format-time settings such as block size and inode density cannot be changed without reformatting the device, mount-time settings require a remount, and only a few `sysfs` knobs can be changed on a live file system.

How to use these results. For current operators, the study maps the policy surface of Linux: which behaviors can be tuned at runtime, which can be changed only by modifying the kernel, and which emerge from interactions among many small decisions. It also identifies policies that operate in the same tradeoff space (e.g., latency versus throughput) and may therefore reinforce or undermine one another when tuned independently.

More importantly, the study is a design guide for future OSes that support instance-optimal policies. It shows where Linux already separates policy from mechanism well enough that a new policy, whether hand-written, synthesized, or learned, could be introduced with little restructuring, and where current implementations would leave such a policy

brittle or incomplete. A policy expressed as a single decision function with explicit inputs and outputs, such as the scheduler's `pick_next_task()`, can be replaced directly. A policy spread across helper functions, constants, and side effects in mechanism code, such as page reclamation, cannot; the OS must first be restructured to expose the decision, its features, its objective, and its constraints. Finally, a policy interface cannot cover only the obvious top-level decisions. It must also account for the constants, thresholds, feature computations, fallback paths, and subordinate decisions that collectively determine behavior.

We make the following contributions.

- We report on the implementation styles of major Linux policies across multiple subsystems. We document the feature types used by policies to demonstrate how Linux policies mix local information about a specific operation with global information about the state of the system, including computed metrics that encapsulate historical state.
- We explain the role of configurable settings and constants in policy for expressing policy objectives.
- We create taxonomies to describe policies by implementation style, features used in decision making, decision type, and policy objective.
- We publish the results of our analysis as a searchable database and demonstrate its value through two case studies showing how it assisted with designing extension interfaces for huge-page policies and identified an unmet need for measuring policy importance.

2 Survey of Linux Policies

We perform a large-scale study of policies in Linux with two main focuses. First, we investigate the *implementation* of policies with a lens of customizing them with ML: are policies defined in compile-time constants, changeable values, separate functions, inlined with mechanisms, or as multi-stage algorithms? Second, we look at the *characteristics* of policies: what is the policy trying to decide, what features does it use, and what objectives does it trade off?

Methodology. We identified which files in the Linux source contain substantial policy decisions, and then asked a group of over 20 human annotators that had previously taken an OS course and were comfortable with C to manually read through every line of these files to identify policy-relevant locations. The annotators used LLMs to understand the purpose and behavior of Linux code, and to provide suggestions as to what code might be policy.

Our first challenge was to define *policy* precisely enough to enable consistent identification of implementations. We employed a three-prong test to determine whether a piece of code is policy:

- *Presence of choice.* The code has the ability to make a choice out of a possible set of actions. If there is only one choice, there is no decision to be made.
- *Lack of a dominant action.* If there is a choice that is always the best for all possible workloads, we consider it an optimization and not a policy. There must be a reason for choosing among the set of actions.
- *Expressing choice.* There must be a mechanism to express the choice of action, such as choosing to free resources synchronously or defer it to another thread.

We consider many compile-time constants and configuration settings as policies when they contain implicit choices, such as favoring a workload or a performance goal. For example, enabling zswap for memory compression is a configuration setting that expresses a policy.

We also observed, particularly with file system code, an overlap of features and policies. For example, file systems support numerous features such as write barriers, deferred allocation, and online resizing (see Section 3.5). From these, we identify policies as those that changed code execution at runtime, such as journal checksumming, or change performance, such as block size. Features that do not meet this criteria, such as reserving space for future resizing, were not counted.

We acknowledge that our three-prong test is an informal heuristic, but it served us surprisingly well in our effort to survey a massive code base like Linux. In particular, it allowed our annotators and LLMs to scale and quickly label vast regions of code. Without this test as part of our prompt, LLMs were not able to provide useful results because they struggled with the concept of a "policy". This is unsurprising: we struggled with this same concept, which is precisely why we wrote this paper.

Scope and limitations. Our study looks at several major OS subsystems: CPU and IO scheduling, memory management, power management, and the ext2 and ext4 file systems. We have not yet studied other subsystems with major policies, such as other file systems or networking. We studied the Linux 6.6.42 kernel, as it was the latest long-term support kernel (valuable as a platform for future research building on this work) available when we started. Hence, our results are representative, but no longer the most up to date.

3 Policy Implementation in Linux

Linux exposes a diverse set of policy implementation styles, reflecting both its historical evolution and subsystem-specific design choices. Figure 1 lists the variety of implementation styles we identified.

Broadly, policies may appear as standalone decision functions, as hierarchies or sets of decision routines, as parametric constants, or as logic inlined directly into mechanisms. These styles differ in how easily they can be separated from mechanisms and in how amenable they are to adjustment. CPU scheduling favors decision functions supplemented by

configuration interfaces, enabling users to adjust some behavior without modifying kernel code. In contrast, the memory management subsystem implements policy through a large number of small decision points dispersed across many code paths. These fine-grained decisions often sit beneath broader top-level policies, such as page allocation, page cache replacement, or compaction, and are frequently intertwined with mechanism-heavy logic. As a result, memory policies are harder to isolate, reason about, or tune in a principled way.

This diversity illustrates that Linux does not provide a uniform policy abstraction. Understanding these implementation styles is essential for assessing how suitable existing policies are for data-driven or learned replacements.

3.1 CPU Scheduling Policies

CPU scheduling in Linux occurs at two levels. First, there is an independent scheduler for each CPU that selects the next task to run from one or more `runqueues` according to a set of scheduling classes. Second, there are load balancing policies to move tasks from one CPU's `runqueue` to another.

Linux supports 5 scheduling classes: `STOP`, `deadline`, `real-time`, `fair`, and `IDLE` (more recent versions of Linux also include support for `sched_ext`). These classes form a strict priority hierarchy: if there is task with `STOP` class (reserved for the kernel whenever it needs to stop the CPU to perform some maintenance task like run queue management or task migration), it will be selected above all else. Otherwise, the scheduler will select one of the tasks in the `deadline` class; if there are none, it will then run one of the tasks in the `real-time` class, etc. `IDLE` is just a dummy task that the CPU runs if there are no other tasks.

Each scheduling class provides a clean abstraction. For example, in `sched/core.c`, the `class->pick_next_task()` function returns the next task within the runnable tasks in the corresponding scheduling `class`. This *decision function* style of policy implementation, where a single function encapsulates policy and returns a decision, represents an ideal separation of policy and mechanism, since it hides the complexity of providing the abstraction. For example, it hides the fact that the `fair` class manages a red-black tree and tracks virtual deadlines and runtimes to ensure fairness among tasks.

The third column in Figure 1 lists the number and types of policies in Linux's scheduling subsystem. Decision functions like `pick_next_task()` are common, but there are also numerous deeply thought-out configuration settings and constants. For example, in `include/linux/sched/prio.h`, the constant `MAX_RT_PRIO=100` fixes the number of real-time priority levels, which itself sets the number of queues in the `real-time` class. This is not an arbitrary value. Kernel developers had entire discussions on what this value should be [10, 52]. Having too many priorities requires managing too many queues, which introduces overhead. Having too few priorities means that developers lack the fine-grained control to to express the right priorities within their applications. This is an example of a significant policy that is expressed as a single compile-time constant.

