The Kernel Always Knows: Understanding Isolation via Kernel Objects

Anjali
Department of Computer Sciences
University of Wisconsin-Madison
Madison, Wisconsin, USA
anjali@wisc.edu

Michael M. Swift
Department of Computer Sciences
University of Wisconsin-Madison
Madison, Wisconsin, USA
swift@cs.wisc.edu

Abstract

Good fences make good neighbors.

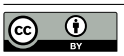
Robert Frost, Mending Wall

Cloud providers employ a spectrum of isolation platforms–ranging from lightweight containers to hardware-assisted microVMs, each designed on the principles of limiting interaction with the host kernel by minimizing their attack surface and strengthening isolation. An underlying theme of these platforms is to move functionality away from the host kernel; however, they still rely on *kernel objects* to implement core system functionality such as memory management, file systems, I/O, and scheduling. As a result, isolation is not determined solely by whether a platform virtualizes a resource, but by which kernel objects remain shared and how workloads interact with them. The platform-specific nature of these dependencies and the *shared kernel state* they inevitably expose remains a poorly understood variable in the isolation equation. Shared kernel state creates implicit structural dependencies between workloads that shape isolation guarantees, introduce interference risks, and ultimately constrain how isolation platforms can be designed, evaluated, and strengthened.

We propose a novel approach to measure this isolation gap by analyzing the usage of *shared kernel objects* across three different isolation platforms. We leverage the observation that access to these shared objects is synchronized. By measuring synchronization activity between workloads on these platforms, we can measure their ability to interact and the degree to which platform design exposes shared kernel state and thus compromises data isolation.

We build *LIMA* (Lock Interference Mapping Analyzer)[1], a fine-grained tracing tool to map the footprint of shared kernel objects and apply it to a comparative analysis of LXC, gVisor, and Firecracker, running a broad set of workloads. Our analysis identifies the isolation properties of different objects under different workloads, and finds that the file system journal and kernel page allocator are the most common sources of shared data. Moreover, this work reveals distinct object sharing profiles for platforms, implying isolation is not a binary property; it is a spectrum shaped by platform design choices that redistribute and reshape object-level sharing.

[1] https://github.com/multifacet/lima



SoCC '26, Singapore, Singapore

ACM ISBN 979-8-4007-2735-1/26/11
https://doi.org/10.1145/3815789.3827955

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Operating systems**; • **Security and privacy → Virtualization and security**.

Keywords

operating systems, virtualization, isolation, cloud computing, containers, firecracker, gvisor

ACM Reference Format:
Anjali and Michael M. Swift. 2026. The Kernel Always Knows: Understanding Isolation via Kernel Objects. In *ACM Symposium on Cloud Computing (SoCC '26), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3815789.3827955

1 Introduction

Isolation is a fundamental requirement in cloud environments, where mutually distrusting tenants share physical hardware. Cloud providers employ diverse isolation platforms to isolate co-located workloads [1, 23, 26, 30, 45, 56]. These platforms, such as Google's gVisor and AWS Firecracker, follow a *lightweight virtualization* design by either stripping down functionality of full system virtualization, such as in a microVM, or by adding more functionality in userspace to limit interactions with the shared host platform, such as in a Secure Container. However, even highly isolated platforms ultimately depend on shared kernel resources, such as memory management, file systems, and schedulers, that are managed through *shared kernel objects*. These kernel objects are kernel-resident data structures that hold system state and coordinate access to shared resources (*e.g.,* inodes, page allocators, scheduling queues).

To manage contention on shared resources, platforms additionally employ OS-level (*e.g.,* cgroups, qdisc) and hardware-level (*e.g.,* cache partitioning) mechanisms to partition CPU, memory, and network bandwidth [8, 17, 18, 28, 53, 62]. These mechanisms provide isolation by controlling the *amount* of resources available to a workload and are effective in limiting interference arising from resource allocation. However, resource partitioning does not eliminate interference caused by shared access to kernel objects. This *shared kernel state* introduces invisible structural dependencies between workloads that can undermine isolation guarantees, manifesting as unpredictable performance behavior (*e.g.,* noisy neighbors) [7], or security vulnerabilities (*e.g.,* framing attacks [48] or denial of service [25]) that are difficult to reason about without a *data isolation* view, *i.e.,* visibility into how kernel-managed data structures are shared across platforms.

Moreover, understanding and measuring this shared state are critical requirements for evolving the design of isolation mechanisms. Many emerging designs can be viewed as pursuing *state minimalism,* aiming to reduce, restrict, or privatize kernel state that is exposed across workloads. Specifically, Address Space Isolation (ASI) [12, 13] restricts the sensitive kernel state accessible to the currently executing process to prevent leakage, while other approaches aim to create private copies of highly shared data for each container instance to eliminate inter-tenant interference [25]. Similarly, Library OS designs [40, 50, 56, 58] migrate OS services out of the shared host kernel, thereby reducing shared kernel state. However, enforcing these boundaries within a monolithic kernel requires identifying existing structural dependencies. The kernel state cannot be effectively partitioned or privatized without understanding which objects are implicitly shared. A systematic study of kernel object sharing is therefore essential not only for assessing current isolation guarantees, but also for providing the empirical foundation needed to design stronger isolation mechanisms.

The key challenge in understanding object-level sharing lies in efficiently identifying shared structures within a large and complex kernel codebase. Prior work [10] has addressed this challenge by analyzing memory-level sharing using full-system emulation, revealing severe contention but at the high cost of lengthy experiments. We instead leverage synchronization of shared state: By design, the operating system synchronizes access to shared kernel objects. Hence, we use synchronization events as a proxy for potential object interference between workloads.

A key insight of this approach is that object-level isolation is not a static property of the platform, but a dynamic function of the workload. Workloads that rarely synchronize (*e.g.,* CPU-bound tasks in user-space) interact with little shared kernel state and effectively operate on disjoint objects, achieving high isolation regardless of the underlying platform. A platform may provide strong isolation for compute-heavy workloads, yet expose substantial shared state under workloads with high dependence on system services. By measuring synchronization, we capture the *effective* isolation experienced by workloads across different platforms.

A fundamental challenge in analyzing isolation at the system software layer is identifying which kernel objects are shared among concurrent workloads. Current profiling tools (*e.g.,* perf, ftrace [49, 51]) can expose where time is spent, but not which kernel objects are shared or how platform design shapes that sharing. Although prior work has used tracing to study performance interference and isolation [15, 16, 46, 55], these approaches do not provide an object-level view of kernel state sharing across platforms.

Motivated by this need, we develop a tool, *LIMA* (Lock Interference Mapping Analyzer), to capture fine-grained system-level interference using shared kernel locks. *LIMA* combines dynamic tracing of kernel locking events with static analysis to map kernel locks to the underlying data structures they protect. We use *LIMA* to conduct a comparative study to understand the sharing and interference observed at the system software level via shared objects when concurrent workloads execute on different isolation platforms. Our focus is on the *isolation of system structure*, meaning how well isolated the data structures of system software are. As many hardware isolation issues are orthogonal to the software structure (*e.g.,* micro-architectural side-channels), we do not measure them in this work [4, 35, 36, 43].

Platform / Resource	**KVM**	**LXC**	**gVisor**	**Firecracker**
Kernel	Isolated	Shared	Shared	Isolated
Network	Same as host	Same as host	Shared/ Isolated	Same as host
Memory Allocation	Isolated	Shared	Shared/ Isolated	Isolated
Filesystem	Isolated	Shared/ Isolated	Shared/ Isolated	Isolated
Hardware	Shared	Shared	Shared	Shared

Table 1: Sharing across different platforms.

We make the following contributions in this work:

- We implement *LIMA*, a tool to detect fine-grained kernel object-level sharing via dynamic tracing and static analysis.
- We use *LIMA* to collect and analyze kernel-level lock traces to measure system-level sharing via objects across a diverse set of applications and isolation platforms.
- LIMA enables direct comparison of platforms by grounding isolation behavior in measurable kernel object sharing rather than platform abstractions.
- We show that isolation is shaped by platform design. Linux containers share a wide variety of objects with high intensity; Firecracker narrows this variety but accesses specific hot objects intensely, especially during initialization; and gVisor shares across a moderate number of objects while maintaining lower acquisition rates.
- We identify memory allocation and file system journaling as the dominant sources of cross-application interference.

2 Background and Motivation

2.1 Isolation and Interference

Isolation in computer systems keeps workloads apart from each other, so that the behavior (functional or temporal) of one workload does not affect other workloads. The goal for perfect isolation is *non-interference* [34, 44], where a workload executes identically to how it would when run alone. While often used for security by looking at what is possible for an attacker, for isolation, we consider it in terms of *structural independence*: a workload should function as if it has exclusive access to the system resources and kernel state required for its execution. Thus, an imperfect isolation mechanism can provide perfect isolation if the two workloads do not exercise or contend on shared system facilities that allow interference.

