

Unfair by Design: eBPF-Based Scheduling of Mixed Database Workloads

Carl-Elliott Bilodeau-Savaria
McGill University
Montreal, Canada
carl-elliott.bilodeau-savaria@mail.mcgill.ca

Jan Kristof Nidzwetzki
FernUniversität in Hagen
Hagen, Germany
jan-kristof.nidzwetzki@fernuni-hagen.de

Stefanie Scherzinger
University of Passau
Passau, Germany
stefanie.scherzinger@uni-passau.de

Bettina Kemme
McGill University
Montreal, Canada
bettina.kemme@mcgill.ca

ABSTRACT

Modern database systems increasingly execute mixed workloads where background tasks should use only spare CPU capacity without affecting latency-critical requests. Existing OS schedulers, despite supporting priorities, do not reliably isolate high-priority tasks and remain vulnerable to priority inversion. Recent advances in application-informed OS primitives, including the Linux eBPF-based sched_ext scheduler class, motivated us to develop **UFS**, a two-tier `sched_ext` scheduler designed for mixed database workloads. UFS confines background tasks to idle CPU capacity and immediately preempts them when time-sensitive work arrives, while supporting configurable scheduling weights within both tiers. To mitigate priority inversion, UFS incorporates application-level hints via eBPF maps, allowing background tasks holding locks needed by time-sensitive tasks to proceed when necessary. Our PostgreSQL extension-based integration shows that, under mixed workloads, UFS improves throughput for high-priority tasks by up to 2× and halves tail latency compared to existing Linux scheduling options. An initial MariaDB integration demonstrates broader applicability.

PVLDB Reference Format:
Carl-Elliott Bilodeau-Savaria, Jan Kristof Nidzwetzki, Stefanie Scherzinger, and Bettina Kemme. Unfair by Design: eBPF-Based Scheduling of Mixed Database Workloads. PVLDB, 20(3): 649 - 662, 2026.
doi:10.14778/3860016.3860030

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DISLMcGill/unfair_scheduler.

1 INTRODUCTION

Database management systems (DBMS) increasingly provide support for in-DBMS data analytics and data science processing [10, 13, 16, 31, 40, 48]. In particular, user-defined functions (UDFs) [12, 13] have become popular for running not only complex SQL pipelines and materialized view refreshes, but also ML inference [31, 40, 48] and ML training tasks within the DBMS, close to the data [10, 16, 63]. These tasks are often long-running and CPU-bound. At the same time, DBMS must continue to ensure timely responses for interactive queries (often with bursty CPU usage patterns) and for certain CPU-bound tasks such as complex interactive queries.

While the prevalence of such mixed workloads has led to the development of specialized hybrid transaction and analytical processing (HTAP) systems [24, 45, 47, 51], this paper analyzes how mixed workloads can be supported by traditional off-the-shelf DBMS. Here, long-running analytics can significantly delay short requests, leading to SLA violations. However, many of these analytics tasks might not require stringent response times, opening the door for priority-based scheduling, a technique widely explored by the database and the operating system (OS) communities. The scheduler executes non-time-critical tasks when free CPU cycles are available and preempts them when high-priority tasks need the CPU.

Work on priority-based scheduling within the database system [4, 19, 38, 59] often puts a particular focus on the priority inversion problem [49], where a low-priority task that holds a lock is preempted, causing a high-priority task waiting for this lock to be unduly delayed. The general solution is to avoid the preemption of low-priority tasks as long as they hold locks.

The OS community, in contrast, has developed a plethora of priority-based scheduling techniques. Today's Linux kernels feature `cgroups`, with a sophisticated mechanism for implementing complex hierarchical resource-control structures (e.g., CPU, I/O, and memory limits). The default scheduler in Linux, EEVDF, provides proportional sharing of the CPU and can be combined with `cgroups` to provide processes with resources proportional to the priority of their group. Linux also has special schedulers for real-time tasks or to exploit idle CPU resources for background tasks.

The current state-of-the-art in modern OS schedulers motivated us to revisit OS-based scheduling for mixed database workloads. One advantage of OS-based scheduling is that it can make holistic decisions. It can consider non-DBMS processes or threads, e.g., kernel threads, and it can grant low-priority tasks the CPU when high-priority tasks wait for user input or I/O, not only when they hold locks. Furthermore, offering an OS solution is promising for existing DBMS, since the integration of a scheduler into a DBMS is a significant engineering effort. Given all these advantages, an OS-based scheduler must nevertheless be able to handle the priority

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 3 ISSN 2150-8097.
doi:10.14778/3860016.3860030

inversion problem, yet this only materializes in the DBMS. As such, some form of communication between DBMS and OS is required.

State-of-the-art OS schedulers. In a first step, we explored the existing OS scheduling options for mixed database workloads, not yet even targeting the priority inversion problem. Our goal was to be completely *unfair* on request: background tasks should only get the CPU when no time-sensitive tasks require computation. However, when there are different types of time-sensitive tasks, they should get a proportional share of the CPU. Unfortunately, and as also repeatedly observed by previous work [28, 34, 65], the Linux scheduling options do not achieve the desired outcome. Our analysis revealed that in the presence of CPU-bound tasks, even if low priority, EEVDF concentrates CPU-bursty tasks on only a few cores, negatively affecting their performance, while Linux's real-time scheduling delays CPU-bursty tasks when CPU-bound tasks have the same priority.

Motivated by the user-space-delegated scheduling paradigm [20], recent Linux kernels now allow new scheduling algorithms to be implemented through the `sched_ext` interface as eBPF programs, which can be loaded seamlessly at runtime, creating a very appealing opportunity for application-specific scheduling. Example implementations are `scx_layered` for large time-sensitive production workloads or `scx_lavd` for latency-critical gaming applications.

Selectively Unfair Scheduler. The limitations of Linux's standard schedulers and the potential of `sched_ext` motivated us to design *Unfair Scheduler* (UFS), a `sched_ext`-based scheduler for mixed database workloads. UFS is designed to be *selectively unfair*: It grants high-priority work immediate access to the CPU when runnable, while scheduling low-priority tasks only when the CPU is idle. To do so, UFS introduces two *scheduling tiers*: "time-sensitive" for high-priority work and "background" for low-priority work. As database systems can run a wide range of both time-sensitive and background workloads, UFS also supports `cgroups`, enabling fine-grained priority-based proportional sharing within each tier.

Priority inversion. UFS actively manages the priority-inversion problem. Recent work [19] indicates that conflicts are no longer a major issue in current multiversion-based database systems, in particular for the read-only analytics work that we expect to run in the background. However, DBMS still use short-lived locks for shared data structures (e.g., *write-ahead log* (WAL)) and use coarse-grained table-based shared locks that can conflict with table restructuring tasks. Priority inversion is typically handled by avoiding the preemption of tasks that hold locks. We take this a step further: only when a time-sensitive task waits for a lock held by a background task, do we avoid (or reverse) preemption by temporarily increasing the priority of the background task. We cross the DBMS/OS boundary via eBPF maps (conceptually similar to [2, 3]) and refer to it as *application-based* scheduler hinting: A database task can signal our scheduler whether it is holding or requesting any locks.

PostgreSQL-based prototype and MariaDB proof-of-concept. PostgreSQL is a popular process-based open-source DBMS that currently relies on the OS to schedule concurrent tasks. We use PostgreSQL's extension system to allow users to conveniently declare priorities for workloads and implement the application-based scheduler hinting directly in PostgreSQL. A MariaDB proof-of-concept implementation confirms the applicability of UFS in a thread-based system and that changes to the DBMS code remain reasonable.

Contributions. This paper makes the following key contributions:

- **Detailed analysis of state-of-the-art OS schedulers.** We evaluate existing OS scheduling algorithms under mixed database workloads. We show how and why Linux scheduling options disadvantage time-sensitive CPU-bursty transactional workloads in the presence of CPU-bound DB analytics.
- **A selectively unfair scheduler.** UFS exploits Linux's eBPF subsystem with the `sched_ext` interface to support mixed workloads. It grants time-sensitive tasks immediate CPU access, schedules low-priority background work via low-overhead deferred dispatch, and preserves `cgroup`-proportional sharing.
- **Handling priority inversion via application hinting.** To address the priority inversion problem, we enable a line of communication between DBMS and OS scheduler via eBPF maps.
- **Minimally invasive prototype.** We present a prototype implementation for PostgreSQL. Application hinting leverages PostgreSQL's wait-event reporting path. Only minimal code modifications to PostgreSQL and no modifications to the Linux source code are required.
- **An SQL interface to assign process priority.** We provide a PostgreSQL extension that enables users to assign priorities to their PostgreSQL sessions directly in SQL.
- **Broader Feasibility.** Our exploration of MariaDB supports feasibility for other DBMSs.
- **Experimental evaluation.** For both PostgreSQL and MariaDB, UFS improves throughput of time-sensitive tasks within mixed workloads by up to 2× while reducing tail latency by half, compared to EEVDF. Even Linux real-time scheduling has tail latencies 1.5× higher than UFS. At the same time, UFS provides proper proportional sharing when CPU-bursty and CPU-bound workloads have equal priority. Performance remains as expected with various `cgroups`, different hardware configurations, and workloads. We show in a micro-experiment that application-hinting indeed succeeds at avoiding priority inversion.

2 BACKGROUND

In this section, we provide background on cgroups, Linux scheduling, eBPF, sched_ext, and PostgreSQL/MariaDB.

Control Groups. With the Linux cgroups feature, processes (but also threads or even individual tasks) can be organized into hierarchical groups for the management of resources (e.g., CPU time, memory, and I/O) [53]. Cgroups are often used in container runtimes. A cgroup can impose limits and accounting for each resource. The weight parameter `cpu.weight` represents the cgroup's relative share of CPU time: Under contention, a cgroup with CPU weight 200 is intended to receive roughly twice as much CPU time as a cgroup with weight 100. By placing high-priority workloads into cgroups with high weight, Linux allocates them more CPU time while still ensuring proportional fairness among all groups.