Many other policies found in the `fair` and `real-time` classes are implemented through runtime-configurable `sysctl` parameters. For example, parameters such as `sysctl_sched_base_slice`, `_child_runs_first`, and `_migration_cost` adjust slice lengths, determine fork behavior, and influence load-balancing decisions [37]. Their existence reflects conflicting constraints observed across workloads: child-runs-first was introduced to improve desktop interactive responsiveness, but was made configurable after server and batch workloads reported throughput regressions [45].

Several policies appear as compile-time constants, selected through `Kconfig` rather than `sysctl` because switching them at runtime would break invariants. Examples include `CONFIG_IRQ_TIME_ACCOUNTING` and `CONFIG_PARAVIRT_TIME_ACCOUNTING` which control how time is accounted in environments where interrupts or virtualization distort CPU usage measurements. Some of these options were added after fairness regressions were traced to incorrect accounting under virtualization and high-interrupt workloads [47].

In Linux 6.6, the `fair` class implements EEVDF, which replaced the CFS algorithm for selecting tasks [12]. More recently, Linux 6.12 added `sched_ext` [3], which allows a scheduling policy to be supplied at runtime by a BPF program. It plugs into the existing `sched_class` interface, which needed only two new hooks. That such a class could be added with so little change to the core scheduler is strong evidence that scheduling policy is separable at the class boundary. In short, the scheduler separates policy from mechanism between classes, and most of its remaining policies are parameters or constants. This suggests that replacing some of these policies with learned values or models is not an onerous undertaking.

3.2 Memory Policies

Memory management (MM) for Linux comprises eight major policy areas and numerous smaller ones: page replacement, NUMA migration, swapping and prefetching, compression, kernel allocation, physical page management, and virtual address management. Overall, memory management has nearly 200 distinct policies managing these areas, of which more than 40% are constants or configuration parameters. Unlike the CPU scheduling subsystem, only recently have extension mechanisms been proposed for memory management policies [28, 62, 66], and as a result there are few clean functions implementing its major decisions.

As shown in the fourth column of Figure 1, a large fraction of policy implementation for memory management is through a single decision function. However, unlike CPU scheduling where these single decision functions express large policies that described the overall behavior of entire

Type	Description	Scheduling	Memory	I/O Scheduling	Power	File Sys.
Decision function	Single function implementing entire decision	11	83	7	2	5
Multi-function	Policy spreads among multiple functions	1	4	7	2	0
Inline code	Policy is mixed in with mechanisms	2	16	8	1	2
Constants	Compile-time constants	3	34	7	3	5
Configurations	Exposed parameters	10	52	12	1	51
Control functions	System calls setting policy	0	3	0	0	0

Figure 1. Types of policy implementations in the Linux subsystems, and the number of instances of each implementation.

scheduling classes, the memory decision functions are most often part of a sequence of small decisions that make up the overall policy.

We characterize the larger policy decisions comprised of small decision functions as *multi-function* policies. An example of this is page reclamation. Conceptually, the policy is simply "select a page to evict from the cache according to some replacement algorithm". In Linux, though, the implementation is far more complex. The top-level replacement function `shrink_lruvec()` first decides which policy family to invoke—choosing between the traditional LRU implementation and the newer multi-generation LRU (MGLRU) based on user configuration. Within each path, additional policies determine how the reclamation proceeds. For instance, whether the system should perform page demotion is governed by detection policies such as `inactive_is_low()` The amount of work to perform in a given cycle is computed by `get_scan_count()`, whose decisions depend on policy functions that weigh the relative reclaim cost of anonymous versus file LRU lists in `lru_note_cost()`. Finally, eviction—as implemented in `evict_folios()`, `isolate_folios()`, and `scan_folios()`—selects specific pages following LRU. These routines mix policy heuristics with folio mechanism operations. None of these helper functions alone represents the reclamation policy; rather, the policy emerges from many small decision points.

Another common use of single policy functions is computing *metrics* used by other policies. For example, `lru_note_cost()` is computed from system configuration settings and runtime statistics. These metrics quantify trade-offs and are used in higher-level policies for decision making.

We find there are often multiple distinct but similar policies. For example, there are separate implementations of the OOM handler [11], writeback [60], and reclaim (e.g., `mem_cgroup_soft_reclaim`) at the global and cgroup levels. As a result, there is added complexity in changing policies and the potential for conflict across implementations.

Specific policies are often embedded with mechanisms, such as LRU policy in `shrink_inactive_list` (or `evict_folios` for MGLRU) and prefetching policy in `ondemand_readahead`, which directly invokes I/O operations. This intertwining of policy and mechanism happens because decision inputs and system behavior are both managed by the same data structures (e.g., `scan_control` for reclamation, and `file_ra_state` for prefetching). Consequently, it is efficient and minimizes execution overhead to combine policy and mechanism in one function.

The many configurations and constants in memory management serve two main purposes: provide high-level guidance and control over low-level behavior. About a third of these settings influence memory management at a broad level, for example, `migrate_mode` which shapes how the system handles memory overall. In contrast, other settings act more like rules directly tied to the system's mechanisms. These include `HPAGE_FRAG_CHECK_INTERVAL_MSEC` and reclaim `priority`, which directly affect how the system performs memory operations. As in the scheduling subsystem, assigning values to these configurations is not done lightly; For example, proactive compaction's `compaction_proactiveness` is set to a compromise default value of 20: it expects to keep the CPU overhead low for general workloads, while still aiming to provide some benefits to the hugepage sensitive applications [20].

We generally find that rather than forming a clean separation between mechanism and policy, memory management implements policy as an accumulation of small decisions whose interactions collectively determine system behavior. This makes the behavior of many memory-management policies *emergent* from the many checks and small policies embedded in larger decisions. The resulting behavior can be unpredictable and hard to understand and debug. As an example, prior work showed that Linux's huge page policy, due to these decisions, could spend 100s of milliseconds allocating a single huge page [44].

3.3 I/O Scheduling

I/O scheduling in Linux's multi-queue block layer operates through the struct `elevator_mq_ops` interface. This interface specifies callback functions that the block layer invokes at various points in a request's lifecycle: insertion, merging, dispatch, and completion. Each scheduler provides its own implementation of these callbacks, allowing for diverse scheduling strategies to coexist. I/O schedulers are specialized for different storage characteristics. `Kyber` and `mq-deadline` target fast devices like NVMe SSDs where latency control matters more than seek optimization. `BFQ` targets slower devices, particularly HDDs, where sequential access patterns and fairness guarantees provide significant

benefits. Each scheduler is a collection of single and multi-function decision functions.

Policy decisions are distributed across many callbacks that interact with block layer mechanisms in complex ways. The function `bfq_dispatch_request()` determines when and in what order requests reach the device. It makes policy decisions while also directly manipulating scheduler state (updating budgets, starting timers, moving requests between internal queues). Similarly, `limit_depth()`, which restricts how many requests the block layer can have in flight, is nominally a mechanism callback but the answer encodes policy about fairness and pressure. This interleaving between policy and mechanisms exists partly because I/O scheduling must coordinate with hardware queue management, request merging, and completion handling. The result is that policy modifications often require understanding mechanism.

The fifth column in Figure 1 lists the number and types of policies in Linux's I/O scheduling subsystem. Within each scheduler, policies manifest as decision logic within callback functions, compile-time constants, and runtime tunables. The boundary between constants and tunables reflect deliberate design choices about what users should adjust versus what the implementation assumes.