The basic approach that isolation systems take is to separate workloads in space and time. Spatial isolation ensures workloads access different data or hardware (*e.g.,* separate memory pages, dedicated CPU cores), while temporal isolation prevents them from accessing the same data or hardware simultaneously (*e.g.,* CPU time-slicing). These approaches are highly effective for resource management, as they control the amount of resources a workload receives. Mechanisms like Linux cgroups [31], qdisc [27], and cache partitioning [8] enforce strict spatial and temporal boundaries on resource allocation when sufficient resources are available.

However, resource partitioning does not fully isolate because it ignores *state*. Workloads depend on the host kernel for functionality,

creating a dependency on shared kernel objects. For instance, two containers that are assigned separate CPU cores and memory limits via cgroups may still interfere if they repeatedly access the global inode cache or filesystem journal, despite no violation on assigned resource limits. To capture this, we define *Data Isolation* as the property ensuring that the kernel objects accessible by one workload are structurally disjoint from those of another. We quantify this via the workload's *Data Footprint*: the set of all kernel objects it accesses during execution.

2.2 Performance vs. Efficiency

Stronger forms of isolation often provide increasingly more private resources and fewer shared resources. At the extreme, running a workload on separate physical machines provides extreme isolation, while workloads running in standard OS processes share most kernel resources. Virtual machines lie in the middle and give workloads a private guest kernel managing partitioned memory and often dedicated CPUs. We observe that isolation is often opposed to efficiency: as a platform increases isolation, it decreases the amount of sharing across workloads, which reduces the opportunity for interference but comes at the cost of efficiency. This provides a strong motivation to find the *right level of isolation* for a workload — using a too-strong platform wastes resources, while a too-efficient platform with sharing increases the risk of interference. Table 1 shows the resources that are shared/isolated across different isolation platforms.

Moreover, the *right* level of isolation depends on the workload. Workloads that rely minimally on kernel services may safely leverage the high efficiency and ease of sharing (*e.g.*, shared libraries) of a less isolated platform. Workloads that frequently interact with kernel subsystems are more sensitive to shared kernel state. However, we lack visibility into a workload's *data footprint* on a platform to determine which platform is appropriate so it is difficult to assess if a platform's efficiency gains come at the cost of detrimental sharing.

2.3 Synchronization as an Identifier of Sharing

Interference and synchronization. Interference implies that some shared resource between workloads acts as an agent of interference. Our key insight is that operating systems *synchronize all access to shared resources*, whether a physical resource (memory, devices) or a logical resource (an object). For instance, concurrent processes must synchronize via memory allocator locks when requesting physical pages, just as they use inode locks when accessing a shared file to ensure consistency. Thus, any interference from shared objects workloads will be controlled by synchronization. This synchronization ranges from private locks protecting local process state, to shared locks guarding global resources, like `tasklist_lock`, which serializes access to the global task list.

We note that OS design can lead to *incidentally* shared data: processes logically access different data, but the OS design requires access to shared data, similar to false sharing with locks. For example, workloads accessing different files on a shared file system, may synchronize access to shared caches such as the Linux `dcache` or `inode` cache and access shared hash structures.

Locks and synchronization. Shared kernel resources require synchronization to ensure a safe and correct order of execution of concurrent processes in the system. However, from an isolation perspective, not all synchronization mechanisms contribute equally to interference. Read-only sharing does not lead to interference, as each process can logically operate on a copy of data without affecting other processes. Our goal is to quantify interference arising from shared kernel state, and thus we focus on synchronization mechanisms that are most likely to impact isolation.

The Linux kernel employs two categories of synchronization primitives: (a) *Lock-based synchronization*, including spinlocks (and variants such as reader–writer locks and seqlocks), mutexes, and semaphores; and (b) *Lockless synchronization*, including atomic operations, Read-Copy Update (RCU), and memory barriers. A key difference between these mechanisms is that lockless synchronization is typically non-blocking [6, 14, 61]. For example, RCU [41] is optimized for read-heavy sharing, allowing readers to access data concurrently without blocking even during updates. Therefore, interactions in lockless paths are transient and usually allow execution to proceed in parallel, making them less likely to expose blocking-induced performance interference. However, from a data-isolation perspective, lockless mechanisms can still represent shared kernel state that is not captured by our approach. We do not expect this limitation to substantially affect subsystems that primarily rely on lock-based synchronization.

In contrast, locks enforce mutual exclusion to serialize access to shared kernel data; hence, we believe that they impact interference and isolation more. This distinction motivates our focus on lock-based synchronization as a proxy for shared kernel data access. Therefore, we use lock-based synchronization as our primary proxy for sharing. While this approach does not capture all forms of sharing, it provides a starting point for quantifying data isolation through kernel synchronization.

	Lock name	Struct name	Struct path	Acq/sec
lxc	mapping->private_lock	address_space	include/linux/fs.h	527
	journal->j_list_lock	journal_s	include/linux/jbd2.h	444
	sbi->s_orphan_lock	ext4_sb_info	fs/ext4/ext4.h	333
	inode_hash_lock			174
	journal->j_state_lock	journal_s	include/linux/jbd2.h	78
	others			11.53
fc	journal->j_state_lock	journal_s	include/linux/jbd2.h	1214
	lruvec->lru_lock	lruvec	include/linux/mmzone.h	1.25
	zone->lock	zone	/include/linux/mmzone.h	0.3
	journal->j_state_lock	journal_s	include/linux/jbd2.h	0.12
	journal->j_wait_commit	journal_s	include/linux/jbd2.h	0.03
	others			0.11

Table 2: Top-5 locks and other locks for *lxc* and *fc* for *listdir*.

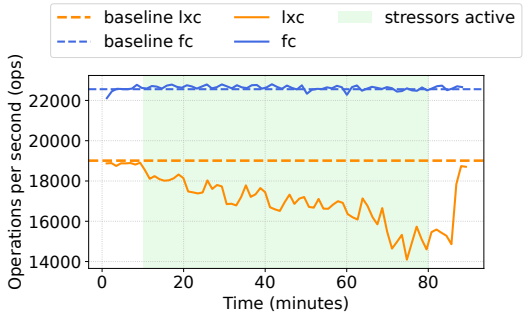


Figure 1: Impact of interfering workloads on *listdirs* performance over time.

3 Case Study

Isolation platforms fundamentally shape how workloads interact with host kernel state. In this section, we show that the same workload exhibits different kernel data footprints depending on the choice of platform architecture, revealing distinct interference vectors *i.e.,* shared kernel objects through which concurrent workloads interact and interfere. We use *listdirs* from filebench [57], which repeatedly enumerates directory entries. It exercises the filesystem metadata path and stresses both fine-grained per-object locks and coarse global locks, making it well-suited for studying the *shared data footprint* across different levels of granularity. We use this workload to highlight several key aspects of object-level sharing: the impact of isolation platform designs, the role of incidental locks on interference, and the impact of co-location with other workloads.

We study native Linux containers built on cgroups (*lxc*), as implemented by Docker's *runc* container engine [19, 47] and Firecracker (*fc*) [1] because they represent two distinct points in the isolation design space. *lxc* provides OS-level isolation by sharing a single host kernel across workloads, while *fc* uses a microVM architecture with a dedicated guest kernel, yet still relies on the host for key functionalities. This contrast lets us examine how platform structure shapes shared kernel state.

We run two concurrent instances of *listdirs* on the same platform and record the lock accesses for each instance. We then identify shared locks by intersecting the sets of unique lock addresses accessed by both workloads. Using lock addresses, we can distinguish between global locks and per-object locks. For example, `inode_hash_lock` is a global lock, `inode->i_lock` is a distinct lock instance within every open file. Our methodology counts locks as shared only when both workloads access the same underlying object instance, ensuring we capture sharing of specific kernel data.

While it is widely understood that microVMs offer stronger isolation, our goal is to move beyond qualitative wisdom and quantify the specific kernel structures that form the interference vector in each platform. In Table 2, we show the top shared locks (locks held in common across workloads) accessed by these platforms and their lock access rates (in locks acquired/sec).

The results reveal a structural difference in how these platforms expose the kernel. Under *lxc*, the workload interacts with a broad set of objects, spanning security (*e.g.,* `aa_buffers_lock`), process management, and filesystem objects. *fc* accesses a narrower set of memory-management objects (*e.g.,* `struct lruvec` holding candidates for page replacement via `lru_lock`), consistent with *fc*'s guest kernel performing most memory management. *fc* concentrates activity on a few shared objects at high frequency, whereas *lxc* spreads sharing across many objects, quantifying the narrower interface of microVMs compared to containers.

A few kernel objects dominate sharing. We observe a few highly accessed shared objects on both platforms that abstract managing shared hardware resources. For memory, we see access to the memory zones, `struct zone`, which organizes and tracks state for physical memory zones. For CPUs, we see the schedulers `tasklist_lock` protecting the global task list, and for the file system we see accesses to `journal_s`, which manages the journal state, synchronized via `j_list_lock` and `j_state_lock`. These recurring shared objects highlight that kernel-level sharing centers on a few critical structures that dominate the shared data footprint.