Scheduling in Linux. The Linux kernel provides different schedulers, each handling the tasks of a *scheduling class*. When the kernel searches for tasks to execute, it first considers runnable tasks in the scheduling class with the highest precedence, then proceeds to lower-precedence classes. Within these classes, a *scheduling policy* specifies the exact rules for scheduling a task. The precedence order of the classes discussed in this paper is (1) real-time scheduling

with the policies SCHED_FIFO and SCHED_RR, (2) fair scheduling with the policies SCHED_NORMAL and SCHED_IDLE, and (3) eBPF-based scheduling with the SCHED_EXT policy.

Real-time scheduling. Using SCHED_FIFO and SCHED_RR, tasks with real-time requirements receive static priorities from 1 to 99. With SCHED_FIFO, a runnable task is executed as long as it is runnable, unless it is preempted by a higher-priority real-time task. SCHED_RR uses round-robin with a run queue for each priority. The first runnable task from the highest-priority queue is executed only for a time quantum and then moved to the end of the queue. Runnable tasks with higher priority preempt tasks with lower priority.

Fair scheduling. Since Linux 6.6, *Earliest Eligible Virtual Deadline First* (EEVDF) has been the default fair scheduler [33]. EEVDF provides approximate proportional fairness [52] and supports cgroups. It assigns each runnable task that uses the SCHED_NORMAL policy a virtual deadline based on its weight (higher-weight tasks receive earlier deadlines and more CPU share) and runs the eligible task with the earliest virtual deadline. A task under the SCHED_IDLE policy runs only when CPU cycles are available.

eBPF. The *extended Berkeley Packet Filter* [44] is a Linux kernel subsystem that enables runtime loading and execution of programs within the kernel. The programs are *eBPF verified* before use and compiled just-in-time. eBPF programs are triggered by events such as function calls in user space applications or by specific kernel hook points. eBPF programs are stateless but can store data in *eBPF maps*, enabling them to maintain state between invocations. More complex data structures, like ring buffers and red-black trees, are built on top of these maps. Furthermore, these eBPF maps can be read and written from user space. eBPF programs are subject to verifier constraints, while map operations can introduce synchronization and contention overheads [2, 3]. These constraints must be considered when designing application-aware kernel functionality.

sched_ext. Custom scheduler policies can be implemented as dynamically loadable eBPF programs [32] via the sched_ext class. sched_ext provides hook points that are called when scheduling decisions are to be made, such as when a task becomes runnable, or a CPU needs work. The actual scheduling is performed by eBPF callback implementations registered on these hooks. Callbacks that are not implemented fall back to default behavior, allowing a scheduler to focus on the parts that are crucial for the application. When a sched_ext scheduler aborts, encounters an error, or is unloaded, the tasks under the SCHED_EXT policy fall back to the scheduler responsible for SCHED_NORMAL policy tasks.

A sched_ext implementation controls task execution via various queues. Local *dispatch queues* (DSQs) are per-CPU run queues that hold tasks intended to run on a specific CPU soon. They are typically small and fast, and the first place a CPU checks when selecting its next task. A task stays in a DSQ until the CPU selects it for execution, at which time it is removed. Once execution stops, the scheduler must place the task again into a DSQ before it can run again. In contrast, custom DSQs are policy-specific queues, such as queues for priority classes. Tasks in custom queues do not execute immediately; they remain queued until a CPU becomes available, and the scheduler dispatches them to a CPU-local DSQ.

PostgreSQL. PostgreSQL is one of the most popular open-source relational DBMS. It uses a process-based architecture; each client connection is handled in a separate process, called a *backend*. PostgreSQL uses *multi-version concurrency control* (MVCC) and read-only transactions do not set row-level locks. They do set shared locks on tables but only conflict with transactions that perform changes on the entire table (e.g., table delete or view refresh). Update transactions can set conflicting row-level locks, but we assume that such transactions never run as background tasks. PostgreSQL uses *lightweight locks* (LWLocks) to synchronize access to data structures in shared memory (e.g., transaction ID generation) and *spinlocks* for short-term locks held for just a few instructions. To avoid keeping processes in a busy-wait loop, processes perform short sleep cycles, releasing the CPU for other tasks.

MariaDB. MariaDB is another open-source DBMS. Unlike process-based PostgreSQL, MariaDB is multi-threaded and delegates much of its concurrency control to storage engines. Its default engine, InnoDB, combines MVCC with row-level locks. MariaDB also provides table and metadata locks and uses short-lived internal locks.

3 SHORTCOMINGS OF CURRENT LINUX SCHEDULERS

Interactive transactions are often CPU-bursty, requesting short amounts of CPU time followed by communication with the client. They should typically always run with high priority, as short response times are crucial. CPU-bound tasks might either be able to run in the background when they are not time-sensitive, or they may actually be part of an interactive analysis, in which case they do need reasonable response times. As such, a scheduler needs to ensure that background tasks use the CPU only when time-sensitive tasks do not need it. At the same time, tasks with the same priority should receive CPU time proportionally. The question is whether mechanisms that Linux provides for priority-based CPU scheduling suffice for mixed database workloads. Our analysis shows that this is not the case. However, failure modes differ between schedulers.

Setup. We analyzed the behavior of the schedulers by executing three workload mixes, as described in Table 1. Workload "SOLO" serves as baseline. We had 4 PostgreSQL-backend workers with exclusive access to 4 cores[1,2] serving 4 CPU-bursty TPC-C clients. In the two other workloads, the 4 TPC-C workers ran concurrently with 4 workers executing a TPC-H query in a tight loop within a UDF representing CPU-bound work. In the "MIN:MAX" workload mix, the CPU-bound tasks are considered background tasks. We therefore assign the CPU-bursty tasks maximum priority, and the CPU-bound tasks minimum priority. In the "50:50" mix, we assume that the CPU-bound TPC-H queries are nevertheless time-critical, so we assign them the same high priority as the CPU-bursty tasks. To achieve these low/high priority combinations, we set the scheduler configurations as outlined in Table 2, using EEVDF and SCHED_NORMAL with different weights for high- and low-priority tasks[3]; using SCHED_IDLE for low-priority tasks (only relevant for MIN:MAX); or using real-time scheduling SCHED_FIFO

[1]The detailed software and server configuration is presented in Section 6.
[2]We selected CPUs in strides of two (0, 2, 4, 6), to confine the experiment to a single NUMA (*non-uniform memory access*) node and thereby avoid interference from cache-preserving heuristics that prioritize NUMA locality over optimal placement.
[3]Note that 1 is the minimum and 10k the maximum weight possible for cgroups.

Table 1: Database workloads, stand-alone (SOLO) and mixed.

Workload	CPU-bursty	CPU-bound
SOLO	TPC-C	–
MIN:MAX	TPC-C, high-prio	TPC-H, low-prio
50:50	TPC-C, high-prio	TPC-H, high-prio

Table 2: Scheduling high- and low-priority tasks.

Scheduler	High-prio	Low-prio
EEVDF	NORMAL (weight = 10k)	NORMAL (weight=1)
IDLE	NORMAL (weight = 10k)	IDLE
FIFO	FIFO (priority = 99)	NORMAL (weight = 1)
RR	RR (priority = 99)	NORMAL (weight = 1)

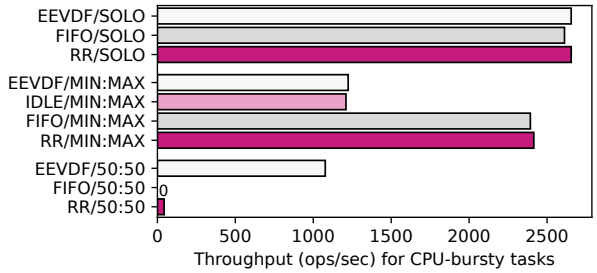


Figure 1: Throughput of CPU-bursty tasks, run stand-alone and as part of mixed workloads. Varying OS schedulers.

or `SCHED_RR`[4] for high-priority tasks, while using `SCHED_NORMAL` for low-priority tasks.

Results. Figure 1 shows the throughput for the CPU-bursty tasks with these scheduler configurations. SOLO is the baseline, as each CPU-bursty task gets an exclusive CPU. This is the best-case scenario, and all scheduling strategies perform equally. For MIN:MAX, we would expect the CPU-bursty throughput to be nearly the same as SOLO, because CPU-bound tasks have much lower priority. However, with EEVDF and IDLE, throughput suffers considerably. In contrast, FIFO and RR can still deliver very good throughput. For 50:50, we expect the CPU-bursty throughput to be reduced proportionally, as the CPU-bound tasks should receive an equal share. This can be observed with EEVDF. However, for FIFO and RR, the throughput collapses, even reaching zero in one case.

EEVDF fails to balance initial task placement. In EEVDF, when a task wakes up or is forked, it must be assigned a target CPU before it becomes runnable. EEVDF first chooses a base CPU, typically favoring the previous CPU or a nearby sibling CPU involved in the wakeup. When the base CPU is busy, the scheduler scans nearby sibling CPUs in a deterministic order and returns the first CPU that appears idle. Since the background tasks keep most CPUs busy most of the time, CPU-bursty tasks frequently fail the base-CPU check and fall into the idle-sibling path. CPUs that already host CPU-bursty work are more likely to become briefly idle again, so they are selected repeatedly. As a result, wakeups tend to stack CPU-bursty tasks onto the same few CPUs instead of spreading them across available CPUs, leading to poor performance for these CPU-bursty tasks.

[4]Note that 99 is the maximum priority for a task in `SCHED_FIFO` or `SCHED_RR`.

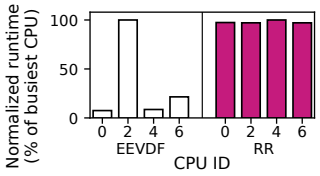


Figure 2: CPU utilization by high-prio workers across 4 CPUs.