Constants typically encode physical device properties or safety bounds derived from empirical measurement. `Kyber`'s `kyber_depth` constants fix queue depth limits at 256 for reads and 128 for writes. These values were selected to prevent queue saturation even though they constitute a small fraction of the possible queue depths supported by NVMe [53]. Batch sizes and internal buffer limits similarly reflect hardware constraints rather than policy preferences. These values remain constants because incorrect settings risk device overload or request starvation, and because appropriate values depend on device physics that users rarely understand.

Configuration parameters, the most common type of policy in this subsystem, exist because workload characteristics can vary significantly and users need ways to make informed trade-offs. `Kyber` exposes target latencies for read and write operations, allowing administrators to balance responsiveness against throughput. `Mq-deadline` allows changing expiration times and controls for prioritizing reads over writes. `BFQ` offers control of fairness guarantees (`low_latency`), throughput optimization (`slice_idle`), and penalties for requests that miss their service expectations. These parameters are tunables because optimal values are workload-dependent: settings that maximize throughput on one hardware configuration may degrade performance on another [49].

3.4 Power Management

Frequency scaling in Linux operates through governors that dynamically adjust CPU frequency based on workload demands. The `cpufreq_governor` exposes hooks for initialization, start, stop, and frequency update decisions.

Linux provides six `cpufreq` governors with varying complexity. The `performance` and `powersave` governors are trivial: they statically set frequency to the maximum or minimum allowed values respectively, requiring no decision logic. The `userspace` governor delegates frequency selection entirely to user-space programs through the `scaling_setspeed` sysfs interface. These three governors contain minimal policy, as their behavior is determined by their single design choice.

The `ondemand` and `conservative` governors implement dynamic frequency scaling through the demand-based switching framework (`dbs_governor`). This framework provides common infrastructure for load sampling and frequency adjustment, with governors supplying policy-specific callbacks like `gov_dbs_update` for frequency decisions. Policy in these governors is expressed as timer event handlers that call the `cpufreq` subsystem with a specified power setting: when the timer fires, the policy decides whether to adjust frequency and to what value, and then notifies `cpufreq` of the new value.

The `schedutil` governor differs architecturally from the others. Rather than periodically sampling CPU utilization statistics, it receives load information directly from the kernel scheduler through the Per-Entity Load Tracking (PELT) mechanism. This tight integration with EEVDF eliminates sampling overhead and provides more accurate, timely utilization data [41]. Policy here is expressed as responses to scheduler events: when task utilization changes, the scheduler invokes the governor with current load data, and the governor computes a target frequency. Beyond CPU utilization, `schedutil` implements an I/O wait boost policy to reduce latency for I/O-bound workloads [41].

As shown in the sixth column of Figure 1, power management governors expose a comprehensive set of tunable parameters reflecting their highly configurable decision structure. The `ondemand` and `conservative` governors expose parameters through sysfs: `sampling_rate` controls evaluation frequency, `up_threshold` sets the load percentage triggering frequency increases, `sampling_down_factor` adjusts frequency decrease rate, and `freq_step` (conservative only) controls increment size [22]. These became tunables because optimal values depend on hardware characteristics and workload profiles. Aggressive scaling suits interactive desktops while gradual changes preserve battery on laptops [50].

3.5 File Systems

Of the 79 file system implementations in our kernel, we examined two for policy: `Ext2` as a simple file system and `Ext4` as a modern high-performance one. Textbook descriptions of file systems focus on a few policies: block and inode allocation time/location, block size, and write-back policies, yet we find many more.

We observe that file system policies are defined at three levels. First, there are compile-time constants that decide which file systems are built into kernel (e.g., `CONFIG_EXT4_FS` for `Ext4`), which is a form of policy as file systems make different performance and reliability tradeoffs. Second, there are generic policies that are inherited by all major file systems in the kernel (e.g., `O_DSYNC/O_SYNC` flags for per-file durability). Third, there are FS-specific policies that make each individual file system unique. We focus our study on `Ext2`- and `Ext4`-specific policies as well as the generic policies applicable to them, and summarize a few key observations:

Across both file systems, the only runtime policies implemented as code are `inode` and data block allocators, which strive for locality and load balancing. Both file systems implement block and `inode` allocation policies mixed in with mechanism, making it difficult to extract and replace placement decisions. There are a few supporting decision functions contributing to these policies, such as `ext4_inode_to_goal_block` and `find_group_orlov`.

Many `Ext4` policies are configurable at run time via different interfaces. For example, `Ext4` implements two durability/sync policies: (1) *data sync* to flush all data plus necessary metadata (e.g., file size if it changed) while ignoring less-critical metadata (e.g., timestamps); (2) *full sync* to flush data and all metadata after every write. The policy decision can be made when opening a file (i.e., `O_DSYNC/O_SYNC` flags at `open` syscall), or explicitly with system calls (`fdatasync` or `fsync`). Moreover, synchronous writes can be enforced on the entire file system at mount time (i.e., `mount -osync`).

We find that FS policies are organized across multiple *layers*, each with a different degree of permanence and a different stage in the filesystem lifecycle at which they can be set. The decisions are largely implemented as configuration settings, generally values or flags enabling an operational mode. At the deepest layer, `mke2fs` writes important data to disk that provides information about the structure of the file system by creating data structures such as the superblock, group descriptor, block and inode bitmaps, inode table and the journal. Format-time policy decisions made through `mke2fs` include choices such as block size, inode density, and which feature flags are enabled. These decisions cannot be changed without reformatting the filesystem.

At the middle layer, when a file system is mounted there are a variety of policies that balance performance against enhancing scalability and reliability, such as enabling `journal_async_commit` or setting a custom value for `max_batch_time`. `tune2fs` allows certain features and mount options, such as journal mode or using hashed b-trees to speed up lookups in large directories, to be modified after the filesystem has been created without reformatting.

Finally, a run-time layer exposes a set of per-device runtime knobs through `sysfs`, such as multiblock allocator parameter `mb_max_to_scan` which is maximum number of extents the multiblock allocator will search to find the best extent. These settings can be adjusted on a live, mounted filesystem.

The layered structure of FS policy has an important implication for automated policy optimization: only the mount-time and `sysfs` layers are practically accessible to a tuning system without requiring destructive operations or kernel rebuilds.

3.6 Takeaways

We draw a few broad observations from our study of policy implementation. *First*, most policies are not single functions that can be replaced to change the policy. Rather, they are most often collections of functions and values that may need to be modified in a coordinated fashion. *Second*, constants and tunable values abound across all subsystems, indicating both a large opportunity for improving performance through tuning or self-tuning designs. *Third*, memory management policies stand out with many control-flow dominated policies, indicating the high complexity and unexpected emergent behavior.

4 Policy Characteristics

The previous section surveys the way in which Linux implements various policies. This section instead looks at policies from a more generic lens. We study the inputs (features) and outputs (decisions) of a policy, the primary objectives a policy seeks to optimize, and what objectives it trades off (e.g., memory for latency). These characteristics are vital to generating new data-driven policies, as they describe the features used, types of decisions required, and metrics that should be measured to evaluate new policies.

4.1 Policy Inputs: Features

We categorize policy inputs into five types of features:

1. *Global state.* System-wide conditions such as per-CPU load or `totalram_pages`.
2. *Local object state.* Fine-grained state of a specific task, page, VMA, file or IO request.
3. *History and feedback.* Metrics that encode or aggregate past behavior such as virtual runtime or refault history.
4. *Configuration and constants.* `sysctl`s, hard-coded thresholds, mode switches, and other tunables that set policy behavior.
5. *Hardware information.* Hardware properties such as CPU topology, NUMA distances, or queue depths.

