Locked In, Leaked Out: Measuring Isolation via Kernel Locks

Anjali
University of Wisconsin-Madison
anjali@wisc.edu

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

Abstract

Isolation is a critical property for shared infrastructure to limit exposure and interference among simultaneous running workloads. Cloud providers use different isolation mechanisms such as full Virtual Machines, microVMs, Linux containers, secure containers *etc.* to confine workloads running in a multi-tenant environment.

We propose a novel way to understand and measure performance interference and isolation at the system software layer that occurs due to shared access to data structures. We observe that interference takes place through shared structures, such as a kernel-level data structure, and that operating systems must synchronize access to these structures for safety. By measuring the level of synchronization between workloads, we can measure their ability to interfere and thus the amount of isolation the platform provides.

We demonstrate our method for measuring isolation by measuring the accesses to locks acquired in common across multiple workloads which indicates the amount of sharing through kernel data structures and hence the interference/isolation between two workloads. Furthermore, we identify the isolation properties of different kernel structures under different workloads, and find that the file system journal and kernel page allocator are the most common sources of interference.

1 Introduction

Isolation is essential in cloud environments where mutually distrusting tenants share physical hardware. Cloud providers use various isolation mechanisms to provide isolation between co-located workloads while maintaining performance guarantees. However, tightening the isolation boundary by running a workload in a virtual machine (VM) compared to running in a container can lower the overall resource utilization of a host. It is difficult to gather information about the amount of isolation a workload needs to minimize sharing with different co-located workloads. When sharing is less, there is less interference; hence, workloads are more isolated.

Lightweight isolation platforms like Linux Containers, gVisor, and Firecracker rely on the host operating system for various resources and functionalities. This dependency leads to sharing and the potential for interference, which may have a detrimental impact on workloads by making them more prone to security attacks and performance degradation. Therefore, to reduce interference, it is important to understand what is being shared at the shared platform layer, *i.e.,* host kernel.

Two common sources of interference through OS resources are: (1) resource allocation, and (2) shared access to objects. The former arises when a workload needs a resource but is not able to due to unavailability as other co-running workloads use the resource. This type of interference is usually solved by leveraging OS (*e.g.,* cgroups, qdisc) and hardware isolation mechanisms (*e.g.,* cache partitioning) to partition resources to eliminate/reduce interference [14, 20, 21, 25, 40, 46]. Resource-partitioning does not eliminate interference due to shared access to an object. Interference via access to objects happens when concurrent workloads want to access the same shared resource, such as a data structure within the host kernel.

Our work tackles interference from shared accesses to data structures. We choose to analyze interference at the software stack because configuring underlying hardware for sharing and isolation (*e.g.,* controlling the DRAM bandwidth or applying cache-partitioning schemes) is mostly the same across platforms. The software stack is where the implementation of these platforms differ, *e.g.,* a userspace kernel in gVisor, and these software implementation choices impact how shared accesses vary across platforms. Also, we believe much research has already looked at sharing and isolation at the hardware layer [31, 39, 46]. We are unaware of any work that has studied interference through shared access to objects. We note that past research [16, 33] has analyzed kernel lock accesses for scalability. Our focus in this work is on studying interference through kernel locks.

We show in Figure 5 the importance of interference via shared access to objects across different isolation platforms by stressing the filesystem for metadata operations [13]. We run conflicting workloads for some periods and observe that such interference can lead to performance degradation. This type of interference can also lead to security vulnerability, leading to framing attacks [37] or denial of service.

In this work, we do a comparative study to understand sharing and interference observed at the system software level via shared objects when concurrent workloads execute in different isolation platforms. Furthermore, we use the data on sharing and interference to understand isolation among concurrent workloads via shared kernel data structures. Our focus is on the *isolation of system structure*, meaning how well isolated the data structures of system software executing

on behalf of a workload 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 [10, 28, 29, 35].

The first challenge in analyzing isolation at the system software layer is to recognize what resources/data structures/objects are shared among concurrent workloads. We observe that limited isolation implies that workloads can interfere, so we look at the opportunities for and frequency of interference between workloads on a platform. For example, two workloads accessing a shared cache in the OS can affect each other's behavior by modifying the cache contents. The key insight of our work is that most forms of interference take place on some shared data, and that access to shared data requires synchronization. As a result, we can identify points of interference by the prevalence of synchronization in workloads. Workloads that rarely synchronize share little data, have little impact on each other, and are well isolated. In contrast, workloads that synchronize frequently (*e.g.,* locking access to shared data) have more opportunity to interfere by changing shared data or stalling waiting for locks; hence they are less isolated. One side effect of this approach is that it recognizes that isolation is workload dependent: two workloads executing purely user-mode code without making system calls may be perfectly isolated in operating system processes, while workloads that make heavy use of system services and multiple processes, accessing more shared data, benefit from more powerful isolation mechanisms.

We make the following contributions in this work:

- We collect and analyze kernel-level lock traces to measure system-level sharing via objects across a diverse set of applications and isolation platforms.
- We identify memory allocation and file system journaling as the dominant sources of cross-application interference.
- We implement a tool that enables fine-grained detection of software-level interference via dynamic kernel tracing.

2 Isolation and Interference Through Sharing

Isolation in computer systems keeps workloads apart from each other, so that the behavior of one workload does not affect other workloads. The goal for perfect isolation is *non-interference* [27, 36], meaning that one workload has no impact on what another workload experiences. While often used for security by looking at what is possible for an attacker, for isolation we look more practically at the performance impact of one specific workload on another specific workload. Thus, an imperfect isolation mechanism can provide perfect non-interference if the two workloads do not make use of facilities that allow interference.

Platform / Resource	**KVM**	**runc**	**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.

The basic approach that isolation systems take is to separate workloads in space and time. Spatial isolation ensures workloads access different data, while temporal isolation prevents them from accessing the same data simultaneously. Much as an operating system process provides separate address spaces and collections of resources, stronger forms of isolation provide increasingly more private resources and fewer shared resources. At the extreme, running workload on separate physical machines provides 100% private resources and no shared resources. This approach also dictates that resource efficiency is usually opposed to isolation: as a platform increases isolation, it decreases the amount of sharing across workloads that provides the opportunity for interference. Thus, increasing isolation decreases resource efficiency, which provides a strong motivation to find the *right level of isolation* for a workload — using a too-strong platform wastes resources. On the other hand, if you increase efficiency by sharing more, you are increasing your risks for performance and security isolation. Table 1 shows the resources that are shared/isolated across different isolation mechanisms.

To reduce interference cloud providers use various isolation mechanisms such as full VMs, containers, microVMs *etc.* All this mechanism takes the approach of reducing the interactions with the shared platform *i.e.,* the host kernel across workloads to minimize interference and sharing.

Building on this insight, we propose to measure the isolation of workloads on a platform by looking at the level of synchronization between the workloads. If there is little synchronization, then there is little access to shared data, and the workloads are well isolated. In contrast, if there is frequent, fine-grained synchronization, then there is poor isolation, as there are many opportunities. Thus, by analyzing the usage of locks in common across workloads in an isolation platform, we can measure the amount of isolation offered by the platform. We can also quantitatively compare the isolation offered by different platforms by looking at differences in synchronization behavior between platforms.