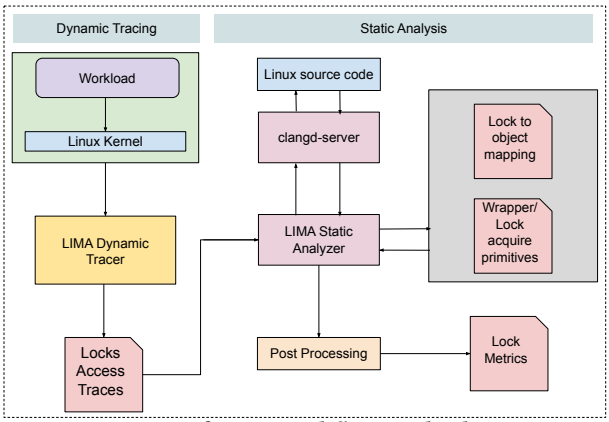


Figure 2: Overview of *LIMA* workflow. In the dynamic tracing phase, *LIMA* collects lock accesses across workloads, and the static analysis phase completes the lock-to-object mapping.

Incidental sharing matters for interference. We define incidental sharing as cases where workloads logically access disjoint data but, due to OS design, synchronize on shared kernel structures. Locks related to incidental sharing like the global `inode_hash_lock` which protects the inode cache hash table show a significant access rate (174 times/sec for *lxc*). Because such sharing arises from kernel data structures like hash tables rather than application logic, it is difficult for tenants to anticipate or avoid. This implies that even workloads without obvious overlap in their data paths may still contend for incidental locks, leading to interference.

Object sharing influences workload performance. Figure 1 shows the impact of object-level sharing on performance. We co-locate *listdirs* with interfering workloads (called *stressors*), running the same *listdirs* workload to create pressure on overlapping objects. We start one *stressor* at 10 minutes, adding additional ones every 10 minutes for 70 minutes. While *fc* exhibits a high level of object sharing in our traces, we observe visible performance degradation only for *lxc*. The lock tracing data show far more sharing of file system data (the top 5 locks acquired in *lxc* are all related to file system), and all are acquired more than 100 times/sec. In contrast, with *fc* only two journal locks are accessed, making it much more robust to competing workloads.

This case study illustrates how a single workload can reveal the role of object-level sharing in software-level interference, reflecting platform design, which kernel paths are exercised, and incidental locks. These findings highlight that object-level sharing is common during workload execution on modern isolation platforms.

4 LIMA Design

We implemented a data-collection tool named *LIMA* that identifies the kernel objects accessed by workloads running on different isolation platforms on Linux, enabling object-level analysis of shared kernel state. To achieve this, *LIMA* is designed with goals:

- *Workload-aware measurement*. Capture runtime lock activity as object sharing depends on real execution, rather than potential code paths.

- *Object-level attribution.* *LIMA* should reveal the underlying objects that those locks protect. This is important so that data sharing can be linked to specific kernel structures.
- *Cross-platform comparability*. Provide a uniform mechanism that works across diverse isolation platforms to allow systematic comparison of their isolation boundaries.
- *No modifications to the kernel.* Ability to run on the host without any modifications to the kernel code.

LIMA combines two complementary components: it traces lock acquisitions dynamically using eBPF and uses static analysis to resolve where in the code each acquisition occurs and, more importantly, what objects the lock protects. We show an overview of *LIMA* in Figure 2. These components enable attribution of runtime shared kernel state access to specific kernel objects.

Challenges. Building *LIMA* revealed several challenges that make object-level sharing analysis non-trivial. First, dynamic tracing via eBPF may produce incomplete stack traces, as inlining and tail-call optimizations [21, 22] can truncate frames before the actual lock acquisition. Second, system-wide tracing introduces noise, since locks may be acquired by unrelated processes or in interrupt contexts (*e.g.,* timers or softirqs), requiring careful filtering to isolate workload-induced sharing. Finally, tracing itself incurs overhead, which can perturb execution and potentially miss short-lived or low-frequency lock events.

Limitations. Despite our mitigations, *LIMA* has limitations. It does not trace lockless primitives such as RCU or atomics. Additionally, it cannot attribute lock activity in an interrupt context to a specific workload. *LIMA* can identify some forms of sub-structural locking, such as locks embedded in nested structures, but it cannot directly distinguish which fields are protected when multiple locks protect different fields within the same structure; in such cases, we rely on kernel documentation and source code when needed. Finally, *LIMA* currently does not resolve all cases where locks are acquired through macros or where object variables are reassigned to local variables within function bodies.

Generality. While kernel changes (*e.g.,* optimizations for multi-core scalability or restructuring of shared objects) may affect the degree and distribution of sharing, *LIMA* is designed to be reusable across kernel versions.

4.1 LIMA Dynamic Tracer

We extended and modified the eBPF-based tool *klockstat* [52] to implement the dynamic tracing component of *LIMA* for the dynamic lock usage. *LIMA* traces lock acquire and release functions of mutexes, semaphores, spinlocks, and reader-writer spinlocks. By instrumenting these primitives, *LIMA* provides comprehensive coverage of the kernel's lock-based synchronization mechanisms.

LIMA dynamic tracer produces a trace of every lock acquired with lock name (the kernel symbol associated with the lock variable), kernel stack trace, acquire time, hold time, and count of occurrences of that stack trace with that lock. In addition, we instrument lock initialization routines to learn when a lock address is reallocated for a new lock. We process this trace to calculate metrics such as the number of *shared locks* — locks acquired in common across workloads — accessed for different workloads and *private locks* — unique and non-shared locks acquired by each workload.

Dynamic analysis produces a trace of all the locks acquired by a running workload. Below is an example of the trace output. The stack details are stored in a different file and matched with the `stack_id` to get the stack traces with an acquisition.

PID	addr	name	count	process	stack_id
38296	0xffff888	&pipe->mutex	1	mmap04	324596

4.2 LIMA Static Analyzer

LIMA static analyzer processes the dynamic lock traces and completes the mapping of lock names to kernel objects. We implemented the *LIMA* static analyzer using the clangd [9] language server [39]. clangd is widely used in many IDEs for different functionalities (*e.g.,* symbol indexing and outgoing call analysis), and it provides convenient APIs for querying code properties.

As summarized in Table 3, *LIMA* static analyzer combines symbol generation to index lock definitions, call graph analysis (incoming/outgoing calls) to resolve incomplete dynamic stack traces and wrapper functions, and AST traversal to disambiguate generic lock members. These core functionalities enable the precise resolution of lock-to-object mappings *i.e.,* identifying the parent kernel structure (*e.g.,* `struct inode`) that contains a given lock member (*e.g.,* `i_lock`). This transforms lock names into semantically meaningful object references, as described in the steps below.

Input. *LIMA* static analyzer takes three primary inputs: the Linux kernel source code, the dynamic lock access traces collected at runtime and the kernel binary (for address resolution). It queries the indexed source code via the clangd server to access the core analysis functionalities.

Object Mapping. The core functionality of *LIMA* is to map an instance of lock acquisition to the object it protects. To do this, we rely on the call stack captured during tracing to identify the code location responsible for acquiring the lock. The stack allows us to determine which function invoked the locking primitive and, consequently, which object instance is being accessed. This involves:

Incomplete stack trace resolution: As noted in challenges, dynamic tracing often truncates stack traces, hiding the actual lock acquisition function. To recover this missing context, we leverage a database of *Outgoing Calls* (outgoing_calls_db) which we have built across the kernel source. Given an incomplete dynamic trace, we take as input the lock name, file, and function where tracing stopped, and then initiate a breadth-first search from that last function over outgoing_calls_db. The search terminates when we reach either a standard lock primitive or a known lock wrapper, a helper function (*e.g.,* mmap_read_lock) that encapsulates the locking logic, identified via our *Incoming Calls*–based wrapper database.

Object resolution: We generated a database for global locks (global_locks_db with 2488 entries) using *Symbol Generation* that contains information about the lock, as in this example:

lock_name	lock_type	file
tasklist_lock	rwlock_t	include/linux/sched/task.h

We also use *Symbol Generation* to create a mapping of locks to the `struct` within which they are defined (lock_struct_map_db), with details:

Functionality	Description
Symbol Generation	Extracts symbol details (name, type, signature, line range) to map locks to structs and list global locks (≈2488).
Outgoing Calls	Uses function info to identify outgoing calls and locate lock acquisition in incomplete stacks.
Incoming Calls	Identifies lock wrapper functions (335 found) from incoming call info, used as stop points in stack resolution.
AST	Builds abstract syntax trees to disambiguate "generic" locks (named "lock") and link them to the right object.
Get Definition	Returns file path and line number of a lock, matched with struct ranges from Symbol Generation.

Table 3: Core functionalities of *LIMA*'s static analysis.

Microbench	Description
fork (stress-ng)	repeatedly creates and reaps processes.
futex (stress-ng)	stresses futex wait/wake with threads.
sysbench	finds prime numbers for a fixed duration.
mem (1MB)	allocates 16GB using 1MB allocation size.
mem (8KB)	allocates 16GB using 8KB allocation size.
createfiles (filebench)	measures file creation performance.
deletefiles (filebench)	measures file deletion performance.
listdirs (filebench)	lists the contents of a large directory.