Figure 2 illustrates this behavior. We reconstruct the per-CPU execution time for CPU-bursty tasks from kernel scheduler traces (`sched_switch` events). For each CPU, whenever a context switch occurred, we computed the elapsed time since the previous switch on that CPU and attributed that interval to the process that had been running. We then sum the total runtime of the processes handling CPU-bursty tasks. In Figure 2, the y-axis shows the CPU utilization of these CPU-bursty tasks, normalized to the CPU with the highest load due to CPU-bursty tasks (100%). The differences in bar height indicate the differences in CPU-bursty load. In fact, EEVDF frequently concentrates multiple CPU-bursty tasks on the same CPUs, creating short-term pile-ups. The other CPUs continue to be used for background tasks. EEVDF eventually mitigates some pile-ups via periodic load balancing. However, because CPU-bursty tasks are short-running, the skew and imbalance often persist for a large fraction of the request lifetime. By the time load-balancing kicks in, throughput has already been impacted. In contrast, RR distributes time-sensitive work evenly across available CPUs because it can preempt lower-priority work immediately, instead of pushing CPU-bursty work toward CPUs that briefly appear idle.

These results show that we need a scheduling strategy with smart initial placement and aggressive preemption.

Real-time scheduling cannot handle same-priority mixed workloads. While real-time scheduling can handle workloads with different priorities well, it fails when CPU-bursty and CPU-bound tasks have the same priority. Under FIFO, a task runs until it blocks, yields, or is preempted by a higher-priority task. Once a CPU-bound task gets the CPU, the CPU-bursty tasks with the same priority are completely delayed. With RR, runnable tasks of the same priority execute in fixed CPU quanta, that is, short bounded turns on the CPU. CPU-bursty interactive tasks frequently block briefly on I/O, locks, or client interaction before consuming their full turn. When that happens, they lose the unused remainder, whereas a CPU-bound worker of the same priority can keep consuming full turns repeatedly. The major problem here is that FIFO and RR lack virtual runtime accounting. In contrast, EEVDF calculates virtual runtimes, allowing it to prioritize tasks that have received less CPU time so far, including bursty tasks that repeatedly block for I/O.

This shows that our scheduler must support virtual runtime.

4 SOLUTION DESIGN

We introduce the conceptual solution design of our *Unfair Scheduler* (**UFS**) here and discuss its implementation in Section 5. Our solution aims at DB-specific scheduling that can be implemented without significantly changing the DBMS or the OS. We want to give time-sensitive tasks absolute priority and run background tasks only

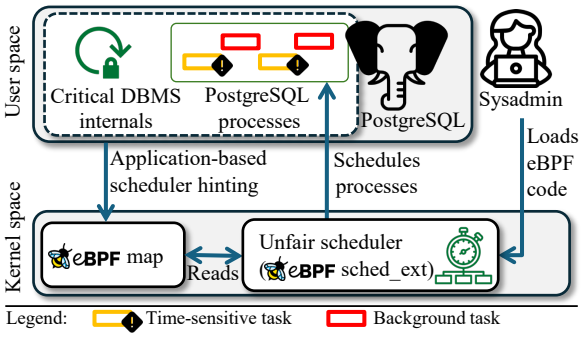
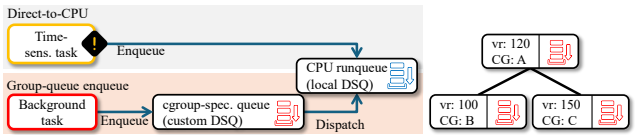


Figure 3: UFS framework architecture.



(a) Enqueue strategies **(b) Runnable tree.**

Figure 4: (a) Direct-to-CPU queues for time-sensitive vs. group-queue enqueue for background tasks; (b) A runnable tree with nodes for cgroups A, B, and C.

when the CPU is available. Yet multiple workloads with the same priority should be scheduled proportionally. Furthermore, UFS must handle the possibility of priority inversion. UFS focuses on CPU scheduling. Other resources, such as I/O scheduling, and priority inversion that might arise from them, are not handled by UFS.

Architecture. Figure 3 shows the architecture of our solution. The mixed-workload application runs in user space. Here, this is PostgreSQL executing time-sensitive and background tasks (yellow/red boxes in the figure). UFS is a novel scheduler policy implementation built on the `sched_ext` class; hence, it does not require invasive in-kernel changes. To install UFS, the system administrator simply loads the eBPF code of UFS at run-time into the Linux kernel space. The DBMS communicates scheduling-critical internals (such as locks held, see Section 2) to the UFS scheduler by writing to an eBPF map allocated in kernel space.

Scheduling Tiers, Task Prioritization, and cgroups. We assign each PostgreSQL backend process, and with it the tasks/workload it executes, to one of two scheduling tiers: (1) *time-sensitive* and (2) *background*. Assignment can change during runtime. In order to enable workloads with different priorities within a tier, UFS supports Linux cgroups. The weights of the cgroups within a tier define the share of resources their workloads will receive. As with weight-based fairness in EEVDF, tasks under UFS inherit the effective share of the cgroup they belong to.[5] As with the conventional cgroup interface, weights and related parameters can be dynamically adjusted. UFS respects the cgroup hierarchy so that each cgroup's parameters are defined relative to its parent, with changes propagating accordingly.

Task Scheduling. Tasks are scheduled according to their tier w.r.t. (1) *priority under contention*: which task is allowed to run when CPU time is scarce, and (2) *CPU selection and dispatch*: how a task is assigned to a CPU and how it reaches that CPU's run queue.

(1) Priority under contention. Time-sensitive tasks always take precedence over background tasks, regardless of cgroup weights. UFS schedules background work only when this does not delay time-sensitive execution.

(2) CPU selection and dispatch. For time-sensitive tasks, UFS applies the *direct-to-CPU strategy*: As shown in Figure 4a, when a time-sensitive task becomes runnable, the scheduler selects an appropriate target CPU based on the current system conditions and directly enqueues the task into that CPU's run queue. The task can run immediately when the target CPU is idle or running background work, in which case UFS issues an idle or preemption kick. An important aspect of this *direct-to-CPU* strategy is that CPU selection ensures a balanced task placement from the start, rather than relying on later corrective load balancing.

For background tasks, UFS implements the *group-queue enqueue strategy*: When a background task becomes runnable, it is enqueued into a cgroup-specific queue. Only when a CPU becomes idle and requests work does the scheduler select a task from an eligible background cgroup and dispatch the task to that CPU's run queue. This *deferred* placement has two benefits: (1) It avoids expensive global CPU searches upon wake-up. (2) It allows for reactive load distribution by having CPUs pull background work on demand. The scheduler can still respect cgroup proportionality and affinity constraints when deciding which background tasks to dispatch.

Application-based Scheduler Hinting. UFS must prevent priority inversion. In PostgreSQL, sustained priority inversion requires three concurrent tasks. If a background task T_1 holds a lock that a time-sensitive task T_2 wants, and T_1 and T_2 are the only two tasks in the system, then T_1 will be able to run until it releases the lock and will then be preempted by T_2. No priority inversion occurs. However, if there are other time-sensitive tasks that are runnable, they will preempt T_1, and T_1 will not be able to release the lock, delaying T_2 with it. T_2 then involuntarily receives the same low priority as T_1.

Note that for spinlocks, PostgreSQL prevents a task T_2 from spinning indefinitely when it tries to acquire a spinlock held by another task T_1. Instead, the task will spin only briefly before sleeping for a substantially longer interval. This behavior effectively prevents the more direct form of priority inversion in which a time-sensitive task starves the background task that must run to release the lock it is waiting for. Nevertheless, the indirect inversion described above remains possible, where T_1 cannot release the lock because yet another high-priority task occupies the CPU.

To address this, we enable application-based scheduler hinting via an eBPF map: user space applications can write hints into this map, which eBPF programs in kernel space can access. This enables communication between user space programs and UFS. When a user space application inserts a hint that a time-sensitive task is waiting for a lock, the scheduler checks whether a conflicting lock is currently held by a background task and temporarily treats that background task as runnable in the time-sensitive tier until the lock is released. If a background task that was upgraded to time-sensitive because of such a conflict blocks on a lock held by another background task, this task will also be upgraded to time-sensitive.

[5] UFS also honors the standard CPU-controller constraints exposed through cgroups. This includes usage limits (`cpu.max`) and CPU affinity restrictions (`cpuset.cpus`), although we do not anticipate that they are relevant for mixed database workloads.

Consequently, upgrading tasks propagates transitively through chains of blocked tasks. Note that background tasks holding locks can be preempted as long as no time-sensitive task wants the locks.

5 IMPLEMENTATION

This section describes our `sched_ext` implementation of UFS, followed by the integration tasks for PostgreSQL and MariaDB.

5.1 Unfair Scheduler

UFS is built on the `sched_ext` framework using callbacks and DSQs (see Section 2), and uses virtual-runtime-based scheduling.

5.1.1 Virtual runtime. UFS allocates CPU time in *time slices*, i.e., fixed-size bounded execution intervals during which a task or cgroup is allowed to run before scheduling may be reconsidered. Virtual-runtime scheduling in UFS tracks how much CPU time a task has received across such slices. Tasks that have received less service are ordered earlier to run. UFS tracks virtual runtime on two levels: (1) *task virtual runtime* tracks task runtime *within* a cgroup, and (2) *cgroup virtual runtime* tracks each cgroup's runtime, scaled according to its cgroup weight (*weight-scaled virtual runtime*). UFS chooses the cgroup with the *lowest* cgroup virtual runtime as the next one to receive CPU service. Within that cgroup, it selects the task with the *lowest* task virtual runtime to run next.