3 Synchronization as an Identifier of Sharing

Interference and synchronization. Within a software platform, interference implies that some shared resource

```c
void set_fs_pwd(
  struct fs_struct *fs,
  struct path *path){
  struct path old_pwd;
  path_get(path);
  spin_lock(&fs->lock);
  // sets fs->pwd to *path
  spin_unlock(&fs->lock);
}
```

Listing 1. Private

```c
void release_task(
struct task_struct *p){
  write_lock(&tasklist_lock);
  // removes task from list
  write_unlock(&tasklist_lock);
}
```

Listing 2. Shared

```c
void insert_inode_hash(
  struct inode *i, u64 hval){
  struct hlist_head *b =
  inode_hashtbl + hash(i->i_sb,hval);
  spin_lock(&inode_hash_lock);
  hlst_add_head_rcu(&i->i_hash,b);
  spin_unlock(&inode_hash_lock);
}
```

Listing 3. Incidentally Shared

Table 2. Example lock usage in the Linux kernel.

between the two workloads acts as the agent for 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). Thus, any interference between workloads will be controlled by operating system synchronization. Some examples include two processes sharing memory must synchronize around the physical pages being shared. Likewise, two processes sharing access to a device, such as a sound or a network card, must synchronize their access to the device. And, two processes sharing access to a file must synchronize access to the file system. Read sharing does not lead to interference, as each process can logically operate on a copy of data (*e.g.*, in a local processor cache) without any effect from other processes.

Furthermore, workloads that only incidentally access shared operating system data structures must still synchronize. Two programs accessing a shared file system, even if they access different files, may synchronize access to shared caches such as the Linux `dcache` or `inode` cache.

Locks and Synchronization. We note that shared resources in the kernel need synchronization to ensure a safe and correct order of execution of concurrent processes in the system. There are various primitives implemented in the Linux kernel to achieve synchronization:

- Lock-based synchronization: These include spinlocks (and variants such as reader-writer and seqlocks), mutexes, and semaphores.
- Lockless synchronization: These include atomic operations, Read-Copy Update (RCU), and memory barriers.

Given the non-blocking nature of lockless mechanisms [12, 19, 45], they lead to less interference and contention compared to lock-based approaches. Locks enforce mutual exclusion and hence more contention, so their impact on interference and isolation will be more.

Table 2 shows examples (code snippets) of private, shared, and incidentally shared locks in the Linux kernel. The `set_fs_pwd` function acquires a *private* lock, `fs->lock`, to protect the `fs->pwd` field of the `fs_struct` data structure that maintains the file system state for a process. This lock is not

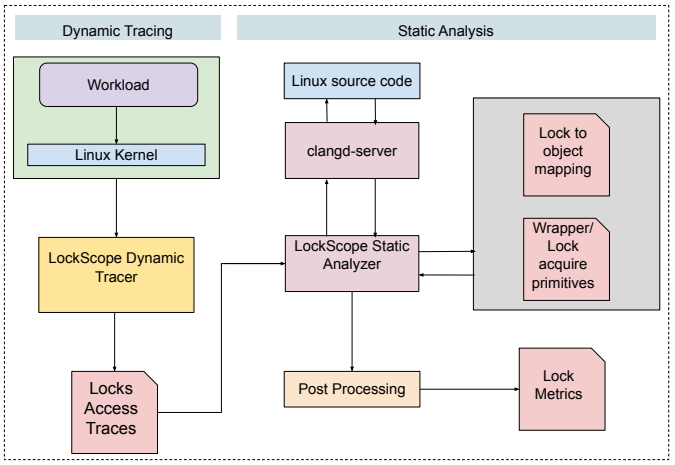


Figure 1. Overview of *LockScope* workflow. In the dynamic tracing phase, LockScope collects lock accesses across workloads, and the static analysis phase completes the lock-to-object mapping.

accessed by other processes. `tasklist_lock` is *shared* and synchronizes access to a resource shared by multiple processes, the list of all tasks. In the example code, it is used to remove a task from the list when the process exits. Finally, processes may *incidentally* synchronize by hashing into the same hash bucket, a form of false sharing, although the protected data is not shared. `insert_inode_hash` acquires `inode_hash_lock` that protects the shared `inode_hashtable`.

We use lock-based synchronization as a proxy for sharing. While this will not cover all types of sharing, we believe that our work is a first step in understanding and quantifying sharing via synchronization, and can be extended to other synchronization primitives in the future.

4 Lock Tracing Implementation

We implemented a data-collection tool named *LockScope* to measure the lock usage of workloads running on isolation platforms on Linux. The goal of LockScope is to trace lock acquisitions dynamically and use static analysis to identify location in code where a lock acquisition in the trace occurs

Lock Type	Lock Primitives
Spinlock	_raw_spin_lock, _raw_spin_lock_irqsave, _raw_spin_lock_irq, _raw_spin_lock_bh, _raw_spin_trylock, _raw_spin_trylock_bh,_raw_spin_lock_nested, _raw_spin_lock_irqsave_nested,
Read-write lock	_raw_read_lock, _raw_write_lock, _raw_read_lock_bh, _raw_write_lock_bh, _raw_read_lock_irq, _raw_write_lock_irq, _raw_read_lock_irqsave, _raw_write_lock_irqsave, _raw_read_trylock, _raw_write_trylock, _raw_write_lock_nested
Mutex	mutex_lock_nested/mutex_lock, rt_mutex_lock_nested/rt_mutex_lock, mutex_trylock, rt_mutex_trylock, mutex_lock_interruptible_nested
Semaphore	down_read, down_write, down_read_trylock, down_write_trylock, down_read_nested, down_write_nested, down_read_killable, down_write_killable, down_read_killable_nested, down_write_killable_nested, down_read_interruptible

Table 3. Lock primitives probed by *LockScope* dynamic tracer.

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

Table 4. An example row in the lock to object mapping generated using symbol information.

and, more importantly, what objects the lock protects. We show an overview of *LockScope* in Figure 1. *LockScope* consists of two components: (a) dynamic tracer and (b) static analyzer. *LockScope* captures and analyzes fine and coarse-grained runtime locking activity, enabling the precise identification of object-level interference across platforms.

4.1 LockScope Dynamic Tracer

We capture the dynamic lock usage using a modified and extended version of *klockstat*, an eBPF lock monitoring tool [38]. This tool traces lock acquire and release functions in the Linux kernel and can be attached to mutexes, RT-mutexes, semaphores, spinlocks, and reader-writer spinlocks. Table 3 lists all the lock primitives we probe with klockstat.

The *LockScope* dynamic tracer produces a trace of every lock acquired, and for each operation, records: the process and thread IDs, lock address, lock name, 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 different points using interference metrics such as the number of *shared locks* — locks held in common across workloads — accessed for different workloads and *private locks* — unique and non-shared locks held by each workload.