Table 4: Microbenchmarks with their descriptions.

Cloud Workloads
data analytics, data caching, data serving, graph analytics, in-memory analytics, media streaming
FunctionBench
image processing, video processing, model training, cnn, rnn, face detection
stress-ng
fork, vm (mem), io, fstat, getdent

Table 5: Cloud, serverless workloads and stress-ng stressors analyzed by *LIMA*.

file	struct_name	lock_type	lock_name	line_range
mm/slab.h	kmem_cache_node	spinlock_t	list_lock	751-779

We start the search for lock-to-object mapping by querying these two files first. A trace may identify an acquisition of a variable named `lock`. However, because hundreds of different structures in the Linux kernel use the name `lock`, simple text matching is insufficient. For example, it does not distinguish whether a specific lock named `lock` belongs to `fs_struct`, `zone`, or any other object.

To resolve this, we use the AST file of the function where *generic* locks are acquired to get the line number associated with the lock variable, which we use to query *Get Definition* for the lock symbol to find the location of its definition. We use this to locate the corresponding object name by matching the line number to line_range (the lock line number should fall within this range for a match) and file in lock_struct_map_db, hence completing the mapping.

Output. At the end of the processing, *LIMA* generates a file containing end-to-end lock to object mapping containing:

file	object_name	lock_type	lock_name
fs/ext4/ext4.h	ext4_inode_info	rw_semaphore	i_data_sem

For all unique locks—identified by their primitive, lock name, function, (which we use to match dynamic trace entries with the static lock-to-object mapping) and source file—acquired across all workloads analyzed in this paper (984 in total), we can map approximately 85% of them to their associated objects. Even when a lock cannot be fully resolved, our approach provides valuable contextual information about the subsystem and object involved by pointing to the line of lock acquisition in the kernel codebase.

5 Object Analysis Methodology

Our primary goal is to measure data interference arising from shared access to kernel resources. For each workload under test, we collect host kernel lock traces to understand how different isolation platforms use various kernel data structures. These traces allow us to observe which kernel objects are accessed, how frequently they are accessed, and which objects are shared across co-running workloads. Importantly, our study focuses on logical data isolation rather than hardware performance interference; hence, we design our experiments to minimize hardware-level contention (*e.g.,* not exceeding local caches or disk bandwidth).

5.1 Experimental platform

We run experiments on Cloudlab [20] c220g5 nodes (2.20 GHz Intel Xeon Silver 4114, 192GB Memory, Intel DC S3500 480 GB SSD, 1 TB 7200 RPM HDD, and 10Gbps NIC) running Ubuntu 22.04 with Linux kernel v6.1. We enable lock-related debugging configurations (*e.g.,* CONFIG_DEBUG_SPINLOCK) for collecting traces. We try to minimize interference through shared hardware: we disable SMT and isolate the LLC by enabling Intel's Cache Allocation Technology (CAT). We have 11-way LLC partitioning, and each Class of Service (COS) is assigned to one cache line. All cores on socket 0 have a one-to-one mapping with a COS.

5.2 Isolation Platforms

We run workloads on three platforms: (a) Native linux container (*lxc*), (b) gVisor release-20250421.0 (*gv*) (we use the KVM platform; recommended for bare-metal [24]), and (c) Firecracker v1.10.1 (*fc*).

These platforms represent three distinct points in the isolation design space: *lxc* provides OS-level isolation using Linux namespaces and cgroups while directly sharing the host kernel. *gv* implements a user-space kernel (sandbox) that mediates most system calls and kernel interactions, reducing direct exposure to host kernel structures. *fc* executes each workload inside a lightweight virtual machine with its own guest kernel, thereby reducing host-kernel sharing at the cost of additional virtualization layers. All platforms use standard Linux mechanisms for CPU and memory resource control where applicable (*e.g.,* cgroups), while fc relies on the VM boundary to isolate kernel state in addition to host-level resource controls. By selecting these three platforms, we capture a spectrum from direct host-kernel sharing (*lxc*), to mediated kernel interaction (*gv*), to guest-kernel virtualization (*fc*), enabling comparison of how platform design shapes kernel object sharing.

5.3 Workloads

Our goal is to understand how the *data footprint* of workloads varies across a diverse spectrum of system behavior (*e.g.,* stressing particular subsystems, light/medium/heavy kernel subsystem usage).

To capture these diverse cases, we chose workloads to measure different dimensions of kernel sharing.

Microbenchmarks. We choose a set of microbenchmarks from stress-ng [33] and Filebench [57]. We wrote our own memory microbenchmark that allocates/deallocates and touches 16GB of memory using mmap/munmap in a loop using different allocation sizes. These microbenchmarks stress different kernel subsystems to understand data footprints under targeted stress environments.

Serverless Workloads. To gain insights into short-lived object usage patterns in a FaaS environment, we analyze lightweight serverless functions from FunctionBench [32], selecting applications (image/video processing), ML training (logistic regression), and ML serving (face detection, CNN, RNN).

Cloud Workloads. To capture the data footprints of long-term and heavy workloads, we run CloudSuite cloud workloads [11] as shown in Table 5. We run all workloads in standalone mode [29] *i.e.,* all the services and components (like client and server) of the workload run in a single instance of the platform.

For microbenchmarks, we run two instances of the same workload simultaneously on the same isolation platform. For each of six FunctionBench and six Cloudsuite workloads, we pair the workload with one of five different stress-ng *stressors* listed in Table 5. This pressures different kernel subsystems for a total of 60 combinations. Filebench requires disabling ASLR, which is not supported on *gv*, so we could not evaluate it on *gv*.

For microbenchmark and serverless workload tracing, each workload is executed for a fixed duration. For cloud workloads, we terminate workloads after 30 minutes. We execute the workload separately for each lock type, tracing one lock type at a time to obtain cleaner traces and improved lock coverage. We collect traces over three iterations per lock type to maximize coverage of unique locks. We do not trace platform startup phases.

To distinguish interference sensitivity during the startup phase, which matters for bursty or short-lived workloads, from steady-state sharing that dominates long-running/persistent services, we trace in two modes: (1) *cold start* and (2) *steady state*. In *cold start* mode, tracing begins at workload launch to capture initialization effects (cache/page faults, inode/dcache population). In *steady state* mode, tracing begins after the workload has run for some time, capturing the steady-state interference pattern. For cloud workloads in *steady state* mode, we completely execute the workload once and trace in the second iteration.

5.4 Analysis Outputs

We use lock traces to determine the following:

- *Shared locks*: We identify shared locks by their lock addresses and report the median count across runs. We calculate the median per lock type (spin, mutex, semaphores, *etc.*) and sum across lock types.
- *Average lock acquire rate*: We report average lock acquire rates, $\text{Rate}_\ell = \frac{\sum_{r=1}^{N} \text{lock count}_\ell^r}{\sum_{r=1}^{N} \text{workload execution time}^r}$, where lock count is the number of times a lock ℓ (uniquely identified by lock name and lock type) was acquired, r is the iteration number and N is the total iterations. Tracing slows execution, so the rate is lower than for an uninstrumented kernel.

Together, these metrics capture the breadth and intensity of the shared data footprint for a workload. A workload may exhibit *wide and shallow* sharing – touching many kernel objects at low rates – or *narrow and deep sharing* – concentrating activity on a small number of hot objects. These patterns reveal which specific kernel objects become the structural points through which isolation breaks on each platform.

In our analysis, we use lock acquisitions as a proxy for accesses to shared kernel objects. Since kernel locks are tightly associated with specific data structures, we use the terms lock and object interchangeably when discussing sharing, with the understanding that a lock represents the kernel object whose access it serializes.

6 Object Level Interference Analysis

Our evaluation goals are:

- How does object-level sharing and hence *data isolation*, vary across isolation platforms?
- What kernel objects constitute the *shared data footprint* of a workload, and how does this footprint vary across different classes of workloads?
- Which kernel subsystems dominate object-level sharing across platforms?

Answering these questions links object-level behavior to system-level isolation, revealing how platforms shift interference and identifying sensitive workloads and critical kernel subsystems.