Figure 2 shows the features used by the policies in the five subsystems surveyed. Each subsystem uses a different combination of feature types, as described below.

CPU scheduling features. CPU scheduling mostly uses global and local state, with some history and feedback and hardware information. The major global state features are from the scheduling subsystem, such as per-CPU runqueue

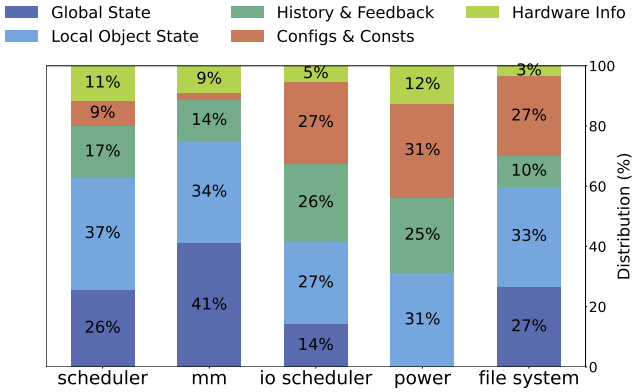

Figure 2. Type and distribution of policy inputs in the surveyed Linux subsystems.

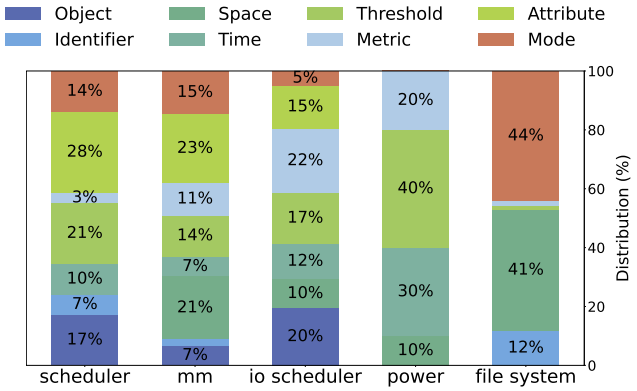

Figure 3. Type and distribution of policy outputs.

length and runnable load, which capture the overall pressure on each CPU and influence whether CPUs should search for work or shed load. The local state features are generally the scheduling state of individual tasks, such as virtual runtime, priority, CPU affinity, and their position in runqueue data structures. For example, ordering policy functions like `__sched_core_less()` and its helper `prio_less()` operate entirely on local object state when they compare per-task attributes such as `core_cookie` (used to control what tasks can share hyperthreads on a core) and static priority to decide which task should run next on a given CPU across all scheduling classes. History and feedback features are captured in per-task aggregated statistics such as `vruntime` deficit or past slice usage, which encode long-term fairness information. There are a few configuration settings and constants, described above in the description of CPU scheduler implementations. Finally, CPU schedulers use hardware information about CPU topology (e.g., core and SMT groups).

Memory management features. Memory management policies make heavy use of global state, such as `totalram_pages` and swap availability that convey overall memory usage. Local state provides information about individual pages or folios and virtual memory areas (VMAs), such as page flags (e.g., referenced, dirty, active) and memory-cgroup usage. Reclaim and migration decisions typically consult these local attributes. History and feedback features are used mostly in page replacement, writeback, and prefetch, to track refaults and reclaim failures. These features capture how past decisions interacted with the workload. Many policies consult configuration knobs and constants, which are often preferences such as `vm.swappiness` or thresholds for detecting conditions requiring action, such as `vm_dirty_ratio` or watermarks, which convey overall memory pressure. Hardware information appears mostly in NUMA policies, which use node layout and distances.

I/O Scheduling Features. I/O scheduling policies make moderate use of global state such as in-flight request counts and aggregate throughput. Local state comprises request attributes (sector address, size, direction, sync/async) and queue-level properties (budget consumption, pending counts, access patterns). Dispatch and merge decisions typically use local attributes in decision-making logic. History and feedback features appear in adaptive schedulers: `Kyber` tracks completion latencies to adjust queue depths, while `BFQ` monitors seek distances to identify sequential workloads. Configuration knobs and constants dominate I/O scheduling policy. Tunables include latency targets, timing parameters, and penalty factors.

Power Management features. Power management policies make light use of global state, primarily CPU utilization averaged across sampling periods. Local state comprises per-CPU properties: current frequency, utilization statistics, and idle time. The `ondemand` and `conservative` governors sample these statistics periodically. History and feedback features appear in adaptive governors: `schedutil` receives utilization data directly from the scheduler's PELT mechanism, and tracks successive I/O requests to ensure I/O-bound tasks receive good frequency despite low mean utilization.

File system features. Many file systems policies are configuration settings that do not have input features. The few policies implemented as code, namely allocators, use a mix of global state, such as the occupancy of allocation groups, as well as local object state about the file or directory being allocated. They also make heavy use of constants, such as `EXT4_FLEX_SIZE_DIR_ALLOC_SCHEME`, which sets the minimum number of groups in a flexgroup. File systems also make limited use hardware information, the RAID strip size.

4.2 Policy Outputs: Decisions

We examined the types of decision outputs expressed by Linux kernel policies to create a taxonomy of decision types and quantify the frequency of each type of decision. These decision types characterize what a policy determines, such as selecting a task in the scheduler, computing an intermediate metric (e.g., OOM `badness_score`), or choosing a policy

mode (e.g., NUMA memory policies). This taxonomy is useful for crafting replacement policies, as it characterizes the decisions that must be made.

In total, we identified the following decision types:

1. *Object*. Selection of a specific object instance by pointer.
2. *Identifier*. Selection of a kernel object by unique identifier.
3. *Space*. A size, count or quantity.
4. *Time*. A temporal quantity, such as a period or timestamp.
5. *Threshold*. A boundary value used for comparison, such as a retry count, memory limit, watermark.
6. *Metric*. A computed value used in decision making.
7. *Attribute*. A property, state or boolean flag of an object.
8. *Mode*. A configurable system state or mode of operation.

Figure 3 shows the distribution of these types for each subsystem. Below we expand on these results.

CPU scheduling decisions. CPU scheduling policies most often output attributes, thresholds, and time-based values. This distribution aligns with the nature of scheduling, which allocates CPU time and classifies tasks by their behavior and other properties. The subsystem produces many attribute outputs, such as determining whether one entity outranks another according to priority, deadlines or core-cookie compatibility. These policy decisions do not select the task but instead provide support in the selection logic for which tasks are eligible or preferred.

By contrast, object and identifier outputs show up primarily in CPU-selection routines such as `pick_next_task_fair`, `pick_next_dl_entity`, and `select_task_rq_rt`, where the scheduler commits to a specific runnable entity or a specific CPU. These outputs are fewer because such decisions are built from many internal checks that shape eligibility and timing. The threshold, metric, and space outputs provide the context the scheduler needs to make decisions such as determining how much load to pull during balancing, while mode outputs toggle qualitative behaviors like child-runs-first or alternate accounting rules.

Memory management decisions. Memory management decisions span the entire taxonomy of output types. This broad mixture of output types reflects its implementation, where behavior emerges from numerous micro-policies rather than a small number of coarse-grained decisions.