In summary, 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. We find the shared locks using the unique lock addresses common to workloads.

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

4.2 LockScope Static Analyzer

The *LockScope* static analyzer resolves a stack trace from the dynamic tracer into the exact location in code where the lock was acquired and, when the locks is a member of an object, the object containing the lock. For global locks, we resolve the lock name and its definition. We implemented the *LockScope* static analyzer using the clangd [15] language server [4]. clangd is widely used in many IDEs for different functionalities (*e.g.,* symbol indexing and outgoing call analysis), and it provides a convenient API for querying code properties. We use clangd for static analysis as it offers simple mechanism to accurately index into a large codebase, including macro resolution, type deduction, and cross-referenced navigation, providing functionality closer to a compiler-driven source code analysis compared to other available tools such as cscope[1].

The stack traces from the eBPF tools are sometimes incomplete *i.e.,* the trace stops 2-3 levels due to inline functions and certain optimizations such as tail call optimization [23, 24]. We use the static analysis tool to resolve the stack traces to the exact line numbers of lock acquisitions in the code. We use the lock names, file names, and function names to determine this. Furthermore, we use static analysis to identify the data structure a lock protects. We do this to learn about the nature of sharing of different data structures and also to understand which data is private vs. mostly shared.

LockScope static analyzer is a command-line utility tool and supports the following functionality:

- *Symbol Generation* produces all the details related to a symbol, including name, type, signature, and line number ranges within the source code where the symbol is defined. We use this to create a mapping of locks to the `struct` within which they are defined. Table 4 provides an example row from this file. Moreover, we use the symbol information to generate a list of system-wide global locks (≈2488).
- *Outgoing Calls* uses function-type symbol information to identify outgoing calls. We use this to resolve the point of acquisition for locks in cases of incomplete dynamic stacks, where the start function is the last function recorded in the collected stack.
- *Incoming Calls* works similar to *Outgoing Calls*. We leverage this information to identify potential lock wrapper functions (335 identified so far), which serve as

stop conditions when resolving the point of acquisition in incomplete dynamic stacks.

- *AST* generates an abstract syntax tree for a source file, which *LockScope* uses to resolve lock to object mapping, where lock names result in many-to-many mapping in the file generated using document symbol (*e.g.,* lock name with *lock* can be associated with multiple objects). We call these locks *generic*.

  We use the AST of the function where *generic* locks are acquired to get 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.
- *Get Definition* returns the file path and line number where the lock is defined. We use this to locate the corresponding object name by matching the line number to the range associated with a struct-type symbol from *Symbol Generation*.

For all unique locks, identified by lock name, function name, and file, acquired across all workloads analyzed in this paper (574 in total), we can map approximately 75% of them to their associated objects. Our tool currently does not resolve cases where locks are acquired through macros or where object variables are reassigned to local variables within function bodies. We manually added support for a small number of such macros (about 5) that we encountered during our analysis.

5 Locking Analysis Methodology

Our evaluation goals are:

- What are the different locks and data structures accessed by a workload?
- What shared data is accessed most frequently?
- How does sharing via objects vary across platforms for the same workload?
- How can we quantify interference and isolation via shared accesses to objects using kernel locks?

5.1 Platform and Workloads

We run all our experiments on Cloudlab [22] c220g5 machine, with twenty cores Intel Xeon Silver 4114 running at 2.20 GHz, 192GB ECC DDR4-2666 Memory, two disks: Intel DC S3500 480 GB 6G SATA SSD and 1 TB 7200 RPM 6G SAS HDs, and 10Gbps NIC. We run on Ubuntu 22.04 (kernel v6.1).

For each workload under test, we collect host kernel lock traces to understand how different isolation platforms use various kernel data structures for functionalities. We also run stress tests to analyze the impact of using shared kernel data structures on performance due to synchronization.

For all tests, we turn off SMT to minimize interference through resource contention and cross-socket communication. We also isolate the LLC by enabling Intel's Cache Allocation Technology (CAT). In our setup, we have 11 LLC way partitioning, and each Class of Service (COS) (1-11) is assigned to one cache line. All cores (on socket 0) have a one-to-one mapping with COS. Our focus is to understand interference caused due to shared access to kernel resources. Hence, our experiments are designed to measure interference via system structures and avoid hardware interference (*e.g.,* not exceeding local caches or disk bandwidth). We use the default kernel configuration for stress tests but enable lock-related debugging configurations (*e.g.,* CONFIG_DEBUG_SPINLOCK, CONFIG_DEBUG_MUTEXES) for collecting traces.

We run workloads on four platforms: (a) host, (b) LXC (runc), (c) gVisor release-20231106.0 (runsc), and (d) Firecracker v1.10.1 (fc). For runsc, we use the KVM platform, which is recommended for bare-metal [9]. Our goal is to understand object usage patterns across a diverse spectrum of system behavior (*e.g.,* stressing particular subsystems, light/medium/heavy kernel subsystem usage). To capture these diverse cases, we run our evaluation on three different sets of workloads across these platforms:

Microbenchmarks. We choose a set of microbenchmarks, stressing different kernel subsystems to understand object usage under targeted stress environments.

Serverless Workloads. As serverless workloads are a major use case for lightweight isolation, we select applications (image and video processing), ML training (logistic regression), and ML serving (face detection, CNN, and RNN) workloads from FunctionBench suite [26] to analyze object usage across modern lightweight serverless functions. This is useful for gaining insights into short-lived object usage patterns in a FaaS environment.

Cloud Workloads. To capture object usage in heavy and long-running workloads, we run cloud workloads as shown in Table 7. Feedsim and VideoTranscode are taken from DCPerf [6]. We were not able to set up the analytics and data caching from DCPerf on gVisor due to its very restricted environment, so we replaced them with the Graph Analytics and Data Caching workloads from CloudSuite [5].

5.2 Lock Usage Tracing

We trace workloads using the *LockScope* dynamic tracer and two simultaneous identical workloads on the same isolation platform. Each workload executes for a fixed duration. We re-execute the workloads once for each lock type described in Table 3 to get cleaner traces (*e.g.,* minimizing nesting) and better lock coverage. We collect the traces in three iterations to maximize the coverage of unique locks. We do not trace the platform startup.

We collect system-wide traces while the workload is running and then select entries belonging to the running workload. For firecracker, we select the traces belonging to the firecracker process. For gVisor, we use the sandbox and gofer pids to select the traces. For runc, we leverage the

namespace filtering [2] from eBPF to select the traces belonging to each container namespace and then further filter it to remove docker and runc-related management activities. Because locks acquired in interrupt context (*e.g.,* timer interrupts, softirqs) may not be caused by the current process, we remove these lock accesses. We inspect the stack track of lock acquires and remove locks whose traces include interrupt-related functions (*e.g.,* __softirqentry_text_start, hrtimer_interrupt).

We use the LockScope static analyzer to process the lock traces and complete the mapping of lock names to kernel objects. For resolving the point of acquisition for some incomplete dynamic traces, we use the outgoing calls functionality of the static analyzer and then do a breadth-first search from the last function in the stack traces. We use the lock primitives and the lock name to stop the search. We also leverage lock acquire function wrappers generated by the tool to stop the search in some cases.