6.1 Platform Impact on Object Sharing

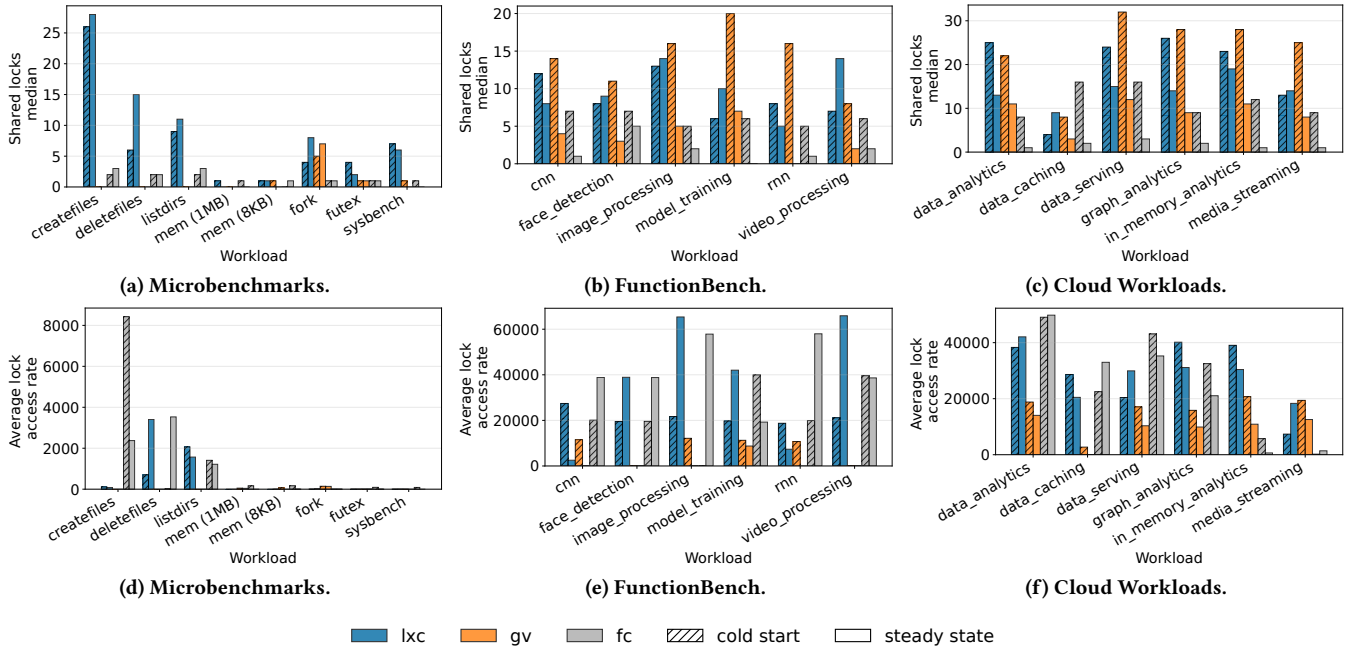
We first quantify the extent of kernel-level sharing across workloads and platforms (§6.1.1), then attribute these events to specific kernel objects driving cross-workload interference (§6.1.2).

6.1.1 ***Sharing magnitude.*** We use two metrics to characterize the sharing magnitude of different workloads on different platforms: (a) the median lock count, calculated by summing the median access count for each lock type (spinlock, mutexes, *etc.*) across runs, and (b) the average lock access rate across all shared locks. These metrics (Figure 3) are reported per platform and mode (cold start and steady state), with FunctionBench and cloud workloads aggregated over all stress-ng *stressors*. While this subsection quantifies sharing, we provide a detailed analysis of the specific kernel objects in §6.1.2.

Overall, *lxc* exhibits the broadest sharing footprint, acquiring 129 locks for microbenchmarks, 114 for FunctionBench, and 199 for cloud workloads, roughly 2.5× to 6× more unique locks than *fc*. In contrast, *fc* has the narrowest object exposure, with 21, 47, and 80 locks in the same categories, respectively. *gv* closely follows *lxc* for FunctionBench (106) and cloud workloads (197). Excluding filebench for a fair comparison of microbenchmarks (as filebench did not run on *gv*), *lxc* acquires 35 shared locks, *gv* (16), and *fc* (7), confirming that *lxc* maintains the broadest footprint while *fc* remains the narrowest. These differences reflect platform design and translate into distinct interference vectors: a broader sharing footprint increases the number of kernel objects through which co-located workloads can interact. This widespread sharing increases the likelihood of cross-workload influence and interference, making isolation guarantees weaker and harder to reason about.

lock name	description	Acquires/sec (trace points)	struct name	struct path	static global
&journal->j_state_lock	Ext4 journaling: serializes updates to the journal state and transaction lifecycle.	57982 (279)	journal_s	include/linux/jbd2.h	
&root->kernfs_rwsem	kernfs/sysfs/cgroupfs root rwsem: protects the global directory tree for kernfs operations.	18684 (8)	kernfs_root	fs/kernfs/kernfs-internal.h	
proc_subdir_lock	synchronize access to the directory structure of the /proc virtual filesystem	12067 (18)			Y
&journal->j_list_lock	Ext4 journaling: protects the lists of running/committing transactions.	10302 (156)	journal_s	include/linux/jbd2.h	
&journal->j_wait_updates	Ext4 journaling: synchronizes waiters for journal updates and transaction progress.	10208 (50)	journal_s	include/linux/jbd2.h	
aa_buffers_lock	AppArmor security module to protect a global pool of memory buffers	5953 (35)			Y
&rq->__lock	Scheduler runqueue spinlock: serializes enqueue/dequeue and scheduling decisions per CPU.	2814 (33)	rq	kernel/sched/sched.h	
&ei->i_raw_lock	ext4_inode_info: synchronizes inode-specific ext4 metadata updates.	2043 (7)	ext4_inode_info	fs/ext4/ext4.h	
tasklist_lock	Process management: global rwlock protecting task list enumeration and updates.	839 (91)			Y
&mapping->private_lock	VFS/address_space: protects private/mapping-specific data structures.	742 (165)	address_space	include/linux/fs.h	
&zone->lock	Memory allocator: per-zone lock protecting free area lists and page allocator metadata.	381 (301)	zone	/include/linux/mmzone.h	
&sbi->s_orphan_lock	Ext4 superblock: protects the list of orphaned inodes (unlink/truncate crash safety).	333 (39)	ext4_sb_info	fs/ext4/ext4.h	
&lruvec->lru_lock	Memory reclaim: protects per-memcg/node LRU lists during page reclaim/rotation.	273 (148)	lruvec	include/linux/mmzone.h	
&sbi->s_es_lock	Ext4 extent status tree: protects in-memory extent status caches for files.	250 (38)	ext4_sb_info	fs/ext4/ext4.h	

Table 6: Top shared locks across our collected traces.

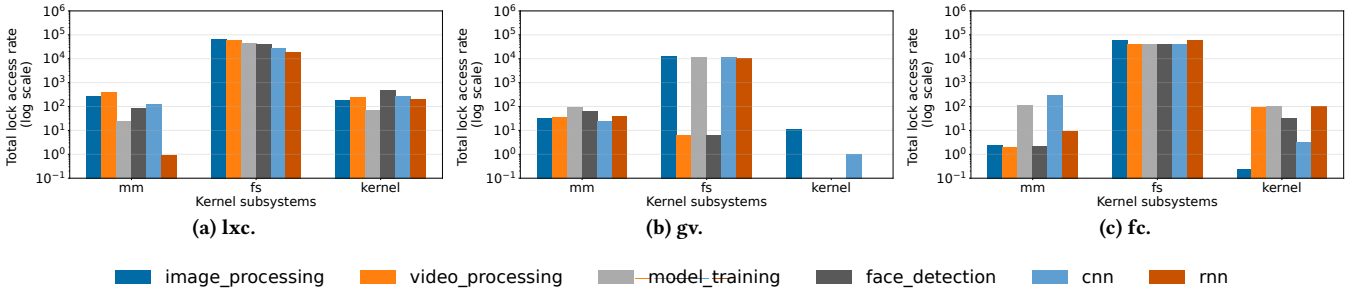


(a) Microbenchmarks. **(b) FunctionBench.** **(c) Cloud Workloads.**

(d) Microbenchmarks. **(e) FunctionBench.** **(f) Cloud Workloads.**

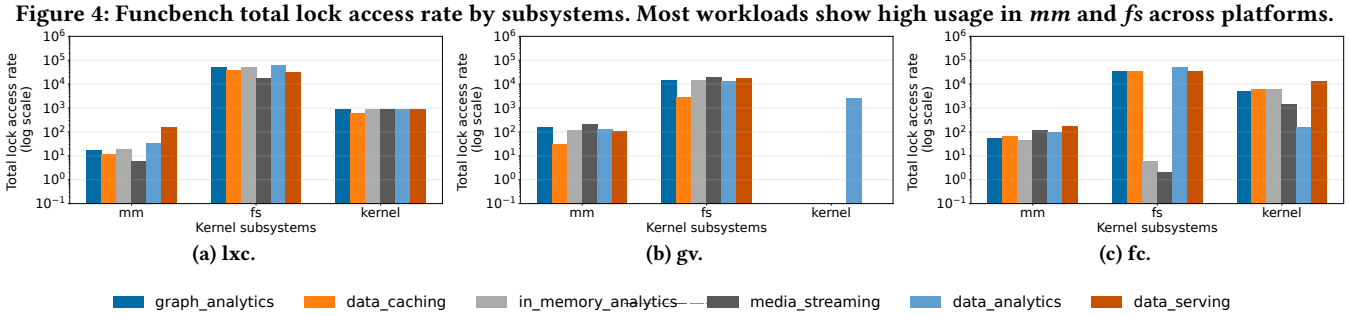
Figure 3: Figures 3a–3c show the median aggregate lock counts and figures 3d-3f show average access rates across platforms for each workload category.