Many MM policies naturally generate object outputs: page allocation, NUMA migration, and hugepage management all revolve around selecting specific pages as part of their decision logic. They also produce a variety of attribute and metric outputs, which often appear as intermediate decisions within a longer policy chain, for example, determining whether it is the time to trigger an action (e.g., `should_proactive_compact_node`), or computing quantitative signals such as `vmpressure_level` for signaling higher-level policies. MM policies also frequently emit space outputs, governing how much memory to operate on (e.g., the amount of data to prefetch, or the size of SLUB objects). Time outputs likewise play an important role, controlling the interval at which a policy should run, a central trade-off in threads such as THP collapse (`khugepaged`) and proactive compaction (`kcompactd`), where CPU overhead must be balanced against memory footprint.

I/O Scheduling decisions. I/O scheduling decisions produce object, metric, time, and attribute outputs, reflecting a decision structure centered on request ordering and queue service. Object outputs are common, as a typical decision is which queue to service next. Similarly, schedulers choose which requests to service, as in the `bfq_choose_req()` function. Metric and time outputs guide scheduling indirectly. `Kyber` has numerous policy functions that compute latency statistics and queue depth targets. Several policies output boolean or attribute classifications, such as policies that determine whether to idle the IO queue to wait for more requests as in `bfq_better_to_idle()`. Another set of policies identifies I/O patterns, such as seek-heavy applications, as attribute outputs which are used for allocating I/O budgets.

Power management decisions. The primary power management decision is to select a frequency for the CPU, which we classify as a metric. Attribute outputs (boolean classifications) trigger frequency transitions. The up and down frequency threshold constants define utilization boundaries indicating high-load or low-load states, creating a hysteresis band within which frequency remains stable. `schedutil` detects I/O-bound workloads (boolean attribute), feeding into subsequent frequency computations.

File system decisions. File system allocation policies, like other allocators, return identifiers as their decision. Many policies relate to space, such as the size of data to scan during allocation, and time, such as the frequency of writeback. The most common decision is a mode, which sets which codepath the FS should take on various operations, such as whether to use a separate device for journaling.

4.3 Policy Objectives

We observe that most policies have a primary *objective*, a property they seek to optimize, and balance that against one or more costs. For example, caching policies have the goal of low latency but balance that against the cost of increased memory usage. Likewise, a CPU scheduler may optimize for low latency while also considering the effect of a policy on fairness and isolation. Although each policy targets a specific goal or workload, certain objectives recur across subsystems, as shown in Figure 4. This data can inform data-driven policies by indicating what objectives to maximize and which properties to monitor; it may be easy to build an I/O scheduler that is faster for one application yet inadvertently starves others.

Type	Sched.	Mem.	I/O	Power	File Sys.
Latency	✓	✓	✓	✓	✓
Throughput	✓	✓	✓	✓	
Fairness	✓		✓		
Isolation	✓	✓			
Stability		✓			
Real-time	✓				
Memory Usage		✓			
CPU Overhead		✓			
Power Saving				✓	
Reliability					✓
Locality					✓

Figure 4. Types of objective and the corresponding subsystem where they are present.

CPU Scheduling objectives. Every Linux CPU scheduling policy has a set of objectives shaped by workload requirements. The EEVDF general-purpose scheduler strives to provide low-latency dispatch, performance isolation, and fairness. These goals are also the costs to be balanced: fairness, throughput, and latency. As a result, portions of the EEVDF scheduler work against each other, with some policies seeking to decrease latency at the cost of decreased fairness, while other policies improve fairness but harm latency. Many parameters adjust this balance, such as `sysctl_sched_base_slice`, which sets the base time quantum.

Another example of competing objectives are the goals of the scheduler and the load balancer. For instance, the `fair` scheduling class aims to allocate the CPU fairly among all tasks in this class. However, the periodic load balancer often moves tasks from one CPU to another to ensure that no CPU is not under/overloaded. As fairness is determined per-CPU, it can be compromised by migration in the pursuit of better latency and throughput.

The real-time schedulers optimize different objectives: `deadline` optimizes for real-time guarantees and throughput, but incurs costs of low fairness and higher latency. The `round robin` scheduler provides fairness, but increases latency. The `priority` scheduler decreases latency for high-priority tasks, but at the cost of fairness.

Memory management objectives. Memory management balances between memory usage, latency, stability (whether the kernel panics on OOM), throughput, and isolation, each of which pulls the system in different directions.

Memory resources are a central optimization target in Linux. Broadly, memory optimization falls into two goals: improving memory efficiency and reducing memory footprint. Memory efficiency relates to how effectively limited memory is used for application performance. Increasing reclaim efficiency to find inactive pages often requires more scanning—for example, through policies like `get_scan_count`—but scanning can increase allocation latency. Allowing more memory to be used can increase application performance, but decrease stability by making OOMs more likely.

Beyond these major trade-offs, there are many less common ones. Isolation through memory cgroups strengthens workload boundaries but carries performance overhead. Stability through ensuring memory availability often requires conservative behavior that limits performance. Limiting CPU overhead requires restricting background work, such as batching dirty pages[61], which saves CPU cycles but increases latency.

I/O Scheduling objectives. I/O scheduling policies have a set of performance-related objectives, often focusing on particular cases. For example, `bfq` has policies that seek to prevent starvation, reduce latency, increase throughput, provide fairness across applications. The costs it considers are similar: decreased throughput, CPU utilization, and increased latency. `Kyber` is a bit different in that it seeks *bounded* latency, and balances this against underutilization, decreased throughput, fairness, and CPU utilization. Finally, the `MQ` scheduler's internal policies try to bound latency and ensure fairness, but balance that against the costs of throughput and CPU overhead.

Power Management objectives. All power management policies generally share the same two objectives: increasing performance or saving energy. The costs are the converse: decreased performance and increased energy use. Different policies trade how they prioritize these. For example, the `ondemand` governor prioritizes performance, and rapidly increases frequency, while the conservative governor prioritizes power and is much slower to increase power. The `schedutil` governor seeks to boost speed quickly for tasks waking up from I/O while lowering frequencies to save power during idle periods.

File system objectives. File systems are unique among the subsystems because reliability (i.e., preserving data after a crash) is a major objective. In addition, FS allocators trade off locality against fragmentation. While many policies make the common tradeoff of throughtput and latency, such as the delay for writeback, others trade off the cost of more data reliability against performance. For example, `Ext4` async commit improves performance by allowing transactions to write the journal commit block earlier, which increases the chance of a transaction being unrecoverable after a failure.

5 Case studies

The results from our study of Linux policies provoke further use and research in several directions. In this section we describe three uses of our findings.

5.1 New Interfaces for Memory Management

Our study showed that Linux memory management has relative few decision functions amenable to replacement with learned policies. We used this observation to develop two extension interfaces for different aspects of huge pages.

Huge-page compaction interface. Huge page compaction creates huge pages by finding 2 MB physical pages with movable 4 KB pages and then relocating valid data elsewhere, creating empty 2MB huge pages. We designed an extension interface that exposes the key decisions of the compaction process to eBPF programs:

- `should_compact_zone`: decide whether a memory zone is suitable for compaction.
- `fragmentation_score`: return a score for a memory zone indicating how fragmented it is.
- `select_migration_source`: choose which huge page to migrate 4 KB pages from.
- `select_migration_target`: choose the destination for migration.
- `compact_finished`: determine whether to terminate or continue compaction after each step.