We use lock traces to determine the following:

- *Shared and private locks:* We identify unique shared and private locks by their lock addresses and report the average count across runs.
- *Locks access rate:* We report lock access rates. For each lock (identified by name, function, file, and type), we compute the rate by dividing the acquire count by execution time. We then sum these per-lock rates across runs to obtain a cumulative access rate, which helps quantify how frequently different locks are accessed.
- *Lock access across kernel subsystems:* We group the lock rates by kernel subsystem (*e.g.,* `mm`, `fs`, `kernel`) based on file paths to understand how lock usage varies across different parts of the kernel.

5.3 Performance Interference

We measure the performance of workloads with a *trasher* designed to stress kernel resources usage through frequent system calls. For each performance test (unless specified otherwise), we first start a workload (worker), and launch an additional trasher every ten minutes, causing increasing interference. For resources that allow interference, this leads to decreasing performance as trashers are added. We pin the worker and trashers to different CPU cores sharing the same socket.

6 Measuring Lock Usage

We begin by measuring the lock usage of microbenchmarks and application workloads to identify opportunities for interference. We run two copies of the same workload simultaneously and use LockScope to record the locks each copy acquires, and of those which are shared across the two instances and which are private to an instance. We show the average shared lock counts and the cumulative lock rate for all workloads traced in Table 5. For lock rate, we aggregate over all rates by calculating the sum.

Workloads	Host	runc	runsc	fc
	shared (rate)			
mem-8KB	0.33 (226)	0.33 (0.0016)	1 (73)	2.67 (399)
mem-1Mb	0.33 (610)	2.33 (0.19)	1 (220)	1.66 (121)
file-list	2.66 (321)	9.67 (897)		1.66 (1336)
file-create	4 (29)	15.67 (1740)		1.67 (2367)
file-delete	3.66 (353)	3.67 (296)		1 (3880)
image processing	5.33 (1799)	4.33 (1012)	0.67 (12)	0.67 (3.751)
video processing	7.33 (3488)	1.99 (1950)	0.67 (0.14)	1.33 (4.3)
lr_training	12.66 (15)	2 (17)	1.67 (1.64)	0.66 (0.86)
face_detection	6.67 (80)	2.67 (12)	1.67 (0.27)	0.33 (0.01)
cnn	3.33 (4.06)	2.67 (239)	1.67 (13)	0.33 (1.85)
rnn	2.33 (1.8)	1.67 (2.82)	1.34 (0.59)	1 (9.23)
graph analytics		8.33 (382)	5 (419)	1.33 (59)
data caching		2.33 (0.008)	0.33 (0.0011)	1.33 (26)
data caching (no warmup)		1.33 (0.018)	0.33 (0.0005)	1.33 (1.41)
feedsim		7.67 (63)	3.33 (43)	1 (0.62)
video transcode		15 (82)	8 (3.06)	3.66 (62)

Table 5. Shared average lock count and cumulative lock access rate (in parentheses) for each workload and platform.

6.1 Microbenchmarks

We use microbenchmarks targeting a particular OS resource to measure interference for those resources. For memory, we stress page allocation, and for files we stress metadata operations. We do not include CPU-bound workloads, as they typically make little use of OS resources and our experiments show little locking or interference. We do not measure lock usage in the networking subsystem, all our experiments are designed to minimize or eliminate networking access paths.

6.1.1 Memory

To stress the memory subsystem, we use a memory microbenchmark that allocates a total of 16GB memory with different `mmap` allocation sizes. The microbenchmark touches one word of each allocated page by writing to it and finally,

calls munmap to free the memory. The benchmark runs in a loop, continuously allocating and deallocating memory for a specified time duration. We allocate a total of 18GB to each running instance to avoid interference due to resource allocation and pin each workload to a separate core. We run the benchmark for two allocation sizes: 8K and 1MB.

We observe that although shared lock counts for platforms remain low, two shared locks stand out. The global buddy allocation locks, zone->lock, is accessed by host, runsc, and fc at a high rate, the highest rate for host (≈610 for 1MB), runsc (≈220 for 1MB), and fc (≈205 for 8KB). zone->lock protects a zone object, which holds the free list used for physical page allocations. This global lock can be the point of interference under high memory pressure, resulting in performance degradation for these platforms.

Firecracker also acquires lruvec->lru_lock with a high rate (about 193) for 8KB allocation size. The lruvec structure is used for maintaining pages in LRU order for a combination of memory zone and memory cgroups to select pages for reclamation. The high acquisition rate may be due to frequent updates to the LRU lists during the initial phases of page allocation, where newly allocated pages need to be inserted into the appropriate LRU list. This suggests that even early-stage memory usage in Firecracker involves significant interaction with reclamation-related data structures.

In contrast, runc exhibits minimal shared lock usage. This behavioral difference, compared particularly to the host arises because the host workload instance runs without memory limits, while the runc container is constrained via cgroups with an upper memory limit. The memory limit being pre-allocated in runc avoids triggering memory pressure and bypasses code paths involving the global allocator.

6.1.2 File Metadata Operations

We measure file system metadata operations to understand object-level interference as they are more prone to performance degradation due to software-level interference [33].

We use Filebench [42] to evaluate file metadata operations, and report the operations per second (ops) for each benchmark. Filebench requires address space layout randomization (ASLR) to be disabled. Currently, gVisor does not support disabling ASLR, so we were unable to get Filebench to run under gVisor and exclude it from this benchmark. Each workload instance operates on separate directories.

List Directory. This benchmark looks for contention around directory entries and inodes. It lists the contents of a large directory (50,000 entries).

Create Files. This measures the performance of file creation operations by creating 50,000 files (4KB) in a directory.

Delete Files. It uses a single thread to measure file deletion operation in a directory with 50,000 files

We observe a high rate of shared locks across operations, highest in runc, followed by host, and fc. We observe a high access rate for locks related to the filesystem bookkeeping and some memory management.

The primary shared locks are associated with the file system journal_s structure and the superblock ext4_sb_info. While host and runc access both, fc only accesses the journal. These are single structures shared by an entire file system volume, so that even if workloads access different files, they will still access the same superblock and journal. journal->j_list_lock, which protects the per-transaction list of modified buffers in the journaling layer, serializing access to the buffer, has the highest rate for the host (≈ 304 for list). bgl->locks[i].lock is a per-block-group spinlock acquired for any updates to a group's block and inode bitmaps is another commonly accessed lock between host and runc. It has a lower access rate on both due to a fine-grained locking mechanism (acquired per block), but can still be a significant interference point under load. This result shows that fine-grained locking mechanisms are also prone to such interference, specifically under load, which can negatively impact performance.

Uniquely, runc acquires aa_buffers_lock with a high rate (highest being ≈660 for create) for all operations that protect AppArmor's buffer pool used for security metadata, ensuring safe allocation and reuse of buffers. The global inode_hash_lock accessed by runc (≈333 for create) and protects access to the global inode hash table. This result emphasizes that despite operating on different inodes, processes can still contend for this lock due to incidental sharing, highlighting how such shared structures can become significant points of interference. fc, on the other hand, mostly acquires lruvec->lru_lock apart from journal->j_list_lock, but the access rate is low.