Looking at the total average lock access rate for sharing intensity, a different pattern emerges. For microbenchmarks, *fc* shows the highest average lock access rate at 17.5K acquires/sec, more than double that of *lxc* at 8K. However, filebench did not run on *gv*. To enable a fair cross-platform comparison, we recompute the average rate excluding filebench workloads where *fc* has a rate of 509 acquires/sec, *gv* at 445, and *lxc* at 29. We suspect that LXC uses code paths with fewer shared objects, while the virtualization layer of *gv* and *fc* causes repeated accesses to shared locks. Overall, we see a narrow but deep sharing pattern for *fc*. For FunctionBench, *fc* again leads with the highest access rate at 391K, slightly exceeding *lxc* at 350K, with *gv* far behind at 55K. For cloud workloads, *lxc*



(a) lxc. (b) gv. (c) fc.

Figure 4: Funcbench total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms.



(a) lxc. (b) gv. (c) fc.

Figure 5: Cloud workloads total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms.

has the highest rate at 346K, followed by *fc* at 294K, while *gv* trails substantially at 152K.

This high sharing intensity of *fc* stems from its reliance on the host for core functionality. Although workloads execute inside a guest kernel, *fc* relies on (i) a KVM-based hypervisor layer and (ii) virtio-backed devices for block and network I/O, both of which ultimately depend on host kernel subsystems for memory management, scheduling, and filesystem operations. As a result, memory provisioning (via host page allocation and reclamation), virtual CPU scheduling (via host runqueues), and host-backed filesystem journaling (as virtio-blk forwards guest I/O to the host kernel) remain shared objects. While this design reduces the number of exposed host objects relative to *lxc*, it does not eliminate sharing; instead, *fc* narrows object breadth while concentrating access on a smaller set of shared host objects.

These results show that platforms differ both in how many kernel objects they share and also in how intensely they access them. While *fc* uses separate guest OSes, it still shows high sharing due to the added virtualization layers, introducing shared kernel objects that become sharing hotspots. In contrast, *gv* intercepts and handles system calls in userspace, aggregating resource acquisition in its user-mode kernel and invoking the host only for essential operations. As a result, *gv* achieves both narrower and lower-intensity sharing by shifting kernel functionality into its user-mode kernel, verifying its design goal of reduced kernel reliance.

Moreover, we observe a higher lock count and access rate across most workload–platform combinations in *cold start* mode (with some exceptions), reflecting initialization overhead. In comparison, *steady state* mode shows lower lock count and rate for some workloads, although some workloads maintain high rates, indicating persistent sharing of kernel objects (*e.g.,* file-metadata operations). Both *lxc* and *fc* exhibit high rates, mostly in *cold start* mode, with *fc* at times higher than *lxc*. This behavior highlights how platform design influences initialization behavior: additional virtualization layers introduce extra kernel interactions for setting up virtual devices, memory mappings, and filesystem metadata, increasing synchronization pressure.

Summary: Isolation platforms differ in both the *breadth* of shared kernel objects and the *intensity* with which those objects are accessed. *lxc* shows the widest sharing footprint, making their interference surface broad. *fc* reduces the number of exposed objects but concentrates activity on a smaller set of objects, producing a narrow and deep footprint. *gv* has the lowest access rates overall and comparatively lower lock counts for some workloads. *cold start* amplifies these effects, highlighting that isolation concerns extend beyond performance latency to kernel-level sharing. Overall, these results show that isolation is not a binary property of a platform: platform design reshapes object-level sharing in different ways, exposing trade-offs that are hidden by coarse abstractions such as VM vs. container.

6.1.2 ***Highly shared objects and type of sharing.*** In practice, most objects are accessed only infrequently, but a few recur across workloads and runtimes, forming persistent sharing points. Table 6 shows the number of *trace points*, the number of distinct workload–stressor–platform–mode combinations (out of 2,785 total executions; microbenchmarks exclude stressors) in which a given lock (uniquely identified by its name, type pair (*e.g.,* `tasklist_lock`, read_rw)) is observed at least once. Higher trace points indicate broader sharing and greater significance. The most significant recurring sharing falls into three categories.

1. Resource-driven serialization. These data structures manage physical resources that require strict serialization. In *memory management*, `struct zone` (via `zone->lock`, a dynamic per-object lock, which protects per-zone free area lists in the memory allocator) is extremely frequent (with 301 trace points).

In memory management, `struct lruvec` protected by `lruvec->lru_lock` (a dynamic per-object lock) appears in 148

trace points. The `lruvec` structure organizes pages into LRU lists that are central to global page replacement, page reclamation, and memory cgroup accounting. While `lruvec` structures can be partitioned by memory cgroups, page reclamation and page cache management require global coordination across workloads to maintain a consistent view of memory pressure. As a result, even workloads with small memory footprints contend for the same `lruvec` objects during reclamation or file I/O, making them a persistent source of cross-workload interference.

2. Filesystem and I/O serialization. These objects manage the filesystem and the integrity of persistent state. `struct dentry`, protected by `dentry->d_lock` (a fine-grained lock with 83 trace points), caches directory entries for accelerated pathname resolution. Because it maintains a shared view of filesystem namespace state, operations on different files may still converge on shared dentry structures and hash tables, making it a frequent synchronization point across workloads.

Similarly, journaling structures such as `struct journal_s` via `journal->j_state_lock` (279 trace points) and `journal->j_list_lock` (156 trace points) are prominent with high lock rates across workloads and platforms. The journal ensures crash consistency and ordering of the filesystem by coordinating metadata modification before disk commit. Since the filesystem maintains a single, consistent state for the underlying physical block device, these locks serialize metadata changes across workloads, making them dominant synchronization points.

Notably, we observe substantial filesystem synchronization under *fc* as well. Although *fc* executes workloads inside a microVM with a separate guest kernel, storage I/O is exposed through virtio-blk devices backed by files on the host filesystem. While the guest has its own virtual filesystem, its persistent metadata updates are eventually serialized by the host's journaling structures. This design reduces direct kernel sharing but does not eliminate shared metadata paths, explaining why filesystem-related locks remain highly active even under VM-based isolation.

3. Global serialization. These objects maintain the global visibility of system state, creating sharing between logically disjoint workloads. `inode_hash_lock` synchronizes the *global inode hash* table, to efficiently manage and access inodes, using a coarser global lock in the VFS layer. This represents incidental sharing: workloads accessing entirely different filesystems or isolated mount namespaces can still collide because the kernel maintains a single global hash table to index all inodes. Similarly, `tasklist_lock` (157 trace points), a static global `rwlock` in *process management*, serializes access to the *global system task list*, a coarse sharing point.

Other observations. We summarize the top-accessed objects (by acquires/sec) in Table 6. Cold start largely accesses the same set of highly shared objects as steady state, but with different acquisition rates. We also see that fine-grained and coarse-grained locks both contribute to sharing, though in different ways. Fine-grained locks such as `bgl->locks[i].lock` (per-block group) associated with `ext4_sb_info` protect specific filesystem metadata regions; although individually less intense, they create partitioned sharing surfaces tied to particular resource subsets. In contrast, global objects such as the global system task list (`tasklist_lock`) or the global page directory (`pgd_lock`) serialize a broad portion of kernel activity. These patterns show that interference stems both from frequent contention on fine-grained objects and from the breadth of coarse global objects—and across platforms, both persist.

Summary: A recurring set of shared objects, including both fine-grained per-object locks and coarse global locks, dominates sharing across workload-platform combinations. These objects persist across isolation boundaries because they protect shared host functionalities such as memory allocation, filesystem consistency, and global kernel state. Workloads in *cold start* amplify kernel-level sharing across platforms, while in *steady state* many reduce but do not eliminate such sharing. This concentration suggests that mitigation should target recurring hot objects rather than broad system-wide redesigns. While some pressure can be reduced at the workload level (*e.g.,* disabling journaling for short-lived serverless tasks), stronger isolation requires privatizing or partitioning frequently contended kernel structures [25]. In particular, further partitioning allocator and journaling data structures could meaningfully reduce cross-workload interference.

6.2 Workloads' Sensitivity to Object Sharing Across Platforms

We now analyze how different classes of workloads impact the isolation of a platform design. We define *sensitivity* as the extent to which a workload's sharing profile varies across platforms in terms of breadth (how many shared objects), intensity (how frequently they are accessed), and composition (which kernel subsystems and data structures are involved).

6.2.1 ***Microbenchmarks.*** Across microbenchmarks as shown in Figures 3a and 3d, we observe that most workloads acquire relatively few locks in the *steady state*, with sharing dominated by initialization effects. Memory microbenchmarks are lean on locking: they show almost no acquisitions in *steady state* but spikes during *cold start*. After initialization, they operate largely within an established working set, and therefore show minimal sustained synchronization. During cold start, however, they converge on global memory-management such as `zone` (via `zone->lock`, which protects per-zone free area lists in the page allocator) and `lruvec` (via `lruvec->lru_lock`, which organizes pages into lists for reclamation and memory pressure accounting). On *lxc*, allocation paths interact directly with the host, exposing the page allocator and reclamation state throughout execution. While on *fc*, steady-state allocation is mostly confined to the guest, but cold start amplifies sharing due to VM setup and host-backed memory provisioning. *gv* lies in between, mediating allocation in userspace while still relying on host memory-management paths for faults and shared regions.