With this eBPF interface, we wrote a more-aggressive compaction policy targeting workloads with pinned buffers, and also applied a code evolution framework [9, 46, 56] that uses the Claude Opus 4.7 LLM to synthesize improved policies automatically.

We evaluate the LLM-generated policies on a CloudLab c220g2 node: two 10-core Intel Xeon sockets at 2.6 GHz with two threads per core (40 logical CPUs), 160 GB of memory. On an allocation-heavy database workload, our interface adds about 1% in overhead, but the hand-written policy was able to maintain an 8 percentage-point higher success rate in huge page allocation with 11% lower CPU utilization due to scanning fewer pages, and reduced huge-page allocation p99 latency by 11%. Similar to the hand-written policy, the LLM-synthesized policy was about 8.5 percentage points better than the default policy.

Huge-page allocation interface. We also looked at the policies for allocating huge pages. Through a comprehensive study of both proposed optimizations [36, 44, 48] and the default Linux implementation, we observed that hugepage allocation follows an explicit decision-making workflow. Nevertheless, previous hugepage extension implementations [55] focus primarily on the initial intent selection step (should huge pages be used for an allocation?) while providing little control over the subsequent decision flow (if no huge pages are available, should they be created?). From this observation, we identified and abstracted the core decision points in the allocation process and exposed them as programmable interfaces:

- `select_intent`: for an allocation: try hugepage, skip, or evaluate cost?
- `on_failure`: if allocation fails: compact, reclaim, or fall back?
- `cleanup_effort`: Decide how hard to push on compaction and reclaim before retrying
- `on_base_alloc`: After final fallback to 4KB page: request background promote operation?
- `on_success`: after successful hugepage alloc: update cost models or learning signals

Each handler is called with a context structure containing information about the operation and global memory management state, which allows flexibly integrating custom features. Our reference implementation mirrors default Linux behaviors, and uses features like fragmentation score and pressure level. While providing behavior parity with default Linux, this interface adds only 0.7% overhead per THP fault.

5.2 Policy Importance: CPU scheduling

Given the large number of policies, determining which ones are important for a given workload is a significant challenge. There have been numerous works seeking to autotune collections of parameters [26, 31, 33, 34], which have been successful at tuning collections of parameters for better performance. However, our study reveals more than 100 policies implemented as decision functions (Figure 1). In the context of learned policies, this reveals an open challenge: how to know which decision functions are important for a given workload?

We propose determining the importance of policies by measuring what happens *when they go wrong*. Essentially, we modify decision functions to fuzz the output and return a slightly different decision, and measure the resulting performance. Our hypothesis is that if the policy is significant, making different decisions will have a measurable impact on performance.

We implement this approach in the Linux EEVDF scheduler at the core decision point of where to run a task. We modified code inside `select_task_rq_fair()`, which decides which CPU a task runs on when it wakes up. Instead of picking the usual best CPU, we randomly choose among three scheduler-tracked options: CFS's target CPU, the previous CPU for a task, and most recent CPU used.

We evaluate the fuzzed scheduler on workloads stressing different aspects of scheduling:

- *CPU*: `stress-ng -cpu` creates compute intensive tasks that continuously compete for CPU time.
- *Spawn*: `stress-ng -spawn` repeatedly creates and destroys processes to task creation and wakeup events.
- *I/O*: `stress-ng -io` generates storage-intensive tasks that frequently block while waiting for I/O.
- *Mixed*: runs all three other workloads concurrently.

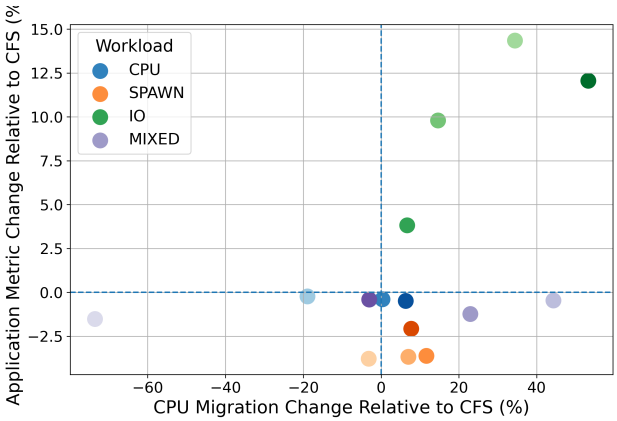

Figure 5. Performance shifts under randomized CPU scheduling. Y-axis shows workload performance change and X-axis shows change in number of migrations. Darker colors indicate more threads.

We ran experiments on a server with 40 logical CPUs and 40–320 workers. With more workers, more runnable tasks compete and cause scheduling decisions such as task placement and migration to occur more frequently.

The results in Figure 5 show that performance sensitivity to randomized decision perturbations is workload-dependent, suggesting an instance-optimal policy could specialize for specific workloads. The I/O workload performance varies widely with the randomized core selection policies, indicating it is an important policy for the workload. In contrast, throughput-oriented CPU tasks and Spawn tasks see little change in performance, indicating little dependence on the policy.

Overall, these results show that randomizing scheduling decisions does affect the system, but not equally across workloads. This motivates instance-optimal policies that should be considered for sensitive workloads.

5.3 Policy Implementation Database

To demonstrate the utility of our dataset, we created a searchable database of all the 338 policies we discovered and make it publicly available. [1] The database provides granular details for each policy, including its location (function name), modification layer (e.g., when a policy can be modified), inputs and outputs value, classification information (e.g., policy category, and implementation style), and trade-offs. Users can easily query the database by subsystem, implementation type, inputs, outputs, and goals.

One motivation for constructing the policy database was to enable analyses that are difficult to perform by reading subsystem code in isolation. As an example, we investigated 49 policies targeted at latency, summarized by subsystem and implementation type (Table 6). Memory Management has the most latency-impacting policies and contributes 39/49 of total policies, where it manages the hierarchy of caches, swap, compression, and prefetch. A notable structural observation is that memory's latency policies split almost evenly between tunable parameters (23) and functions (16), while I/O scheduling's are predominantly tunables (4/6). This reflects a fundamental difference in how the two subsystems express latency intent: I/O schedulers expose explicit latency targets (`kyber_latency_targets`, `read_expire`) that operators set declaratively, while the memory subsystem encodes latency policy implicitly across many small decision functions whose collective behavior produces latency outcomes that are difficult to predict or control directly.

It also displays an opportunity for cross-subsystem coordination. A clear example is the path for memory's dirty writeback policies. First, every file system write that goes through the page cache is subject to this policy: when `vm.dirty_background_ratio` is set aggressively low to reduce write latency, it increases writeback frequency, which affects file system journal commit timing and the durability guarantees that policies like `data=ordered` rely on. Second, the resulting writeback I/O competes directly with application I/O in the block layer: a burst of background flushes can flood an already-loaded I/O queue and displace latency-sensitive requests that I/O scheduling policies like `kyber_latency_targets` and `read_expire` were configured to protect, worsening the very latency the memory policy aimed to improve. Because these subsystems do not talk to each other, a tuning choice meant to improve memory latency can accidentally ruin other subsystem's performance. We aim to make similar interactions visible.