Both host and runc acquire multiple shared locks due to their high reliance on a shared host filesystem, leading to potential interference and performance overhead during metadata operations. High sharing of locks such as journal->j_list_lock and inode_hash_lock can result in low performance isolation.

Firecracker has an isolated filesystem, which runs most operations inside the guest, and helps minimize lock usage. However, it still accesses some host kernel objects like journal_s when writing back to the disk image, which creates potential channels for interference at the host level.

6.2 Application Workloads

The preceding results look at single-resource microbenchmarks. We now consider applications in two categories: serverless workloads and some popular cloud workloads.

6.2.1 Serverless Workloads

We run application and ML-related serverless workloads adopted from the FunctionBench suite [26] and vHive [43].

Category	Name	Subsystem loads			Description	Input	Output
		CPU	Memory	Disk			
Application	image processing	medium	medium	low	Image transformation using different effects (PIllow)	image	image
	video processing	high	high	medium	Applies gray scale effect (OpenCV)	video	video
ML training	logistic regression	high	high	medium	Review analysis and training (logistic regression, scikit-learn)	text	model
ML Serving	face detection	medium	medium	low	Annotates face in a video (CascadeClassifier, OpenCV)	video	video
	cnn	medium	medium	low	Image classification (SqueezeNet, Tensorflow, CNN)	image	JSON
	rnn	low	low	low	Words generation (PyTorch, RNN)	JSON	JSON

Table 6. Serverless functions adopted from FunctionBench and vHive.

We briefly describe each benchmark in Table 6. We modified these workflows to read inputs and write output locally instead of S3. Hence, the network load is low, and disk I/O may be higher as we read/write data locally.

Shared lock count and access rate. As observed in Table 5, host has the highest number of shared lock counts, followed by runc, runsc, and fc for all workloads. The shared lock rate for fc and runsc is lower compared to host and runc for most workloads (fc has the highest rate for RNN, and runsc has higher rate than host for CNN), suggesting that these more isolated environments experience less frequent interference on shared locks for these workloads.

Host and runc have the same highest accessed shared locks (`j_state_lock` and `journal_s`) for image processing and video processing. RNN shows the lowest access rate for host, while runc has the lowest rate for face detection.

For most workloads, `zone->lock` for memory pages is one of the highly accessed locks by fc and runsc. `j_state_lock` is also acquired by runsc and fc for model training. As expected, we do not observe any networking lock. This indicates that journaling and memory management are potential sources of system-level interference for these serverless workloads, even in stronger isolation platforms like runsc and fc.

Kernel Subsystem. As shown in Figure 2, most shared locks across platforms are concentrated in the *mm*, and *fs* subsystems. *fs* usage dominates on the host, particularly for model training and video processing workloads. There is considerable diversity in subsystem usage, suggesting deep reliance on kernel services. CNN and RNN exhibit the lowest usage of *fs* locks. runc shows a similar trend with sharing across multiple subsystems, with the highest activity in *fs* and *mm*. In contrast, runsc acquires fewer shared locks, with activity concentrated in *fs* and *mm*. fc exhibits the least sharing, with shared locks confined to either *fs* or *mm*.

There is a wide variation in how all these workloads use kernel objects based on the platform they are using and the amount of stress they put on different subsystems, indicating the impact of design choices in kernel object usage. The same workload behaves differently in terms of kernel usages, signifying the importance of the *right* platform choice for a given workload.

Benchmarks	Libraries/ SW
Graph Analytics	Apache Spark
Data Caching	Memcached
Feedsim (Object Aggregation, Ranking/Inference)	Oldisim Library, ZLIB, Boost, OpenSSL, BZIP2, LZ4, Snappy, libevent, jemalloc, lzma, libsodium, rsocket, fmt, FBThrift, Folly, wangle, fizz
VideoTranscode (Video Processing)	ffmpeg, svt-av1, libaom

Table 7. Cloud Workloads from DCPerf and CloudSuite.

Moreover, we note that serverless workloads are typically stateless, so frequent writing of file data is surprising may be due to using legacy code in a serverless environment. This also suggests that file systems for serverless workloads do not need crash consistency, so journaling could be disabled.

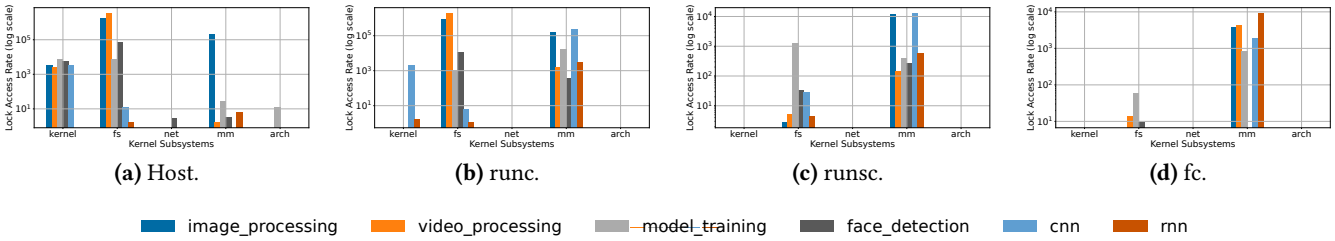
6.2.2 Cloud Workloads

We use LockScope to trace four cloud workloads across different application domains, as shown in Table 7. We run two concurrent workload instances, each with 24GB of memory and pinned to four separate cores on the same socket.

We run all workloads in standalone mode *i.e.,* all the services and components (like client and server) of the workload run in a single instance of the platform. While Feedsim and VideoTranscode do not have a separate server/client setup, we modified the other two workloads to run in standalone mode. Because the workload depends on virtual networking, we cannot run them directly on the host and only report results for runc, runsc, and fc.

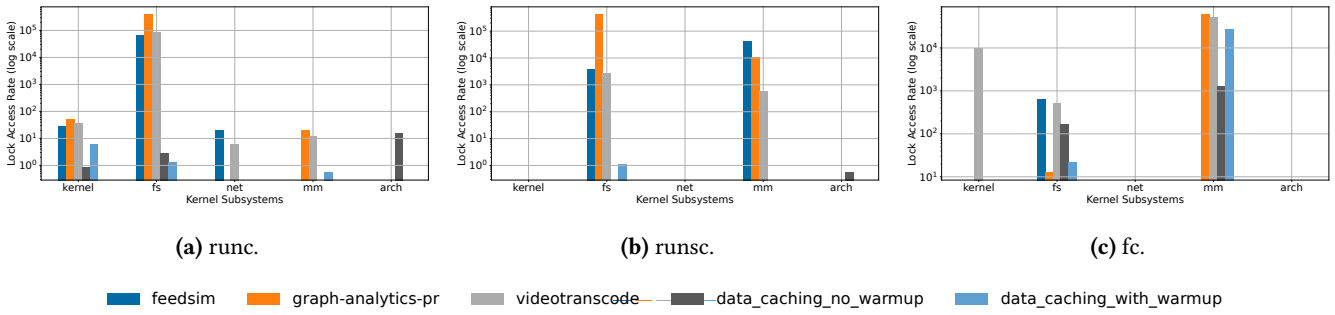
Graph Analytics uses the Spark framework to perform graph analytics on large-scale datasets. We run the PageRank algorithm for three iterations on a Twitter dataset with Spark's driver and executor memory set to 8GB each.