Example. Assume cgroup A with weight 2 and cgroup B with weight 1. If each receives a 10 ms time slice, A's virtual runtime increases by 5, and B's increases by 10. Thus, A is selected more often and receives roughly twice as much CPU time as B. If UFS selects A for execution and A contains two tasks A_1 and A_2, with task virtual runtimes 20 and 10, resp., then UFS chooses A_2.

Note that this implementation differs from EEVDF, as EEVDF uses virtual deadlines rather than virtual runtime ordering[6].

5.1.2 Enqueue Callback. The enqueue callback is triggered whenever a task becomes runnable (e.g., upon wake-up) or must be re-enqueued (after preemption, yield, or slice expiration). Our callback implementation involves: (1) looking up the task's state and cgroup, (2) clamping the task's virtual runtime to prevent credit hoarding, and (3) enqueuing the task.

State lookup: UFS first retrieves the task's virtual runtime and identifies its current cgroup, as provided by the kernel. It then loads the corresponding per-cgroup scheduling state, including metadata such as the cgroup's weight, tier, and virtual runtime.

Clamping virtual runtime: Each task maintains a virtual runtime, reflecting its accumulated runtime, and used to order tasks within its cgroup. Before inserting the task into a DSQ, the scheduler limits how far behind the cgroup's current virtual runtime a task may lag. Specifically, if the task's virtual runtime is older than "one task slice" relative to the cgroup's current virtual runtime, it is raised to that bound. This prevents a task that has been idle for a long time from accumulating scheduling credit and immediately jumping ahead of the cgroup's recently active tasks. Virtual runtimes of time-sensitive tasks are kept lower than those of background tasks, so that they are always prioritized on a given local DSQ.

[6]Implementing virtual deadlines would require us to determine whether a task is owed CPU based on its lag, which would make the scheduler significantly more complicated – something we want to avoid within an eBPF program.

Enqueuing: UFS implements two enqueuing strategies depending on the task tier: (1) *direct-to-CPU* for time-sensitive tasks and (2) *group-queue enqueue* for background tasks (see Figure 4a).
(1) Direct-to-CPU enqueue: The task is inserted into a CPU-local DSQ, and immediate rescheduling is performed if needed. In detail, UFS determines a CPU that can run the task promptly and enqueues the task into the CPU's DSQ. If there are already other time-sensitive tasks in the queue, its virtual runtime is used to determine the queue position. Otherwise, UFS issues a CPU "kick": It wakes the CPU if it is idle or requests preemption if the CPU is running background work. The CPU-selection function first searches the task's previous NUMA node and then falls back to all allowed CPUs. It prefers an idle CPU, then the least-loaded CPU with background tasks, and finally the least-loaded CPU with time-sensitive tasks.
(2) Group-queue enqueue: The task is inserted into its cgroup's custom DSQ using the virtual runtime as key. Since wake-up latency is not a concern, direct CPU placement is intentionally avoided. The dispatch callback is responsible for eventually moving these tasks into a local DSQ and is discussed next.

5.1.3 Dispatch Callback. The dispatch callback is invoked whenever a CPU needs work and its DSQ is empty. This means that no time-sensitive tasks need the CPU at the moment. Therefore, the dispatch callback must check whether a runnable background task requires CPU and move this task to the CPU-local DSQ.

Runnable tree. UFS maintains cgroup semantics, in particular preserving scheduling proportional to weight. To achieve this, it uses a data structure that we call the *runnable tree*. It manages the runnable background cgroup DSQs. Since eBPF is used, the scheduler must use bounded, eBPF verifier-friendly data structures such as maps and DSQs. The existing eBPF red-black tree meets these requirements because its operations expose a constant number of instructions to the eBPF verifier, independently of the number of stored elements. We therefore use it to implement the runnable tree as a binary search tree, as in Figure 4b: Each node in this tree represents a cgroup DSQ, using its virtual runtime as the key. The leftmost node of the tree indicates the cgroup with the lowest virtual runtime.

When UFS tries to find the next background task to run, it selects the next eligible cgroup DSQ from the runnable tree and transfers the task that has run the least so far into the CPU-local DSQ:

Selecting the next cgroup DSQ. UFS repeatedly tries (up to a small bounded number of iterations) to obtain work as follows: (a) Peek the cgroup with the minimum virtual runtime from the runnable tree. (b) Verify active state. If the cgroup disappeared (e.g., tasks exited), dispatch removes the node from the runnable tree and retries. (c) Try to obtain runnable tasks. Dispatch attempts to move the runnable task with the smallest virtual runtime from the selected cgroup's queue to the CPU's local DSQ. There are two outcomes:
Task found: the CPU commits to this cgroup. Its virtual runtime is then advanced by one time slice, scaled inversely by the cgroup's weight, and the cgroup is reinserted into the runnable tree, with the updated virtual runtime as key. This allows sharing proportional to weight across background cgroups without scanning.
Empty: if the cgroup has no remaining enqueued tasks, dispatch removes the cgroup from the runnable tree and places the corresp. bookkeeping node into a per-cgroup stash so it can be reused on the next enqueue. This avoids empty cgroups in the runnable tree. If

no cgroup yields a task within the bound, dispatch returns without work, and the CPU idles until the next dispatch invocation.

Selecting the next task within a cgroup DSQ. Within each cgroup DSQ, tasks are ordered by their virtual runtime. The task with the lowest virtual runtime has executed the least amount of work so far and is positioned at the front of the queue. When UFS selects that cgroup to run, the task at the head of its custom DSQ is moved to a CPU's local DSQ and thus runs first.

Weight-scaled virtual runtime enables weight-proportional scheduling. Cgroup virtual runtimes advance inversely with their assigned cgroup weights, so higher-weight cgroups are selected proportionally more often.

With group-queue enqueue, dispatch implements a slice-based policy proportional to cgroup weight using the runnable tree data structure. Dispatch selects the earliest eligible cgroup, removes truly empty cgroups, and charges cgroups by advancing their virtual runtime before reinserting them for future selection.

5.2 Integration into PostgreSQL

5.2.1 Application-Based Scheduler Hinting. UFS relies on user space hints, stored in an eBPF map, to prevent priority inversion. We write the PostgreSQL lock events into an eBPF user ring buffer. This user ring buffer is consumed by UFS, which constructs an eBPF map with $(waiter\ pid) \rightarrow (holder\ pid)$ entries. When a time-sensitive task blocks on a lock held by a background task, the lock holder is upgraded to time-sensitive. It is again downgraded to background when PostgreSQL reports that the waiter has acquired the lock.

PostgreSQL provides generic infrastructure for reporting *wait events*. These occur in situations where a blocking operation is executed, such as waiting for I/O, reading data from a network client, or attempting to acquire a lock. To update the application-based hints, we add our own calls along PostgreSQL's wait-event reporting path so that lock activity is reported when a lock is attempted, acquired, or released. These calls are inserted at the same points where PostgreSQL invokes the methods `pgstat_report_wait_start()` and `pgstat_report_wait_end()`. Our PostgresSQL patch for lock-hinting requires fewer than 200 lines of code.[7] On the side of UFS, we need fewer than 100 lines to process these hints.

A crashed PostgreSQL backend process can leave stale entries in the eBPF map. PostgreSQL detects such crashes and automatically restarts all backend processes if the crashing process was in a phase where it had connected to shared memory and could hold locks. Independently, UFS uses a `sched_ext` task-termination callback to remove every entry in which the terminated process appears as a waiter or a holder.

5.2.2 PostgreSQL Management Extension. UFS configuration is done via PostgreSQL extensions and SQL configuration variables. A cgroup belongs either to the background or time-sensitive tier. The tier of a task is determined by the cgroup it belongs to.

By default, PostgreSQL backends are in the background scheduling tier in a default cgroup. To change the cgroup and/or tier from within PostgreSQL, we provide a PostgreSQL extension that exposes two configuration parameters: (1) one to specify the workload tier (time-sensitive or background) and (2) one to specify the weight of the associated cgroup. Should no cgroup for that tier exist with the given weight, such a cgroup is created automatically.

Since regular PostgreSQL *Grand Unified Configuration* (GUC) parameters are exposed by our extension, they can be manipulated directly via SQL. A client can assign a tier (`SET task_tier = ...;`) and weight (`SET task_weight = ...;`) to change these values for the current session. By using `SET LOCAL`, these values can be set at transaction scope. Since these values are managed through PostgreSQL's existing configuration mechanisms, role-level defaults can be set; each time this role connects to PostgreSQL, these predefined values are automatically loaded.

5.3 MariaDB

To analyze UFS's implementation effort and applicability beyond PostgreSQL's process-based architecture, we integrated it with MariaDB, a multi-threaded DBMS.

MariaDB can be extended using plugins. As in PostgreSQL extensions, custom configuration parameters can be introduced. We have reimplemented the PostgreSQL management extension from Section 5.2.2 for MariaDB version 10.6.23, which allows setting the priority of the current connection via configuration parameters. The MariaDB plugin consists of only 200 lines of code and shares much of the same code with the PostgreSQL extension (same general control flow and data structures). A main difference is that we need to register thread IDs in the cgroups instead of process IDs.

We also analyzed the implementation effort for application-based hinting, which requires updating the eBPF map that tracks lock status. There are different implementations of locks for metadata, tables, and InnoDB. We have identified about 10 code locations that need to update the eBPF map (e.g., `thr_lock()` and `thr_unlock()` for table-level locks). As such, compared to PostgreSQL, code changes might be more scattered.

6 EXPERIMENTAL EVALUATION

We evaluate UFS and compare it with Linux's default scheduling under a variety of settings and workloads. We also analyze application-hinting in micro-experiments and evaluate its overhead.

Implementation. We implemented UFS in C on Arch Linux, targeting Linux kernel 6.17. The loader was compiled with GCC 15.2.1, while the scheduler eBPF program was compiled with Clang 22.1.1. Our database extension and patch are integrated into PostgreSQL 17.0. Our implementation totals about 3,450 lines of C: 250 for the loader, 2,750 for the eBPF program, 250 for the SQL management extension, and fewer than 200 for the PostgreSQL lock-hinting patch.