Subsystem	Policies	Config/Const	Func/Inline	Primary Cost
Memory	39	23	16	Available memory / Disk I/O
I/O Scheduling	6	4	2	Throughput
CPU Scheduling	2	0	2	Throughput / Fairness
Power Management	1	0	1	Energy
Filesystem	1	0	1	Fragmentation

Figure 6. Latency-targeting policies by subsystem and implementation style.

6 Related Work

A growing body of work replaces or tunes systems heuristics with learned policies. Examples span I/O admission control (LinnOS [23], Heimdall [35]), cache eviction (Parrot [42], Glider [57]), prefetching (Hashemi et al. [24], Voyager [58], Pythia [4]), memory allocation (Llama [43]), data placement in tiered memory and storage (Kleio [15], Sibyl [59]), and load balancing (MLLB [7]). Other systems make policies easier to replace or tune: pluggable frameworks such as sched_ext [3], ghOSt [25], and Syrup [32] let developers supply new scheduling policies, while tools such as Wayfinder [31] and AutoOS [6] search for good settings of existing parameters. These systems are effective within their target subsystems, but they share two assumptions: they address one policy at

[1] `https://ldos-project.github.io/Policy_Dataset/`

a time, and they treat each policy as a self-contained unit, realized as a single decision function or exposed as a tunable parameter. Our study shows that many Linux policies, particularly in memory management, do not fit this mold.

LAKE [18] provides a general framework for integrating ML into the kernel. The primary focus of LAKE was on OS design to enable neural policies, specifically how to capture and manage features and how to enable ML policies to access accelerators such as the GPU. Importantly, LAKE also treats each policy as a monolithic unit, and draws conclusions about OS structure based on on a small number of policies.

Most recently, researchers have applied LLM-guided program search to systems problems, synthesizing or evolving heuristics from performance feedback [9, 16, 21], including Linux scheduling policies deployed through sched_ext [65]. These approaches reduce the human effort needed to find a better policy, but they too target one policy at a time and depend on an existing interface or simulator through which candidate policies can be deployed and evaluated.

Rather than optimizing individual, well-studied policies, we step back to examine OS policy broadly. Our goal is to inform an OS design whose interfaces allow any policy to be adapted and explored, whether its replacement is handwritten, synthesized, or learned.

7 Conclusions

Well-designed policies provide much of the benefit of using an OS, such as efficient multiplexing of hardware and protection. In this section we discuss how our findings may be helpful in the design of new policies and OSes.

Guidance for Learned Policies. Our taxonomies of policy inputs (features), outputs (decisions), and objectives provide concrete guidance for researchers and practitioners designing learned policies [9, 16, 23].

First, Figure 2 shows that across all surveyed subsystems, decisions rely predominantly on local object state and history/feedback metrics rather than global system telemetry. This suggests that models designed around localized and historical features can be highly effective rather than trying to ingest global system states.

Second, the policy objectives and tradeoffs in Section 4.3 serve as a checklist for defining multi-objective reward or loss functions in learned policies. For instance, memory management policies trade off latency not just against throughput, but against CPU overhead and OOM stability. If a policy is trained to optimize only for page allocation latency, it risks degrading CPU utilization or violating memory availability constraints. Multi-objective models can use these documented tradeoffs to formulate realistic reward functions.

Third, Figure 3 helps inform the choice of ML model architecture. Scheduling policies primarily output relational attributes (comparing two tasks), suggesting that learning-to-rank or pairwise classification models are suitable. In contrast, memory management policies produce a heterogeneous mix of space, time, thresholds, and objects, requiring models and architectures capable of outputting scalar sizes or temporal intervals.

Implementation Barriers. Our study of implementation styles (Section 3) highlights a structural barrier that make production OSes like Linux difficult to optimize: the large number of policies that spread across many files and function boundaries. One consequence is that behavior is emergent: a high-level policy is the net result of many micro-decisions that compose in ways that are not well understood. This prevents developers from simply replacing a single function to implement a new policy because it is not clear how that policy will impact others. To address this, OSes should refactor their policy code to separate policy from mechanism [39], perhaps through an interface like decision functions.

Design Principles for Future OSes. Based on these findings, we propose three design principles to guide the architecture of next-generation OSes:

1. **Decouple Telemetry from Action Selection:** Future OSes should extract local state and historical metrics separately from policy execution. This allows external policies (whether eBPF, user-space agents, or ML models) to read features without contending on kernel data structures. LAKE [18] is an example of this philosophy.
2. **Declarative Policy Interfaces:** Instead of embedding policy inside mechanisms, operating systems should expose explicit decision interfaces where the kernel mechanism guarantees safety (e.g., respecting CPU affinity and resource limits) while the policy programmatically selects actions. Our case study on compaction shows that exposing structured hooks allows safe, external policy logic to optimize execution with low overhead.
3. **Unified Tradeoff Coordination:** Many policies make similar tradeoffs, such as latency for throughput. Future OSes should refactor these into generic policies that make the same tradeoff in a consistent fashion. This avoids conflicting optimizations across subsystems.

These principles combine to form a new approach to OS design.

Acknowledgments

We would like to thank Dhruv Pratik Desai, Yashwanth Ranjan Singaravel, Amitav Rawat, Rajit Mahesh, Monisha Krothapalli and Leshna Balara for their work studying policies. We thank our shepherd Han Dong for valuable feedback preparing the final manuscript. This work was supported by NSF "Expeditions in Computing: Learning Directed Operating System (LDOS)" Grant No. 2326576, and NSF CSR Grants No. 1943204 (CAREER), 2402858, and 2505085.

References

[1] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A Machine Learning Framework to Improve Storage System Performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. 94–102. https://doi.org/10.1145/3465332.3470875

[2] Daroc Alden. 2024. Weighted memory interleaving and new system calls. https://lwn.net/Articles/969379/. https://lwn.net/Articles/969379/

[3] Daroc Alden. 2024. What's scheduled for sched_ext. https://lwn.net/Articles/974387/. https://lwn.net/Articles/974387/

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. ACM, 1121–1137. doi:10.1145/3466752.3480114

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. 267–283. https://doi.org/10.1145/224056.224077

[6] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, Qi Guo, and Yunji Chen. 2024. AutoOS: Make Your OS More Powerful by Exploiting Large Language Models. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24) (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 7511–7525.

[7] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine Learning for Load Balancing in the Linux Kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*. ACM, 67–74. doi:10.1145/3409963.3410492

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374. *arXiv:2107.03374* (2021).

[9] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. https://arxiv.org/abs/2510.06189. arXiv:2510.06189 [cs.AI] https://arxiv.org/abs/2510.06189

[10] Jonathan Corbet. 2008. SCHED_FIFO and realtime throttling. https://lwn.net/Articles/296419/. https://lwn.net/Articles/296419/

[11] Jonathan Corbet. 2018. Teaching the OOM killer about control groups. https://lwn.net/Articles/761118/. https://lwn.net/Articles/761118/

[12] Jonathan Corbet. 2023. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/. https://lwn.net/Articles/925371/

[13] Jonathan Corbet. 2025. Custom out-of-memory killers in BPF. https://lwn.net/Articles/1019230/. https://lwn.net/Articles/1019230/

[14] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[15] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. ACM. doi:10.1145/3307681.3325398

[16] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators!. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*. 51–60. https://doi.org/10.1145/3772356.3772413

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. 251–266. https://doi.org/10.1145/224056.224076

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

[19] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. 2021. SmartOS: Towards Automated Learning and User-Adaptive Resource Allocation in Operating Systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 48–55. https://doi.org/10.1145/3476886.3477519

[20] Nitin Gupta. 2019. RFC: mm: Proactive compaction. https://patchwork.kernel.org/project/linux-mm/patch/20190816214413.15006-1-nigupta@nvidia.com/. https://patchwork.kernel.org/project/linux-mm/patch/20190816214413.15006-1-nigupta@nvidia.com/

[21] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. https://arxiv.org/abs/2510.27176.