Data Caching runs a Memcached server simulating a Twitter data caching workload using a 10GB dataset. We trace this setup in two modes: (a) with server warmup, and (b) without



(a) Host. **(b)** runc. **(c)** runsc. **(d)** fc.

Figure 2. Serverless workloads shared locks access rate across kernel subsystems. *mm* and *fs* are highly accessed across platforms.



(a) runc. **(b)** runsc. **(c)** fc.

Figure 3. Cloud workloads shared lock access rate across kernel subsystems. Most workloads show high usage in *mm* and *fs* across platforms.

server warmup (tracing is skipped for warmup phase).

Feedsim [7] represents the aggregation and ranking workloads in recommendation systems. It searches for the maximum QPS that the system can achieve while keeping p95 latency to be no greater than 500ms.

VideoTranscodeBench [8] is based on ffmpeg representing video encoding workloads. It can apply different encoders and videos, and run them at various encoding levels.

Shared lock count and access rate. runc and runsc have the highest shared locks for VideoTranscode. For all other workloads, runc has the highest number of shared locks, while fc has the least, except for data caching (in both modes).

runc has the highest rate for feedsim and videotranscode, with most locks belonging to the filesystem. The top two locks are `root->kernfs_rwsem` (protects `kernfs_root` responsible for maintaining the root of the kernel virtual file system) and `journal->j_state_lock`. The journal lock remains the top accessed for graph analytics as well. Notably, it does not exhibit any significant access to data caching.

Similar to runc, runsc also grabs mostly filesystem (`journal->j_state_lock`, `bgl->locks[i].lock` and `root->kernfs_rwsem` with a higher rate) locks and `zone->lock` for most workloads. Data caching has minimal sharing for runsc.

`lruvec->lru_lock` is the highest source of interference for fc based on rate for all workloads but videotranscode. It uses a handful of filesystem locks for videotranscode.

These results further indicate variation in kernel object accesses by different workloads based on the platform they use, varying their isolation level via system structures.

Although there are shared lock accesses by these workloads, but the rate with which they are accessed is lower compared to microbenchmarks and serverless for most, indicating low resource pressure.

Kernel Subsystem. We make a similar observation compared to serverless workloads: filesystems, and memory management subsystems remain a significant source of system-level interference.

6.3 Lock Interference Summary

These results demonstrate a high variance in shared lock access, predicting variation in the interference of co-located workloads. They also show the data structure likely to cause interference, namely the page allocator and reclamation mechanism, and the file system journal. While much of the file system uses fine-grained locking, such as the inode and dentry caches, the journal is a single point of contention.

We also note that both coarse and fine-grained sharing can contribute to object-level interference, which can lead to performance overhead during high accesses under load.

7 Performance Interference

The preceding section looked at the level of shared locking across workloads. In this section we evaluate the performance interference through system resources for the same

workloads, showing that the level of shared locking relates to the amount of interference.

7.1 Microbenchmarks

We run the same microbenchmarks stressing different kernel subsystems from Section 6 to understand how sharing different kernel objects impacts the performance of co-running workloads. As before, we do not evaluate CPU-only workloads.

7.1.1 Memory

For this benchmark, we run one worker and seven trashers (Section 5.3). The worker runs for 80 minutes, and the first trasher starts after ten minutes, and every ten minutes, an additional trasher starts. We run the benchmark three times and report the average performance. Between each run, we reset the environment.

We observe the effects of object-level interference on host, runc, and runsc for both allocation sizes in Figure 4. gVisor has two levels of virtual-to-physical page mappings [3], one from the application to Sentry and the other from Sentry to the host. This leads to a lower baseline performance compared to *host* and runc. As noted, runsc and host acquire the global allocator lock `zone->lock` with a high rate, which becomes a point of high interference for both under memory load, as indicated by their performance degradation. As the load increases in the system, we can see the impact of such sharing on runc from its degraded performance.

Firecracker is not impacted by other co-running microVMs over time as the total mmap size remains consistent across iterations. After the initial boot, where it maps the necessary pages into guest memory, fc does not need to make additional `mmap` calls to the host to allocate more physical memory for subsequent allocations. The guest OS within the microVM allocates and deallocates within the already mapped region for subsequent allocations. However, it does has have a higher execution time after startup, which comes from a high rate of access to `zone->lock` and `lruvec->lru_lock` due to initial page allocations.

7.1.2 Filesystem Metadata Operations

For all metadata benchmarks, we observe performance degradation for host and runc with fc having the lowest baseline for create and delete metadata operations as shown in Figure 5. We calculate the baseline by running a single instance for each for ten minutes and averaging over the iterations completed during that time. fc has stable performance for these operations under stress, with some minor degradation at times. Filesystem locks acquired during metadata operations showed a very high access rate from our lock usage analysis, and we can see the impact of those locks' interference (journaling locks in particular) on the degraded performance on all three platforms. Even incidentally shared locks like `inode_hash_lock` acquired by runc for metadata operations contribute to significant object interference under pressure, resulting in performance degradation.

host and runc are impacted more as they share more of the host kernel's filesystem internal state compared to fc, where isolation at the microVM level limits such interference between instances.

7.2 Serverless Workloads

We show the performance of the worker over time for all serverless workloads in Figure 6. Similar to the microbenchmarks, we calculate the baseline by running a single instance with no co-runner interference and averaging the results across multiple runs, and compare against an instance with increasing numbers of trashers.

We observe performance degradation across several workloads on all platforms. Our lock usage analysis revealed that these workloads exhibit a wide range of lock access patterns, with high acquisition rates across most workloads, particularly in the filesystem and memory subsystems.

These workloads involve complex sharing patterns, ranging from fine-grained locks such as `bgl->locks[i].lock` on ext4 blockgroups, to incidentally shared locks like `inode_hash_lock`, and several global locks (`zone->lock`, several journaling locks) acquired along multiple kernel code paths. These overlapping access patterns lead to numerous points of object-level interference within shared kernel subsystems.