Execution environment. We use servers within the same network: (1) PostgreSQL with UFS runs on a dual-socket server with two 24-core Intel Xeon Gold 6248R processors and 400GB of DRAM. Each processor has a base frequency of 3.0GHz and 36MB of cache. (2) The database clients run on a separate dual-socket server with two 24-core Intel Xeon Gold 6242R processors and 200GB of DRAM. Each processor has a base frequency of 3.1GHz and 36MB of cache.

Workloads. In most of our tests, we use the same workloads that were already described in Section 3. As *CPU-bursty work*, we run BenchBase [9] (commit `46fc66f`) on the client machine and execute

[7] We directly changed PostgreSQL because wait events are not propagated via hooks to extension code. An option could be to add a new hook for this purpose, and then handle the actual hinting entirely in extension code. This would make the changes in PostgreSQL less invasive: adding a new hook definition would be ≈20 lines of code.

TPC-C. We deviate from the default BenchBase configuration and choose scale factor 24. We vary the number of clients. Each client is served by a different PostgreSQL backend (worker). As *CPU-bound work*, we start UDFs in PostgreSQL at the beginning of each benchmark run. Each UDF issues TPC-H Query 17 in a continuous loop. We also include a test with `schbench` [36] (commit `6300b8f`), a standard benchmark for Linux schedulers, a test with an ML task, and a micro-application for testing application-hinting. Details of these applications are provided in the respective subsections.

We further conducted experiments with the YCSB benchmark, yet we do not include them in the paper: While they confirmed our observations, they revealed no additional insights.

Schedulers. We use the same scheduler configurations, EEVDF, IDLE, FIFO, and RR, with corresponding weights/priorities as outlined in Table 2 to handle low-priority background tasks and high-priority time-sensitive tasks. Using our UFS scheduler, low-priority tasks are in a background tier cgroup with weight 1, and high-priority tasks are in the time-sensitive tier with weight 10k.

Configuration. Simultaneous multithreading (SMT) is enabled on CPUs. To reduce run-to-run variability, we disable turbo boost and restrict the CPUs to the C1 C-state (avoiding transition overheads to deeper C-states). Using cores from both NUMA sockets, we capture any cross-node migration overhead caused by poor placement.

We use a fixed set of CPUs (mostly eight) from the server for our workload, assigned either using cgroup functionality or the `taskset` feature of FIFO/RR. UFS does not operate in partial mode, so it schedules all tasks of the OS, but we do ensure that our workload has exclusive access to the designated CPUs.

A 1-minute warm-up phase is followed by 1 minute of measurement. Throughput and latency measurements for CPU-bursty tasks are collected on the client side, directly from BenchBase; the remaining tasks are measured at the server side.

6.1 Mixed Workload Throughput

We explore whether UFS can handle the workload varieties for which the original schedulers fail. For that, we repeat the experiment in Section 3 including UFS. Our focus is first on throughput.

Setup. We evaluate the workload mixes from Table 1, using 8 cores: SOLO with only 8 CPU-bursty workers, MIN:MAX with 8 time-sensitive CPU-bursty and 8 background CPU-bound workers, and 50:50 where both the 8 CPU-bursty and the 8 CPU-bound workers have the same (high) priority. We include a second SOLO run with only 8 CPU-bound workers to observe them in isolation.

Results. Figure 5 shows the achieved throughput for background workload (to the left) and for time-sensitive workload (to the right). Running these workloads stand-alone ("solo") serves as baseline against which to compare mixed workloads. In this case, very similar throughput is achieved by all schedulers, both for CPU-bound and for CPU-bursty tasks. This is not surprising, as each PostgreSQL backend can get its own CPU, so scheduling is quite trivial.

For EEVDF, IDLE, FIFO, and RR, we observe the same behavior as in Section 3. In MIN:MAX, EEVDF and IDLE fail to give the CPU-bursty workloads the requested high priority, reducing their throughput by 50% compared to running SOLO. Instead, they allow background CPU-bound tasks to reach unexpectedly high throughput. In contrast, FIFO and RR, as well as UFS, are able to keep the

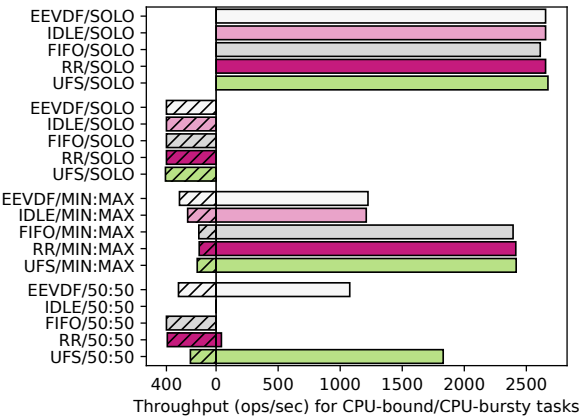


Figure 5: Throughput for CPU-bound vs. CPU-bursty tasks (shown left and right resp.), running solo or mixed.

throughput of the CPU-bursty tasks high at the expense of the background tasks, offering 2× the throughput of EEVDF and IDLE.

In 50:50, EEVDF provides the CPU-bursty tasks with reasonable CPU resources, although the share does not appear quite 50:50. The throughput of CPU-bursty tasks is only 40% of what it is in SOLO, while it is more than 75% for CPU-bound tasks. With FIFO and RR, the throughput of the CPU-bursty tasks collapses, yet it remains close to 100% for the CPU-bound tasks compared to running SOLO. In contrast, UFS provides reasonable performance for both task types: It maintains 70% of the CPU-bursty throughput and 50% of the CPU-bound throughput compared to running SOLO.

Discussion. The main reasons for the bad performance of EEVDF and IDLE for workload mix MIN:MAX, and FIFO and RR for 50:50, were presented in Section 3. UFS explicitly avoids the pitfalls.

In the MIN:MAX scenario, in contrast to EEVDF and IDLE, UFS does account for weights and thus avoids CPU-bursty tasks piling up on the same CPU. A closer analysis confirms that CPU utilization from CPU-bursty tasks is the same on all CPUs because it preempts background tasks when time-sensitive tasks arrive.

For the 50:50 scenario, the bad behavior of FIFO and RR for CPU-bursty tasks is discussed in Section 3. CPU-bursty tasks give up their slice of CPU whenever they block and are then starved by CPU-bound tasks. In contrast, UFS's virtual runtime accounting gives CPU-bursty tasks a fair share of the CPU, even alongside CPU-bound tasks. It is even "fairer" than EEVDF: both task types keep at least 50% of their SOLO throughput, while EEVDF's initial placement problem still causes some imbalance that UFS avoids.

In summary, UFS's aggressive placement for tasks in the time-sensitive tier and its virtual runtime management combine the respective strengths of Linux's real-time and fair scheduling classes, allowing it to handle challenging database workloads.

6.2 Mixed Workload Latency

So far, we have only looked at throughput. But for time-sensitive workloads, it is important to control end-to-end latencies, in particular for CPU-bursty workloads that consist mainly of short transactions. In particular, the long tail of latencies should not contain

Table 3: Latencies for CPU-bursty tasks achieved by different schedulers. Minimum values are highlighted in bold.

Scenario	Scheduler	Mean lat. (ms)	95th percentile lat. (ms)
SOLO	EEVDF	3.06	5.80
	RR	3.09	5.92
	UFS	**3.03**	**5.79**
MIN:MAX	EEVDF	6.63	14.23
	RR	**3.36**	9.66
	UFS	**3.36**	**6.37**
50:50	EEVDF	7.53	17.19
	RR	183.24	1397.97
	UFS	**4.43**	**11.42**

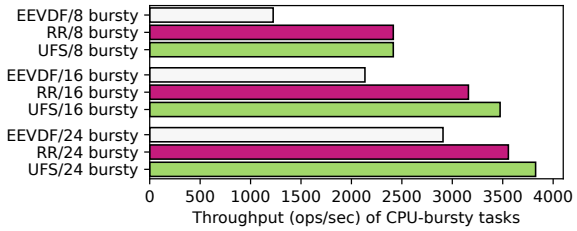


Figure 6: Scaling throughput for CPU-bursty tasks for 8/16/24 CPU-bursty versus 8 CPU-bound tasks.

extreme outliers. We therefore examine the latencies of the time-sensitive workloads within a mixed workload. Our expectation is that in mixed workloads, UFS produces overall lower latencies for time-sensitive workloads than EEVDF and RR.

Setup. The experimental setup is the same as in Section 6.1.

Results. Table 3 reports mean and 95th percentile latency summaries for the CPU-bursty tasks for the SOLO, MIN:MAX, and 50:50 scenarios. We omit results for the IDLE and FIFO scheduling options as they were similar or worse than EEVDF resp. RR.

While response times are very similar for the SOLO scenario, for MIN:MAX, EEVDF's mean latency is 2× higher than UFS, and its 95th-percentile latency is 2.2× higher. RR's performance is much steadier than EEVDF, with a mean latency matching UFS. However, its 95th percentile latency remains 1.5× higher than UFS. In fact, UFS latencies are only minimally higher than when the CPU-bursty tasks run in isolation. For the 50:50 workload, EEVDF's mean latency is 1.7× and the 95th percentile 1.5× higher than those of UFS. RR's latencies have completely deteriorated.

Discussion. UFS's advantage extends beyond mean latency: under mixed-priority workloads (MIN:MAX), its aggressive preemption not only largely outperforms EEVDF's response time, but also results in much lower tail latencies, even compared to RR. It can achieve latencies very close to CPU-bursty tasks run in isolation.

For 50:50, UFS achieves a fair and proportional assignment of CPU time to CPU-bursty tasks, despite their bursty behavior. When they are ready to execute, UFS gives them their fair share according to the virtual runtime ordering, providing fairer results than using deadlines as in EEVDF, leading to overall lower latencies.