File-metadata operations (createfiles, deletefiles, listdirs) are the most lock-intensive across platforms. This behavior is dominated by filesystems structures such as `journal_s` (via `journal->j_state_lock` and `journal->j_list_lock`), `ext4_sb_info` (storing filesystem-specific state), `super_block`, *etc.* We also observe high acquisition rates even on *fc* for `journal_s` during *steady state*. Although *fc* runs workloads inside a microVM, storage I/O is ultimately handled by the host through virtio-based devices backed by host filesystem. As a result, filesystem metadata updates from the guest are translated into host-side operations that still interact with journaling state. This shows that filesystem hierarchy is difficult to

isolate, even in microVMs; metadata operations remain a universal contention point regardless of the isolation boundary.

Process creation (*fork*) lies between memory and filesystem workloads in sharing sensitivity: it acquires more locks than memory workloads but at lower rates than file-metadata workloads, indicating coarse yet infrequent sharing. The objects involved differ by platform. Under *lxc*, *fork* interacts directly with host memory and scheduler structures, including page-cache objects such as `address_space` (represents the page-cache mapping of a file and tracks cached pages associated with an inode) and `list_lru_node`, as well as global task state via `tasklist_lock`, reflecting tight coupling to host memory-management and VFS subsystems. Under *gv*, *fork* primarily surfaces allocator-related objects such as `zone` and `swap_cgroup_ctrl` (tracks memory usage for swap control groups), since memory provisioning and scheduling still rely on host resources despite sandbox mediation. In contrast, under *fc*, most process management occurs within the guest kernel, avoiding host task-table updates, but *fork* still manifests through memory provisioning (`lruvec`) and scheduler structures such as runqueues (which serialize access to the CPU's per-core runqueue via `rq->__lock`), especially during cold start. Thus, process creation introduces short-lived but unavoidable shared-state interactions across all platforms, making bursts of task creation structurally sensitive phases for isolation.

Finally, synchronization (futex) and compute (sysbench) benchmarks show the narrowest sharing footprint, though platform designs shift the point of contention. *fc* concentrates sharing on scheduler structures like `rq->__lock` to manage virtual CPUs, while *lxc* exposes a broader range of host VFS and memory objects. In contrast, *gv* limits interaction primarily to host-side memory allocators via the `zone->lock`, as its sandbox layer shields the host from most other activity.

6.2.2 ***Serverless Workloads.*** FunctionBench workloads (Figures 3b and 3e) reveal distinct sharing patterns driven primarily by memory-management and filesystem structures. During *cold start*, all platforms exhibit elevated acquisition rates, with sharing consistently concentrated in memory-management and filesystem subsystems. Across workloads, the dominant objects are `zone`, journaling structures such as `journal_s`, and page-cache related objects such as `address_space`; model training, video processing, and CNN stress allocator and journaling paths more heavily, while RNN shows a narrower but still allocator-driven spike. Platform design shapes how this sharing manifests: *lxc* exposes a broad mix of host VFS and memory objects due to direct interaction with page-cache and filesystem metadata; *fc* narrows object diversity but concentrates peak acquisition on `zone`, `journal_s`, and scheduler structures such as runqueues during VM setup and virtio-backed I/O; and *gv* presents a broader but generally lower-intensity profile, as memory faults and file accesses still converge on host `zone` and journaling paths despite user-space mediation.

In *steady state*, workloads separate into two groups. Image and video processing maintain comparatively higher sustained sharing across platforms due to repeated file I/O and memory access, keeping `journal_s` and `zone` active. In contrast, model training and RNN stabilize after initialization and become more compute-bound, with sharing largely confined to allocator structures such as `zone` and occasional `lruvec` activity. CNN and face detection occupy a middle ground, showing moderate sustained allocator and page-cache interaction. Platform differences persist in the steady state. *lxc* continues to expose a broader set of VFS and memory objects, reflecting direct host-kernel interaction. *fc* maintains a narrow-and-deep profile, concentrating sharing on allocator, journaling, and scheduler objects. *gv* spreads interaction across a moderate set of objects but generally at lower acquisition intensity.

Unlike microbenchmarks, serverless workloads spend much of their lifetime in cold-start and bursty execution phases [59], making transient sharing patterns central to isolation. Isolation is therefore shaped less by steady-state averages and more by how platform design amplifies or dampens short-lived object-level sharing. *lxc* exposes broad and consistently intense sharing, *gv* maintains lower-intensity behavior, while *fc* concentrates sharing into high-intensity cold-start bursts, creating focused interference despite stronger steady-state isolation. This highlights a trade-off between efficiency and isolation: aggressive cold starts improve utilization but increase object-level sharing during initialization.

6.2.3 ***Cloud Workloads.*** Cloud workloads shown in Figures 3c and 3f diverge most across platforms, reflecting how each platform handles sustained, resource-heavy execution. In *steady state*, *lxc* consistently shows both high lock counts and high acquisition rates across data analytics, data serving, graph analytics, and in-memory analytics, showing a broad and intense exposure to the host kernel, leaving these workloads highly vulnerable to object sharing from co-located workloads. *fc*, in contrast, maintains a narrower footprint with fewer locks but continues to exert high pressure on specific hot objects, notably in data analytics and in data serving. While it achieves negligible sharing with in-memory analytics and media streaming, its narrow but deep sharing profile in data-heavy tasks suggests that isolation is concentrated rather than eliminated. *gv* occupies an intermediate position: it has moderate lock coverage but has lower rates, except in media streaming surpassing *fc*.

As we previously observed for serverless workloads, this divergence is also visible in the diversity of kernel objects, *lxc* using 43 unique locks, *gv* 25, and *fc* 15 across modes. *lxc* spans sharing across diverse subsystems: `uts_sem` (protects uts namespace), `proc_subdir_lock` (protects procfs task subdirectory lists), and `swap_lock` (protects global swap subsystem state), reflecting exposure to broad kernel resources. *gv* shows virtualization-specific locks like `free_vmap_area_lock` (protects vmap area allocation lists) and `folio_wait_table[i]` (protects folio waitqueues), while *fc* adds fewer unique locks but includes important shared locks like `journal->j_wait_commit`.

For cloud workloads, isolation is driven by sustained interaction with shared kernel services rather than short-lived phases, making steady-state sharing the primary concern for long-running co-location. Platform design determines whether this pressure is broadly distributed across many kernel objects, as in *lxc*, or concentrated on a small set of hot objects, as in *fc*, with *gv* occupying a middle ground by spreading access while keeping intensity lower. Hence, for cloud workloads, isolation is less about avoiding transient spikes and more about controlling long-term exposure to shared kernel state through platform design.

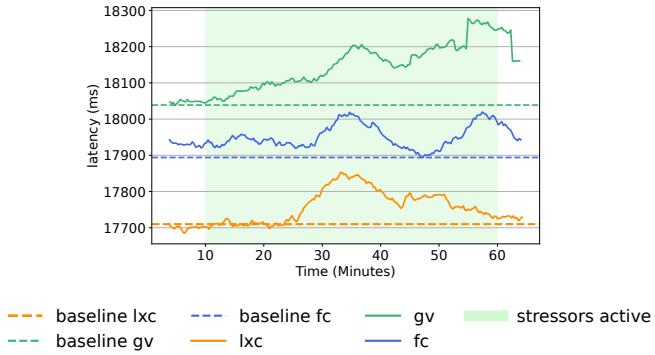


Figure 6: Performance impact on model training when run with varying stress-ng *stressors*.

Summary: Across all workload categories, we find that the same workload can vary significantly in object usage across platforms, shown in both lock counts and acquisition rates. This implies that a workload's sensitivity to objects depends strongly on the platform it runs on. Platform design alters not only the intensity of object usage but also the type of sharing: *lxc* exposes the widest variety of objects, *gv* introduces new synchronization points in memory and filesystems, and *fc* narrows diversity but concentrates pressure on hotspots like journaling and memory. Thus, sharing is platform-specific, and selecting the right platform for a workload directly shapes its object-level interference profile.

6.3 Dominant Kernel Subsystems

Kernel object usage is spread widely across the Linux codebase. We plot the total of maximum average access rate (summed over the maximum rate for unique locks across modes for a given workload) for FunctionBench and cloud workloads in Figures 4 and 5, respectively. Our results show that across platforms, most acquisitions are concentrated across *mm* and *fs* followed by *kernel*. Improving workload isolation, therefore, requires rethinking how memory and filesystem state are shared, rather than simply reducing exposure to less frequently exercised kernel subsystems.

6.4 Performance Isolation

To understand the impact of object sharing on performance interference, we run each FunctionBench workload for 70 minutes, launching a different stress-ng *stressor* every 10 minutes, starting at 10 minutes in sequence: *getdent* (10-20), *fstat* (20-30), *io* (30-40), *mem* (40-50), and *fork* (50-60). Figure 6 shows model training, where latency degrades as stressors are introduced. *lxc* and *gv* experience the largest slowdown. *fc* is more resilient, but still has some performance dips. These results highlight that the degree of performance sensitivity is coupled to the breadth and focus of lock sharing exposed by each platform. We make similar observations for other FunctionBench workloads.