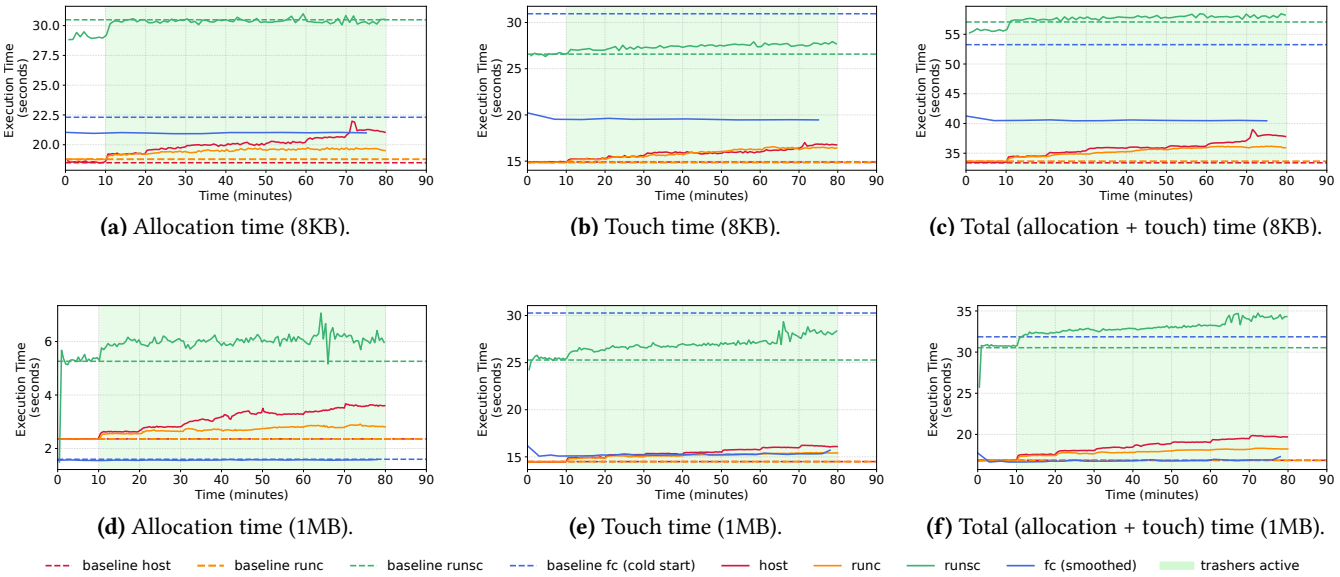
The result is compounded performance degradation, as seen in the Figure 6. Notably, no single platform consistently outperforms the others across all workloads in terms of providing isolation. The degree to which each platform isolates kernel objects, and thus avoids lock-level interference, determines which workloads benefit more from a given isolation platform.

7.3 Cloud Workloads

We run one (baseline) and two instances to measure performance interference for cloud workloads rather than using trashers. We observe minimal performance degradation (less than 1%) across these workloads. This is likely due to limited resource pressure, as only two instances are running. Our lock usage analysis supports this observation, showing relatively low lock access rates for these workloads, insufficient to cause significant interference under this low load.

7.4 Performance Interference Summary

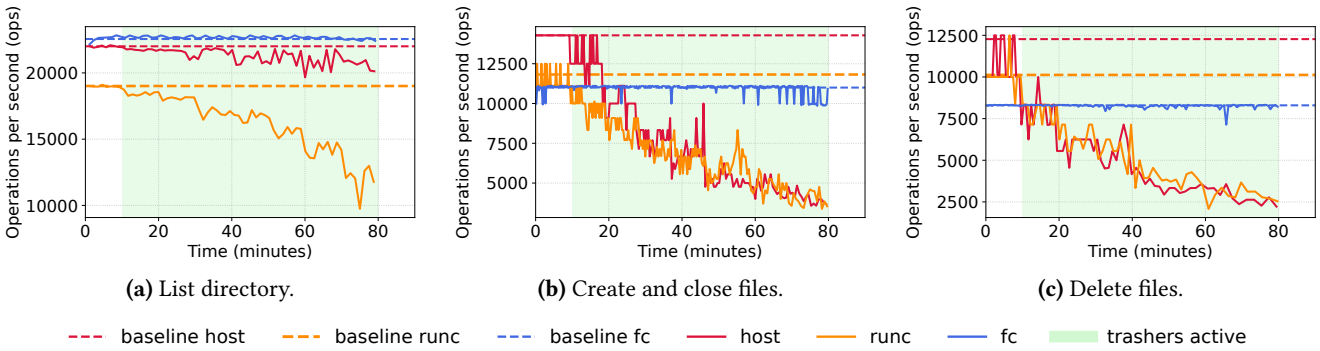
Our performance evaluation reveals that object-level sharing can significantly impact performance as access rates to shared kernel objects increase. This has important implications for workload scheduling in multi-tenant environments: co-located workloads that frequently access the same shared kernel objects can interfere with one another, with the extent of interference varying based on the isolation platform in use.



(a) Allocation time (8KB). **(b)** Touch time (8KB). **(c)** Total (allocation + touch) time (8KB).

(d) Allocation time (1MB). **(e)** Touch time (1MB). **(f)** Total (allocation + touch) time (1MB).

Figure 4. Memory stress tests for 8KB (4a- 4c) and 1MB (4d- 4f) allocation sizes for a total memory of 16GB. All but fc show performance degradation over time under load. Note that y-axes do not go to zero. Lower numbers are better.



(a) List directory. **(b)** Create and close files. **(c)** Delete files.

Figure 5. Filebench stress tests for filesystem metadata operations. Both, host and runc show significant performance degradation for create/close and delete operations. fc performs uniformly under load. Note that y-axes do not go to zero. Higher numbers are better.