6.3 Throughput Scaling Under Oversubscription

In real database deployments, the number of time-sensitive CPU-bursty tasks may exceed the number of available CPUs. We explore how the throughput of time-sensitive tasks scales under oversubscription in a mixed workload setting.

Table 4: Varying priorities for CPU-bound and CPU-bursty tasks. UFS throughput reflects the configured priorities.

	Assigned weight		EEVDF ops/sec		UFS ops/sec	
CPU usage	bound	bursty	bound	bursty	bound	bursty
Lower weight	2	6.67k	84	865	23	911
Higher weight	3	10k	120	868	32	1,531
Ratio (ca.)	**0.7**	**0.7**	0.7	1.0	**0.7**	**0.6**

Setup. Our baseline is the MIN:MAX mix from Section 6.1, which prioritizes time-sensitive maximally. We vary the setup: The background workload remains at 8 workers, the number of time-sensitive CPU-bursty workers increases from 8, to 16, and then to 24.

Results. Figure 6 compares how throughput for the time-sensitive workload scales under varying degrees of oversubscription, for different schedulers. As expected, overall throughput increases with increasing number of tasks but not linearly, because time-sensitive tasks start to compete among themselves for CPU resources.

Throughput with EEVDF remains well below that of RR and UFS, as observed in Section 6.1, but its disadvantage is not as pronounced when there are more time-sensitive tasks. With 8 time-sensitive tasks, it achieves only 50% of the throughput of RR and UFS, compared to 61% with 16 tasks and 76% with 24 tasks relative to UFS.

With increasing workload, UFS starts to outperform RR by up to 10% when there are 24 time-sensitive workers. Comparing RR and UFS, UFS performs better. For the baseline, both UFS and RR achieve a throughput 2× higher than EEVDF. However, as the number of time-sensitive workers increases, the throughput of UFS exceeds the throughput of RR by almost 10%.

Discussion. The cause of EEVDF's worse performance is its initial placement policy. However, the more time-sensitive tasks are in the system, the more likely background tasks are already preempted when a time-sensitive task is scheduled, leading to better placement options and improving EEVDF's relative performance. Still, UFS's aggressive preemption gives it an edge over the others.

6.4 Mixed Workloads with Varying Priorities

We evaluate how well UFS performs under weight-proportional scheduling, that is, we analyze the performance when it has to support tasks of different weights in each of the tiers.

Setup. We run 16 CPU-bound background workers and 16 CPU-bursty time-sensitive workers concurrently on the same set of 8 CPUs. We configure the priorities by assigning weights as shown in Table 4: Background tasks (CPU-bound) are equally split into two cgroups with weights 2 and 3. Time-sensitive tasks (CPU-bursty) are split into two cgroups with weights 6.67k and 10k. So in both tiers, the ratio between lower- and higher-weight workload is 2/3.

Results. Table 4 further shows the throughput achieved for each of the four workloads under EEVDF and UFS. As seen before, UFS is much better at providing time-sensitive tasks higher throughput than EEVDF (at the cost of background tasks).

What is interesting is the relative throughput each workload achieves. Both UFS and EEVDF preserve proportional CPU allocation among the two background workloads. With both schedulers,

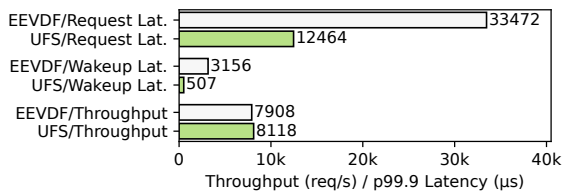


Figure 7: Benchmarking schedulers under general workloads, indicating UFS competes well with the standard scheduler.

the workload with weight 2 achieves roughly two-thirds of the throughput of the workload with weight 3.

For time-sensitive workloads, however, the schedulers differ. UFS preserves the expected ratio: time-sensitive workload with weight 6.67k achieves ca. two-thirds of the throughput of time-sensitive workload with weight 10k. EEVDF, in contrast, gives the two time-sensitive workloads similar throughput, despite different weights.

Discussion. UFS preserves weight-proportional scheduling not only among background workloads, but also among time-sensitive workloads. This is important as time-sensitive tasks should preempt background work, but they should be treated according to their weights when they compete with each other.

EEVDF fails for the same reason as with MIN:MAX in Figure 5: its wakeup-time placement does not account for weight. Thus, time-sensitive tasks with different weights are placed in a way that gives them similar CPU time. Later load rebalancing reacts too slowly to correct this for these CPU-bursty tasks.

6.5 Comparison Under General Workloads

UFS has been designed primarily for mixed database workloads. We also explore how it performs under the general workloads under which standard OS schedulers are assessed.

We evaluate UFS using `schbench` [36], a standard benchmark for Linux schedulers.[8] UFS treats all tasks as "background" and schedules them with default cgroup weight 100.

Results. Figure 7 compares the standard scheduler EEVDF with UFS regarding throughput and extreme latency values (reporting 99.9th percentile). UFS achieves comparable throughput and features better wakeup and request latencies: UFS reduces wakeup p99.9 latency by 6.2×, and request p99.9 latency by 2.7×.

Discussion. These results indicate that UFS is also promising for non-DB workloads. We attribute the lower extreme latencies to the UFS mechanism of pulling background work whenever a CPU is free. This motivates a more extensive evaluation in future work to assess robustness in practice.

6.6 Lock-Induced Priority-Inversion

As mentioned in Section 4, contention on locks (spinlocks, lightweight locks or table-level locks) can induce priority inversion. If a background task holds a lock required by a time-sensitive task, another time-sensitive task may starve the background lock holder of CPU time, delaying lock release. This creates an indirect form of priority inversion in which one time-sensitive task effectively blocks another through the delayed execution of a background task. Our next experiment evaluates how effectively UFS can mitigate this effect through our scheduler hinting mechanism.

[8]Performing a 10 sec calibration run and measuring a 90-second regular run. We use the benchmark's default/example configuration of a 256 KB cache footprint (`-F 256`) and 5 operations per compute phase (`-n 5`), representing moderate settings that preserve some cache pressure and frequent sleep/wakeup behavior.

Table 5: Completion times for the materialized-view refresh, TPC-H Q17 workload, and view query, reported in seconds.

Sched.	Refresh	Q17 workload	View query	Overall
Baseline	**7.21**	–	**6.24**	**7.21**
EEVDF	2284.71	2284.71	2283.21	2284.71
FIFO	167.35	167.35	166.56	167.35
RR	156.91	156.91	155.80	156.91
UFS	**17.68**	**17.68**	**16.53**	**17.68**

Under our standard mixed TPC-C and TPC-H workload, we were unable to reproduce priority inversion for a sustained period. Although situations did arise in which a background task held a lock needed by a time-sensitive task, the lock was never held for long, since the amount of work performed by the background task while holding it was minimal. Therefore, we created a micro-experiment to illustrate how UFS avoids priority inversion.

Setup. We use a PostgreSQL materialized view that summarizes customer orders from the TPC-C database. First, we launch a background *refresh* task, which executes a non-concurrent `REFRESH MATERIALIZED VIEW`, acquiring an `ACCESS EXCLUSIVE` lock on the materialized view. Shortly after starting the view refresh, we launch a time-sensitive *view query*, which attempts to read from the materialized view. The view query automatically requests an `ACCESS SHARE` lock on the view table, and thus must wait until the refresh releases its lock. Once the query is blocked, we launch four time-sensitive TPC-H Q17 workers that continuously perform CPU-bound work. All tasks are restricted to the same four CPUs. The Q17 workers starve the background refresh of CPU time, thus indirectly preventing the view query from making progress. We compare EEVDF, FIFO, RR, and UFS, configured as before. The view query and the Q17 workers are assigned the highest possible priority, while the refresh is assigned the lowest possible priority. A baseline runs the refresh and the view query without concurrent Q17 workers. Here, the refresh will execute until it releases the lock. This baseline captures the time it takes for the refresh to complete and release the lock without interference from concurrent tasks.

Results. Table 5 reports how long the refresh, Q17 workload, and high-priority query take to complete. The last column indicates the overall time until all requests have completed.

The baseline overall takes 7.21 seconds, which is basically the time of the refresh. The query starts around 1 second after the refresh, thus it waits 6.24 seconds, and once it gets the lock, it completes nearly instantaneously. The existing Linux scheduling policies, in contrast, show severe priority inversion. Under EEVDF, the four Q17 workers occupy the available CPUs, while the low-priority refresh receives very little CPU time. As a result, the refresh and the query blocked behind it both take approximately 38 minutes to complete. FIFO and RR complete the experiment in approximately 167 and 156 seconds respectively, with the high-priority query remaining blocked for nearly the entire duration. This is substantially faster than EEVDF, and due to *Fair Server*, a

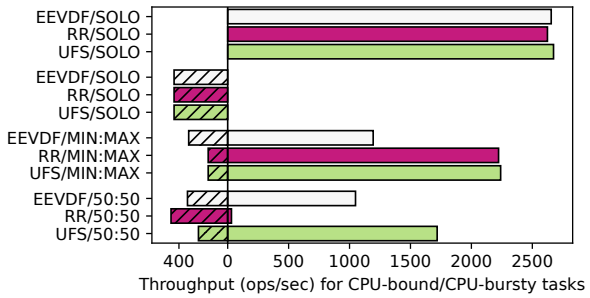


Figure 8: Throughput for a workload mix of CPU-bursty and ML workloads. Left shows MADlib logistic-regression training; right shows CPU-bursty workload.