[22] Junyeong Han and Sungyoung Lee. 2020. Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones. *IEEE Access* 8 (2020), 11031–11045. https://doi.org/10.1109/ACCESS.2020.2965548

[23] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 173–190. https://www.usenix.org/conference/osdi20/presentation/hao

[24] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 1919–1928. https://arxiv.org/abs/1803.02329

[25] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, 588–604. doi:10.1145/3477132.3483542

[26] Oracle Inc. [n. d.]. bpftune - BPF driven auto-tuning. https://github.com/oracle/bpftune. https://github.com/oracle/bpftune

[27] Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. 117–130. https://doi.org/10.1145/502034.502046

[28] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 397–408. https://www.usenix.org/conference/atc24/presentation/jalalian

[29] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR)*.

[30] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. 1997. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling

of Independent Activities. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*. 198–211. https://doi.org/10.1145/268998.266689

[31] Alexander Jung, Hugo Lefeuvre, Charalampos Rotsos, Pierre Olivier, Daniel Oñoro Rubio, Felipe Huici, and Mathias Niepert. 2021. Wayfinder: towards automatically deriving optimal OS configurations. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 115–122. doi:10.1145/3476886.3477506

[32] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, 605–620. doi:10.1145/3477132.3483548

[33] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proc. VLDB Endow.* 15, 11 (July 2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[34] Konstantinos Kanellis, Sujay Yadalam, Hayden Coffey, Shivaram Venkataraman, and Michael Swift. 2026. From Good to Great: Parameter Tuning in Memory Tiering Systems. *IEEE Trans. Comput.* (2026).

[35] Daniar Heri Kurniawan, Rani Ayu Putri, Peiran Qin, Kahfi S. Zulkifli, Ray A. O. Sinurat, Janki Bhimani, Sandeep Madireddy, Achmad Imam Kistijantoro, and Haryadi S. Gunawi. 2025. Heimdall: Optimizing Storage I/O Admission with Extensive Machine Learning Pipeline. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. 1109–1125.

[36] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[37] Hyeonmyeong Lee, Sungmin Jung, and Heeseung Jo. 2022. STUN: Reinforcement-Learning-Based Optimization of Kernel Scheduler Parameters for Static Workload Performance. *Applied Sciences* 12, 14 (July 2022), 7072. https://doi.org/10.3390/app12147072

[38] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. 2016. Deadline scheduling in the Linux kernel. *Software: Practice and Experience* (2016), 821–839. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2335

[39] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/mechanism separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. 132–140. https://doi.org/10.1145/800213.806531

[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158

[41] Linux. 2024. Schedutil - The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/scheduler/schedutil.html. https://www.kernel.org/doc/html/latest/scheduler/schedutil.html

[42] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning (ICML '20) (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 6237–6247.

[43] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM. doi:10.1145/3373376.3378525

[44] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 593–608. https://www.usenix.org/conference/atc22/presentation/mansi

[45] Ingo Molnar. 2009. sched: Turn off child_runs_first. https://lkml.org/lkml/2009/9/9/254. https://lkml.org/lkml/2009/9/9/254

[46] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. https://arxiv.org/abs/2506.13131. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[47] Venkatesh Pallipadi. 2010. Proper kernel irq time accounting. https://lwn.net/Articles/405889/. https://lwn.net/Articles/405889/

[48] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 347–360. https://doi.org/10.1145/3297858.3304064

[49] Steven L. Pratt and Dominique A. Heger. 2004. Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers. In *Proceedings of the Linux Symposium*, Vol. 2. 425–448. https://www.kernel.org/doc/ols/2004/ols2004v2-pages-139-162.pdf

[50] Red Hat, Inc. 2024. Power Management Guide: CPUfreq Governors. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/power_management_guide/cpufreq_governors. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/power_management_guide/cpufreq_governors

[51] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical Discoveries from Program Search with Large Language Models. *Nature* 625 (2024), 468–475. doi:10.1038/s41586-023-06924-6

[52] Steven Rostedt et al. 2005. [RFC][PATCH] Make MAX_RT_PRIO and MAX_USER_RT_PRIO configurable. Linux kernel mailing list thread, https://lkml.iu.edu/0507.3/0792.html.

[53] Omar Sandoval. 2017. [RFC PATCH 4/4] blk-mq: introduce Kyber multiqueue I/O scheduler. https://lore.kernel.org/all/d076751c9aefa1c1179b4f36ef69e96ec8803035.1489787289.git.osandov@fb.com/. https://lore.kernel.org/all/d076751c9aefa1c1179b4f36ef69e96ec8803035.1489787289.git.osandov@fb.com/

[54] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 213–227. https://doi.org/10.1145/238721.238779

[55] Yafang Shao. 2025. [PATCH 0/2] bpf: Support hugepage allocation policy via bpf. https://lore.kernel.org/bpf/20250930055826.9810-1-laoar.shao@gmail.com/. https://lore.kernel.org/bpf/20250930055826.9810-1-laoar.shao@gmail.com/ Linux Kernel Mailing List.

[56] Asankhaya Sharma. 2025. Openevolve: an open-source evolutionary coding agent. https://github.com/codelion/openevolve. https://github.com/codelion/openevolve

[57] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*. ACM, 413–425. doi:10.1145/3352460.3358319

[58] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A Hierarchical Neural Model of Data Prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS '21). ACM. doi:10.1145/3445814.3446752

[59] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. ACM. doi:10.1145/3470496.3527442

[60] Greg Thelen. 2010. memcg: per cgroup dirty page accounting. https://lwn.net/Articles/408349/. https://lwn.net/Articles/408349/

[61] Fengguang Wu. 2010. [PATCH 07/35] writeback: per-task rate limit on balance_dirty_pages(). https://marc.info/?l=linux-mm&m=129225297106082&w=2. https://marc.info/?l=linux-mm&m=129225297106082&w=2 linux-mm Mailing List Archive.

[62] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novakovic, Alex Snoeren, and Kim Keeton. 2025. PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. https://www.usenix.org/system/files/atc25-yelam.pdf

[63] Yiying Zhang and Yutong Huang. 2019. "Learned": Operating Systems. *SIGOPS Oper. Syst. Rev.* 53, 1 (July 2019), 40–45. doi:10.1145/3352020.3352027

[64] Yifan Zhang, Xinkui Zhao, Ziying Li, Guanjie Cheng, Jianwei Yin, Lufei Zhang, and Zuoning Chen. 2025. Integrating Artificial Intelligence into Operating Systems: A Survey on Techniques, Applications, and Future Directions. https://arxiv.org/abs/2407.14567. arXiv:2407.14567 [cs.OS] https://arxiv.org/abs/2407.14567

[65] Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. 2025. Towards Agentic OS: An LLM Agent Framework for Linux Schedulers. https://arxiv.org/abs/2509.01245.

[66] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. 2025. cache_ext: Customizing the Page Cache with eBPF. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 462–478. https://doi.org/10.1145/3731569.3764820