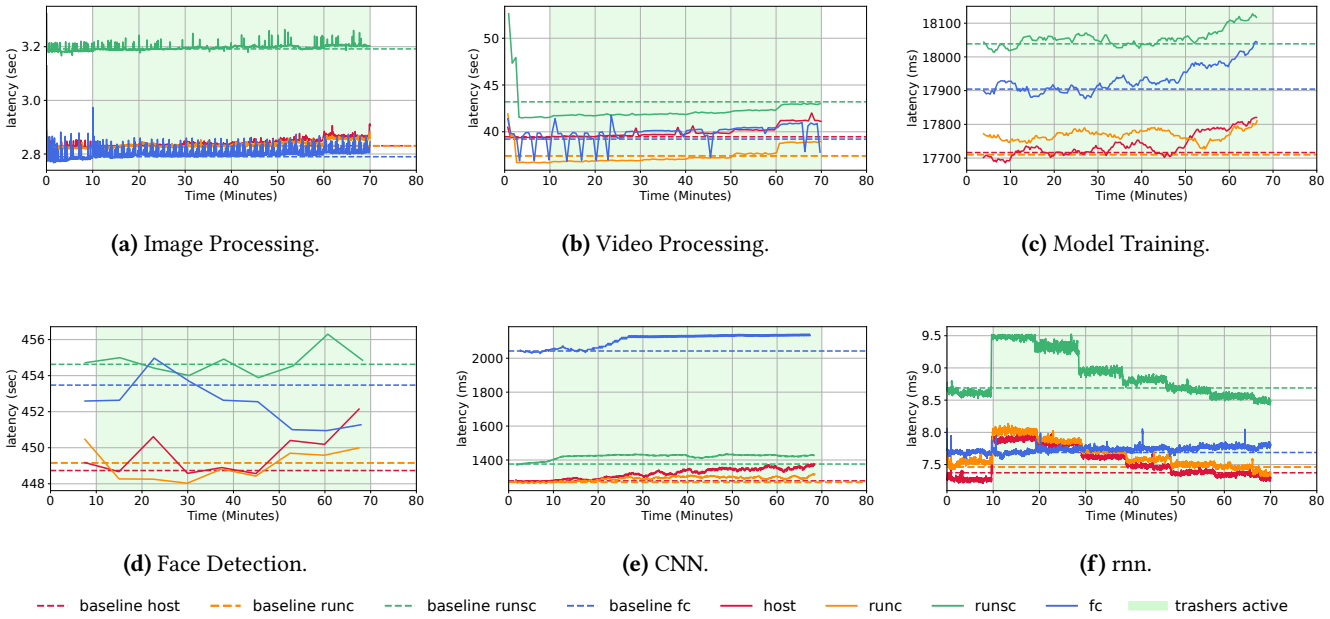
8 Interference Analysis Use-Cases

We envision multiple use cases for our analysis of kernel lock interference.

Quantify isolation via a metric. We can use the lock interference data to formulate a metric that measures isolation by measuring the amount of sharing and the impact it has on interference. This metric can be used in scheduling workloads in the cloud to minimize interference in the cloud. *e.g.,* interfering workloads mostly reading shared objects are less likely to interfere and can be safely scheduled together.

Identify the type of sharing. By identifying objects that are heavily shared vs those that are not can help us understand the nature of sharing. Sharing in the kernel varies across different subsystems. Sharing is necessary to ensure consistent access to globally visible resources, such as file data. In other cases, sharing is used for convenience to create simpler data structures. *e.g.,* the futex table is shared by all processes, but each process only accesses its private locks in the table.

Using the metric for design. By identifying which data has to be shared and which need not, kernel developers can work to reduce the amount of interference possible between workloads, and eliminate or minimize things shared only for convenience. This analysis can provide insights for application developers to optimize their implementations, minimizing data sharing where feasible. Additionally, it can guide isolation platform developers in enhancing their platforms



(a) Image Processing. **(b)** Video Processing. **(c)** Model Training.

(d) Face Detection. **(e)** CNN. **(f)** rnn.

Figure 6. FunctionBench stress tests. All platforms show some performance degradation under load for most workloads. Note that the Y-axes do not go to zero. Lower numbers are better.

by adding layers, similar to 'sentry' in gVisor, to segregate highly shared data structures.

9 Related Work

Kernel Locks Multiple tools have been proposed to detect race conditions and deadlocks. Eraser [41] proposes a novel lockset algorithm for dynamically detecting data races in multithreaded programs. "Locksets" consist of all held locks accessing shared variables. If the lockset for shared variable is empty then the variable is flagged as not being consistently protected. LockDoc [32] proposes a dynamic trace-based analysis of an instrumented kernel to infer locking rules of members of data structures to understand the implementation and to detect possible locking rule violations.

Scalability and Commutativity Min *et al.* [34] studied the scalability behavior of some popular file systems and identified some kernel objects as the source of scalability bottlenecks. Clements *et al.* [16] introduced a scalable commutativity rule, suggesting that when interface operations are performed in any order without affecting the outcome, they can be implemented in a manner that allows for scalability. Multikernel [11] proposes a new architecture for scaling multicore systems by avoiding any inter-core sharing.

Interference Measurement A recent work [30] 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 [14, 21] under different stress levels or by proposing an analytical model [44].

Address Space Isolation Address-space isolation (ASI) [17, 18] is the technique of unmapping unnecessary kernel memory, making it inaccessible to the current running context. The implementation of ASI depends on marking memory as sensitive and non-sensitive; sensitive memory is unmapped, restricting the address space for system calls. 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.

10 Conclusions

In this paper, we introduced and evaluated a new approach to understanding and measuring system-level interference. By collecting and analyzing kernel-level lock activity, we identified heavily accessed kernel objects across a range of workloads and isolation platforms. Our results reveal significant variation in object access patterns, leading to differing performance implications across platforms. The analysis also highlights the file system and memory management subsystems as the most frequently accessed components across these platforms.

References

[1] cscope, 2012. https://cscope.sourceforge.net/.

[2] bcc reference guide, 2024. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#12-bpf_get_ns_current_pid_tgid.

[3] gvisor mm, 2024. https://github.com/google/gvisor/tree/master/pkg/sentry/mm.

[4] Language server protocol, 2024. https://microsoft.github.io/language-server-protocol/.

[5] Datacenter benchmarking with cloudsuite 4.0, 2025. https://www.cloudsuite.ch/.

[6] Dcperf, 2025. https://github.com/facebookresearch/DCPerf.

[7] Feedsim, 2025. https://github.com/facebookresearch/DCPerf/blob/v1.0/packages/feedsim/README.md.

[8] Ffmpeg, 2025. https://github.com/facebookresearch/DCPerf/blob/v0.2.0/packages/video_transcode_bench/README.md/.

[9] Platform guide, 2025. https://gvisor.dev/docs/architecture_guide/platforms/.

[10] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 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, page 375–388, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

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

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

[14] Shuang Chen, Christina Delimitrou, and José F. Martínez. 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, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

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

[16] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 1–17, New York, NY, USA, 2013. Association for Computing Machinery.

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

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

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

[20] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems, 2016.

[21] Christina Delimitrou and Christos Kozyrakis. 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, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[22] 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. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[23] Brendan Gregg. Bpf performance tools: Linux system and application observability (chapter 2). Addison-Wesley Professional, 2019.

[24] Brendan Gregg. The return of the frame pointers. https://www.brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html, 2024.

[25] Harshad Kasture and Daniel Sanchez. 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, page 729–742, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[27] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *2009 30th IEEE Symposium on Security and Privacy*, pages 61–76, 2009.

[28] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 341–352, USA, 2008. IEEE Computer Society.

[29] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association.

[30] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 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, page 427–441, New York, NY, USA, 2023. Association for Computing Machinery.

[31] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. *SIGARCH Comput. Archit. News*, 43(3S):450–462, jun 2015.

[32] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[35] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 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, page 29, USA, 2003. IEEE Computer Society.

[36] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP'12, page 126–142, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift.

Using Trāṭ to tame adversarial synchronization. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3897–3916, Boston, MA, August 2022. USENIX Association.

[38] Prathyush PV. klockstat: An ebpf tool to monitor linux kernel lock contentions, August 2019. https://prathyushpv.github.io/2019/04/30/kLockStat.html.

[39] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, 2006.

[40] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News*, 39(3):57–68, jun 2011.

[41] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997.

[42] Vasily Tarasov, Erez Zadok, , and Spencer Shepler. Filebench: A flexible framework for file system benchmarking, 2016. https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[43] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 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, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Scott Votke, Seyyed Ahmad Javadi, and Anshul Gandhi. 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)*, pages 232–243, 2017.

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

[46] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 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, page 771–788, New York, NY, USA, 2023. Association for Computing Machinery.