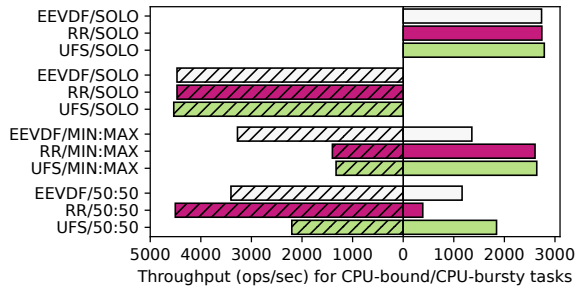


Figure 9: Throughput for CPU-bound vs. CPU-bursty tasks (shown left/right resp.), running solo or mixed *on MariaDB*.

Linux kernel feature that ensures tasks under the default policy (`SCHED_NORMAL`) receive at least 5% of the run time on a given CPU, preventing real-time tasks from fully starving normal tasks [7, 54]. The refresh therefore receives more CPU time under FIFO and RR than under EEVDF, although it still progresses at a much slower rate than in the baseline.

UFS, through application-based scheduler hinting, completes the experiment in about 18 seconds, approximately 2.4 times the baseline duration. The refresh task is boosted and temporarily treated as time-sensitive, allowing it to share CPU time with the four Q17 workers. It can therefore complete the refresh and release the lock required by the high-priority query.

Discussion. In MVCC systems, priority inversion should be rare. However, if it happens just once in production, it can constitute a major problem. As table-level exclusive locks are used infrequently, developers might not be aware of the impact they might have in a priority-based system. Our application hinting approach can transparently avoid a major impact. Standard Linux schedulers cannot provide this, as they are unaware of application semantics.

6.7 ML-based Workloads

UFS was designed not only to support complex analytical SQL, but also lengthy ML jobs that, if run within the DBMS, should not interfere with interactive workloads. As such, this section evaluates UFS with an in-database machine-learning background workload.

Setup. We reuse the workloads and configurations from Section 6.1, but this time, the CPU-bound background workload is ML-based. The ML tasks run inside PostgreSQL using Apache MADlib[9], more specifically, its logistic-regression training routine `logregr_train`. Each ML task repeatedly trains a logistic-regression model over a fixed input table, so the workload exercises PostgreSQL execution as well as MADlib's ML implementation.

Results. Figure 8 shows the throughput results (measured in number of iterations for the regression task). UFS preserves the same qualitative behavior observed for the earlier mixed workloads. In the MIN:MAX configuration, UFS outperforms EEVDF by better protecting the CPU-bursty workload from the background ML tasks. In the 50:50 configuration, UFS again outperforms RR while maintaining a fairer balance between the workloads than EEVDF.

Discussion. These results indicate that UFS's benefits are not limited to traditional mixed database workloads. Its main feature is that it can handle workload mixes of both CPU-bursty and CPU-bound work, regardless of the actual computation. As such, UFS is a promising approach for emerging in-database ML.

6.8 Overhead

The `sched_ext` callbacks could introduce overhead, while the shared eBPF maps could cause contention under high load. However, a micro-experiment with increasing workload and number of cores (up to 32) revealed that `select_cpu` and `enqueue` callbacks remain well under 1 µs, and `dispatch` under 2 µs even with a throughput of over 250k callbacks per second. Latencies per map access across various access types increased only by 20% when we moved from an aggregated map rate of 3.6 Mop/s to 13.6 Mop/s. This indicates that contention remained limited even at the highest tested rate.

To analyze the overhead of application hinting specifically, where PostgreSQL also has to access the maps, we ran UFS with the MIN:MAX configuration as described in Section 6.1, with application-hinting enabled and disabled. Throughput differences were negligible ($\leq$ 1%) for runs with and without application-based hinting and within the range expected from run-to-run variability. The main reason for this is that hint-related work is triggered only in the wait-event paths and has negligible overhead.

6.9 MariaDB

We ran the experiment described in Section 6.1 also with MariaDB. Figure 9 shows the results. Conceptually, we can observe the same behavior for all schedulers as with PostgreSQL. EEVDF substantially degrades time-sensitive task performance in the presence of background tasks, while the real-time schedulers handle equal-priority mixed workloads poorly. In contrast, UFS behaves as requested.[10]

7 RELATED WORK

Scheduling has been well explored by both the database and operating system communities.

[9]In order to run Apache MADlib in PostgreSQL 17, we built from commit `d03af81` with the PostgreSQL 17 support patch from PR 633.

[10]The background TPC-H throughput in MariaDB is much higher than with PostgreSQL because MariaDB uses a much better execution plan.

7.1 Database Scheduling

Concurrency control focuses on isolating conflicting concurrent transactions. Many database systems are now multi-version and use optimistic concurrency control for read-only transactions [6, 27, 30, 56], avoiding locks for reading. How runnable transactions access the CPU is traditionally left to the OS.

Resource Scheduling. Fair or priority-based resource sharing within the DBMS has been widely studied [4, 5, 19, 38, 50, 59]. Pisces [50] is a replicated and partitioned storage service that ensures proportional resource sharing among workloads while redistributing unused resources. Cheng et al. [5] ensure fair sharing under both physical resource and data contention. PreemptDB [19] enforces preemption of low-priority transactions using userspace interrupts. Code snippets that might lead to priority inversion must be marked, and a transaction cannot be preempted within this region even if there are no conflicts. UFS, in contrast, avoids preemption only on lock conflicts. PreemptDB also runs a high-priority transaction to completion before scheduling the next one on the same CPU and thus cannot properly handle blocking events such as I/O. UFS, instead, gives the CPU to another task if one blocks, may run several high-priority tasks concurrently on the same CPU, or migrate tasks. It also supports differently weighted tasks within a tier.

Commercial systems like Oracle or DB2 allow to specify service classes with resource constraints (e.g., CPU) [21, 22, 39]. However, their documentation does not describe the implementation nor mention priority inversion.

Recent work proposes fine-grained CPU scheduling where analytical queries are split into small morsels, and parallel execution is controlled by the DBMS [11, 25, 46, 57]. Transactions can also be bundled into co-routines to fully utilize the CPU [15, 18].

Real-time database schedulers use priorities to ensure transaction deadlines [41, 49, 62]. HTAP engines [24, 45, 47, 51] specialize in mixed workloads. Most of the effort has been on concurrency control, efficient database architectures, and data structures [51]. In contrast, external query schedulers sit on top of the DBMS, avoiding deep DBMS modifications [1, 58]. Their focus is on predicting query execution time for concurrently executing queries.

DBMS-OS Integration. Offloading DBMS tasks to the OS and bypassing user space via eBPF is being explored increasingly. For example, *Tigger* [3] is a PostgreSQL connection proxy that uses eBPF to forward network packets directly within the kernel, avoiding context switches and data-copy overhead. *BPF-DB* [2] is a kernel-embedded DBMS that offers ACID-compliant transactions to eBPF applications. The authors of [26, 64] propose to completely rethink how a DBMS could run with a specialized OS. In contrast, our goal is not to change the DBMS or build a new OS, but to use eBPF as a bridge between DBMS and OS for workload scheduling.

7.2 Operating System Scheduling

Priority-based Scheduling has long been supported by operating systems. Unix has included the `nice` utility since 1973 [55]. The fair scheduler CFS, and then EEVDF and cgroups followed. Shortcomings of these priority-based "fair" schedulers, including the interference between CPU-heavy and interactive workloads, were discussed in previous work [28, 35]. However, the proposed patches are presented as workload-specific and "not intended as generic bug fixes" [34, 65]. The open-source community also noted poor performance and offered patches [8, 60, 61]. Several of them hinted at the same underlying concern of CFS's and EEVDF's initial idle-CPU placement and wake-up heuristics. Ultimately, none of the proposed patches has ever been merged upstream. This is understandable, as maintainers are cautious about workload-sensitive heuristics that may improve a narrow class of workloads while risking regressions elsewhere. Yet with eBPF and `sched_ext`, we can implement an application-specific scheduler, including application-based hinting, without changing the Linux kernel or adding new interfaces to the kernel. UFS is different from previous patches not only because it uses eBPF and `sched_ext`, but also because it defines two tiers and offers proportional scheduling within each tier.

Application-informed scheduling. GhOSt [20] established the userspace-delegated scheduling paradigm, advocating application-informed scheduling, and heavily influenced `sched_ext`. Several `sched_ext` implementations exist, targeting specific workloads. `scx_layered` is designed for large time-sensitive production workloads. It classifies tasks into a set of predefined layers and dynamically adjusts the specific CPUs assigned to each layer. In contrast, UFS can dynamically create cgroups in each tier as needed, and also handles priority inversion in database workloads. `scx_lavd` infers latency sensitivity from task behavior, mainly for video gaming.

Park et al. [42, 43] push application semantics into the kernel to manage contention for OS resources such as kernel locks. UFS applies an analogous approach to a different resource: CPU cycles. pBox [17] is an OS abstraction that allows applications to inform the OS about their threads accessing virtual resources such as application-specific common data structures. The OS can then predict and detect contention and enforce fair scheduling.

Specialized Operating Systems. Recently, completely new operating systems and/or schedulers have been proposed, in particular for cloud-based systems where short and long tasks must co-exist [14, 23, 29, 37]. In these systems, different applications are assigned different cores in principle.

8 CONCLUSION

UFS is a selectively unfair, two-tier scheduler that lets background tasks run only on idle CPU cycles and preempts them for time-sensitive workloads. Within each tier, weight-proportional scheduling is maintained. UFS is implemented as an eBPF program in the Linux `sched_ext` framework, has low overhead and handles priority inversion by allowing DBMS/OS information exchange via eBPF maps. UFS mitigates the negative impact CPU-bound tasks have on CPU-bursty workloads in Linux's existing standard schedulers. Our PostgreSQL prototype requires a lock-hinting patch of only ~200 lines and improves interactive throughput by up to 2×.

Future work will analyze whether UFS's priority concepts apply to analytical systems such as DuckDB or DataFusion (e.g., to assign fine-grained weights to tasks or data pipeline steps). Another direction is GPU scheduling, especially for ML-based analytics.