Summary: Object-level sharing translates into measurable performance interference under co-location, with severity depending on platform design. Accounting for hot shared kernel objects, such as memory-management or filesystem-journaling structures, in placement decisions could improve performance predictability and reduce cross-workload interference.

7 Related Work

We now discuss prior work in the context of our contributions.

Kernel Locks. Multiple tools have been proposed to detect race conditions and deadlocks. Eraser [54] proposes a novel lockset algorithm for dynamically detecting data races in multithreaded programs. LockDoc [38] performs dynamic trace-based kernel analysis to infer locking rules for data structures and identify locking violations. Unlike these tools, which focus on correctness and bug detection, we use locking behavior as a proxy to quantify cross-workload object interference and architectural isolation.

Scalability and Commutativity. Min *et al.* [42] identified kernel objects that limit filesystem performance and scalability. Clements *et al.* [10] proposed the scalable commutativity rule; interface operations that are order-independent can be implemented to allow scalability. Multikernel [5] proposes a new architecture for scaling multicore systems by avoiding any inter-core sharing. While these works focus on scalability, we examine how modern isolation mechanisms expose shared kernel objects across diverse workloads.

Interference and Isolation Measurement. KIT [37] detects inter-container functional interference by comparing the system call traces of a container across two different executions (running with and without another container). Some works have studied and measured performance interference for workloads by either characterizing workloads [8, 18] under different stress levels or by proposing an analytical model [60]. OSmosis [2] provides a formal framework for reasoning about resource sharing across isolation abstractions. In contrast, our work analyzes kernel object sharing directly, exposing structural interference rooted in shared kernel state.

Address Space Isolation. Address-space isolation (ASI) [12, 13] restricts access to sensitive kernel memory by unmapping it from the current execution context. Implementing ASI requires marking memory as sensitive and non-sensitive. A possible approach to identifying sensitive memory is to detect critical data structures and unmap their memory when not needed. Our analysis can serve as an initial step toward detecting these sensitive data structures.

Lightweight Virtualization. While prior work [3] analyzes host-kernel code coverage and performance across containers, gVisor, and Firecracker, we quantify the object-level sharing and interference that arise across these isolation boundaries.

8 Conclusions

We introduced and evaluated a new approach to understanding data isolation via shared kernel objects by analyzing kernel-level lock activity. Across workloads and isolation platforms, we found significant variation in object access patterns, with the file system and memory management subsystems emerging as the most frequent sources of sharing.

Acknowledgments

We thank the anonymous reviewers and Bijan Tabatabai for their time and valuable feedback on this paper. We also thank Nathan Ridge and the clangd team for their help with clangd-client. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF grant CNS-1763810.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 419–434.

[2] Sidhartha Agrawal, Shaurya Patel, Arya Stevinson, Linh Pham, Ilias Karimalis, Hugo Lefeuvre, Aastha Mehta, Reto Achermann, and Margo I. Seltzer. 2025. Comparing Isolation Mechanisms with OSmosis. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems (PLOS '25)*. 1–9. doi:10.1145/3764860.3768325

[3] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending containers and virtual machines: a study of firecracker and gVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. 101–113. doi:10.1145/3381052.3381315

[4] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. 375–388. doi:10.1145/2046707.2046752

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 29–44. doi:10.1145/1629575.1629579

[6] Paolo Bonzini. 2023. An introduction to lockless algorithms. https://lwn.net/Articles/844224/.

[7] Gianluca Borello. 2017. Container isolation gone wrong. https://sysdig.com/blog/container-isolation-gone-wrong/.

[8] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 107–120. doi:10.1145/3297858.3304005

[9] clangd 2024. clangd language server. https://github.com/clangd/clangd.

[10] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 1–17. doi:10.1145/2517349.2522712

[11] cloudsuite 2025. Datacenter Benchmarking with CloudSuite 4.0. https://www.cloudsuite.ch/.

[12] Jonathan Corbet. 2022. A call to reconsider address-space isolation. https://lwn.net/Articles/909469/.

[13] Jonathan Corbet. 2022. Generalized address-space isolation. https://lwn.net/Articles/886494/.

[14] Jonathan Corbet. 2024. Lockless algorithms for mere mortals. https://lwn.net/Articles/827180/.

[15] Christina Delimitrou and Christos Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 23–33. doi:10.1109/IISWC.2013.6704667

[16] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 77–88. doi:10.1145/2451116.2451125

[17] Christina Delimitrou and Christos Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 473–488.

[18] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 599–613. doi:10.1145/3037697.3037703

[19] Docker. 2026. Empowering App Development for Developers: Docker. https://www.docker.com/.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1–14.

[21] Brendan Gregg. 2019. BPF Performance Tools: Linux System and Application Observability (Chapter 2). Addison-Wesley Professional.

[22] Brendan Gregg. 2024. The Return of the Frame Pointers. https://www.brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html.

[23] gvisor 2024. gVisor. https://gvisor.dev/.

[24] gvisor-platform 2025. Platform Guide. https://gvisor.dev/docs/architecture_guide/platforms/.

[25] Hang Huang, Honglei Wang, Jia Rao, Song Wu, Hao Fan, Chen Yu, Hai Jin, Kun Suo, and Lisong Pan. 2024. vKernel: Enhancing Container Isolation via Private Code and Data. *IEEE Trans. Comput.* 73, 7 (2024), 1711–1723. doi:10.1109/TC.2024.3383988

[26] hyperv-containers 2026. Hyper-V Containers. https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container.

[27] iproute2 project. [n. d.]. tc(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[28] Harshad Kasture and Daniel Sanchez. 2014. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 729–742. doi:10.1145/2541940.2541944

[29] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. doi:10.1109/IISWC.2016.7581261

[30] kata 2024. The speed of containers, the security of VMs. https://katacontainers.io/.

[31] Michael Kerrisk. [n. d.]. cgroups(7) — Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[32] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. doi:10.1109/CLOUD.2019.00091

[33] Colin Ian King. 2025. stress-ng - a tool to load and stress a computer system. https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html.

[34] M. Krohn and E. Tromer. 2009. Noninterference for a Practical DIFC-Based Operating System. In *2009 30th IEEE Symposium on Security and Privacy*. 61–76. doi:10.1109/SP.2009.23

[35] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. 2008. Online Estimation of Architectural Vulnerability Factor for Soft Errors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. 341–352. doi:10.1109/ISCA.2008.9

[36] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.

[37] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. 427–441.

[38] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. 2019. LockDoc: Trace-Based Analysis of Locking in the Linux Kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Article 11, 15 pages. doi:10.1145/3302424.3303948

[39] lsp 2024. Language Server Protocol. https://microsoft.github.io/language-server-protocol/.

[40] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 218–233. doi:10.1145/3132747.3132763

[41] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *SIGOPS Oper. Syst. Rev.* 54, 1 (Aug. 2020), 47–63. doi:10.1145/3421473.3421481

[42] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. 71–85.

[43] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. 29.

[44] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. [n. d.]. Noninterference for Operating System Kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs (CPP'12)*. 126–142. doi:10.1007/978-3-642-35308-6_12

[45] nabla 2024. Nabla containers: a new approach to container isolation. https://nabla-containers.github.io/.

[46] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 219–230.

[47] Open Container Initiative. 2026. Open Container Initiative Runtime Specification. https://github.com/opencontainers/runtime-spec.

[48] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2022. Using Trāṭr to tame Adversarial Synchronization. In *31st USENIX Security Symposium (USENIX Security 22)*. 3897–3916.

[49] perf 2024. perf: Linux profiling with performance counters. https://perfwiki.github.io/main/.

[50] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the library OS from the top down. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 291–304. doi:10.1145/1961295.1950399

[51] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. Locksmith: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems* 33 (January 2011), 3:1–3:55.

[52] Prathyush PV. 2019. kLockStat: An eBPF Tool To Monitor Linux Kernel Lock Contentions. https://prathyushpv.github.io/2019/04/30/kLockStat.html.

[53] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. 57–68. doi:10.1145/2000064.2000073

[54] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (nov 1997), 391–411. doi:10.1145/265924.265927

[55] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[56] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 121–135. doi:10.1145/3297858.3304016

[57] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *Usenix ;login* 41, 1 (2016). https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[58] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. Article 9, 14 pages. doi:10.1145/2592798.2592812

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 559–572. doi:10.1145/3445814.3446714

[60] Scott Votke, Seyyed Ahmad Javadi, and Anshul Gandhi. 2017. Modeling and Analysis of Performance Under Interference in the Cloud. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 232–243. doi:10.1109/MASCOTS.2017.11

[61] Wikipedia. 2024. Non-blocking algorithm. https://en.wikipedia.org/wiki/Non-blocking_algorithm.

[62] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2023. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. 771–788. doi:10.1145/3582016.3582033