ACKNOWLEDGMENTS

This project/research was partly funded by the Passau International Centre for Advanced Interdisciplinary Studies (PICAIS) of the University of Passau, Germany.

REFERENCES

[1] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2008. QShuffler: Getting the Query Mix Right. In *2008 IEEE 24th International Conference on Data Engineering*. 1415–1417. https://doi.org/10.1109/ICDE.2008.4497574

[2] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M Patel, and Andrew Pavlo. 2025. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. *Proc. ACM Manag. Data* 3, 3, Article 135 (June 2025), 27 pages. https://doi.org/10.1145/3725272

[3] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A Database Proxy That Bounces with User-Bypass. *Proc. VLDB Endow.* 16, 11 (July 2023), 3335–3348. https://doi.org/10.14778/3611479.3611530

[4] Michael J. Carey, Rajiv Jauhari, and Miron Livny. 1989. Priority in DBMS Resource Scheduling. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 397–410. http://www.vldb.org/conf/1989/P397.PDF

[5] Audrey Cheng, Aaron Kabcenell, Xiao Shi, Jolene Huey, Peter Bailis, Natacha Crooks, and Ion Stoica. 2025. Fair transaction processing for multi-tenant databases. *Proceedings of the VLDB Endowment* 18, 8 (2025), 2602–2615.

[6] Audrey Cheng, Aaron N. Kabcenell, Jason Chan, Xiao Shi, Peter D. Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* 17, 11 (2024), 2694–2707. https://doi.org/10.14778/3681954.3681956

[7] Daniel Bristot de Oliveira. 2023. DL Server. Linux Plumbers Conference 2023. https://lpc.events/event/17/contributions/1491/attachments/1239/2519/DL%20Server%20LPC%202023.pdf, Accessed: 2026-04-21.

[8] Mathieu Desnoyers. 2023. [RFC PATCH] sched/fair: Bias runqueue selection towards almost idle prev CPU. https://lore.kernel.org/lkml/20230905171105.1005672-2-mathieu.desnoyers@efficios.com/. Accessed: 2026-10-26.

[9] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[10] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA: Abstraction for Advanced in-Database Analytics. *Proc. VLDB Endow.* 11, 11 (July 2018), 1400–1413. https://doi.org/10.14778/3236187.3236194

[11] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.* 12, 12 (2019), 2218–2229. https://doi.org/10.14778/3352063.3352137

[12] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. The UDFBench Benchmark for General-purpose UDF Queries. *Proc. VLDB Endow.* 18, 9 (2025), 2804–2817. https://doi.org/10.14778/3746405.3746409

[13] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *Proc. VLDB Endow.* 16, 12 (2023), 3874–3877. https://doi.org/10.14778/3611540.3611574

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried Accessed: 2026-10-26.

[15] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444. https://doi.org/10.5555/3430915.3442440

[16] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[17] Yigong Hu, Gongqi Huang, and Peng Huang. 2023. Pushing performance isolation boundaries into application with pbox. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 247–263.

[18] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proc. VLDB Endow.* 17, 3 (2023), 577–590. https://doi.org/10.14778/3632093.3632117

[19] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? *Proc. ACM Manag. Data* 3, 3 (2025), 182:1–182:25. https://doi.org/10.1145/3725319

[20] 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 *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 588–604.

[21] IBM Redbooks. 2008. *DB2 Workload Manager for Linux, UNIX, and Windows*. IBM Corporation, Poughkeepsie, NY, USA. https://www.redbooks.ibm.com/abstracts/sg247524.html Accessed: 2026-10-26.

[22] IBM Redbooks. 2023. *IBM Db2 13 for z/OS Performance Topics*. Number SG24-8536-00 in IBM Redbooks. IBM Corporation, Poughkeepsie, NY, USA. https://www.redbooks.ibm.com/redbooks/pdfs/sg248536.pdf IBM Form Number: SG24-8536-00. Accessed: 2026-10-26.

[23] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[24] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1675–1687. https://doi.org/10.1145/2882903.2882905

[25] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[26] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.* 17, 8 (2024), 2115–2122. https://doi.org/10.14778/3659437.3659462

[27] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-Version Range Concurrency Control in Deuteronomy. *Proc. VLDB Endow.* 8, 13 (2015), 2146–2157. https://doi.org/10.14778/2831360.2831368

[28] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel (Eds.). ACM, 4:1–4:14. https://doi.org/10.1145/2592798.2592821

[29] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 922–936. https://doi.org/10.1109/HPCA57654.2024.00075

[30] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 21–35. https://doi.org/10.1145/3035918.3064015

[31] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1794–1807. https://doi.org/10.1109/ICDE53745.2022.00180

[32] Linux kernel documentation. [n.d.]. Extensible Scheduler Class (sched_ext). https://docs.kernel.org/scheduler/sched-ext.html. Accessed: 2026-01-04.

[33] Linux kernel documentation. 2026. EEVDF Scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html. Accessed 2026-05-01.

[34] Jean-Pierre Lozi. 2016. *wastedcores: Important note about the patches.* https://github.com/jplozi/wastedcores Repository README section stating the patches are not intended as generic bug fixes. Accessed: 2026-10-26.

[35] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. ACM, 1:1–1:16. https://doi.org/10.1145/2901318.2901326

[36] Chris Mason. 2016. schbench: Scheduler Benchmark. https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench. Accessed: 2026-04-10. A scheduler benchmark that measures wakeup latency and throughput.

[37] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 1–18. https://www.usenix.org/conference/nsdi22/presentation/mcclure Accessed: 2026-10-26.

[38] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. 2005. Improving Preemptive Prioritization via Statistical Characterization of OLTP Locking. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 446–457. https://doi.org/10.1109/ICDE.2005.78

[39] Oracle Corporation. [n.d.]. *Oracle Database 19c: Managing Resources with Oracle Database Resource Manager*. Oracle Corporation. https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-

resources-with-oracle-database-resource-manager.html Accessed: 2026-03-20.

[40] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions Into DBMSs. *IEEE Trans. Knowl. Data Eng.* 35, 10 (2023), 10295–10308. https://doi.org/10.1109/TKDE.2023.3269592

[41] Sarvesh Pandey and Udai Shanker. 2020. Transaction scheduling protocols for controlling priority inversion: A review. *Comput. Sci. Rev.* 35 (2020), 100215. https://doi.org/10.1016/J.COSREV.2019.100215

[42] Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2021. Contextual concurrency control. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 167–174.

[43] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 667–682.

[44] The Linux Kernel Documentation Project. [n.d.]. *BPF Documentation*. https://docs.kernel.org/bpf/index.html Accessed: 2026-01-11.

[45] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric N. Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2340–2352. https://doi.org/10.1145/3514221.3526055

[46] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2 (2016), 37–48. https://doi.org/10.14778/3015274.3015275

[47] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer, 97–112. https://doi.org/10.1007/978-3-319-15350-6_7

[48] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proc. VLDB Endow.* 17, 8 (2024), 1830–1842. https://doi.org/10.14778/3659437.3659441

[49] L. Sha, R. Rajkumar, and J.P. Lehoczky. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185. https://doi.org/10.1109/12.57058

[50] David Shue, Michael J Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 349–362.

[51] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *VLDB J.* 33, 5 (2024), 1485–1515. https://doi.org/10.1007/S00778-024-00858-9

[52] Ion Stoica and Hussein Abdel-Wahab. 1995. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Technical Report TR-95-22. Old Dominion University. https://people.eecs.berkeley.edu/~istoica/papers/eevdf-tr-95.pdf Revised January 1996.

[53] The Linux kernel community. [n.d.]. Control Groups Documentation. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html. Accessed: 2026-01-11.

[54] The Linux Kernel Development Community. 2025. Real-Time group scheduling. https://docs.kernel.org/scheduler/sched-rt-group.html. Accessed: 2026-04-21.

[55] Ken Thompson and Dennis M. Ritchie. 1973. *UNIX Programmer's Manual, Fourth Edition*. Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. https://dspinellis.github.io/unix-v4man/v4man.pdf Accessed: 2026-10-26.

[56] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[57] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1879–1891. https://doi.org/10.1145/3448016.3457260

[58] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Accurate Performance Prediction on Concurrent Queries. *Proc. VLDB Endow.* 18, 11 (July 2025), 4185–4198. https://doi.org/10.14778/3749646.3749686

[59] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1 (2023), 44:1–44:24. https://doi.org/10.1145/3588724

[60] Chen Yu. 2023. [PATCH v8 0/2] sched/fair: Introduce SIS_CURRENT to wake up short task on current CPU. https://lkml.org/lkml/2023/4/28/518. LKML posting (v8), with links back to earlier versions including v1 (Sep 2022). Accessed: 2026-10-26.

[61] Chen Yu. 2023. Re: [RFC PATCH] sched/fair: Bias runqueue selection towards almost idle prev CPU. https://lists.openwall.net/linux-kernel/2023/09/30/89. Accessed: 2026-10-26.

[62] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. 1994. On real-time databases: concurrency control and scheduling. *Proc. IEEE* 82, 1 (1994), 140–157. https://doi.org/10.1109/5.259432

[63] Yunjia Zheng, Yuxuan Tian, Joseph Vinish D'silva, and Bettina Kemme. 2023. DBMLSched: Scheduling In-database Machine Learning Jobs. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings)*. CEUR-WS.org. https://ceur-ws.org/Vol-3462/AIDB3.pdf Accessed: 2026-10-26.

[64] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical DB-OS Co-Design with Privileged Kernel Bypass. *Proc. ACM Manag. Data* 3, 1 (2025), 64:1–64:27. https://doi.org/10.1145/3709714

[65] Peter Zijlstra. 2016. Re: [RFC] The Linux Scheduler: a Decade of Wasted Cores Report. Linux Kernel Mailing List (LKML). https://lkml.org/lkml/2016/4/25/176 Email dated 2016-04-25. Accessed: 2026-10-26.