Pushing Performance Isolation Boundaries into Application with *pBox*

Yigong Hu
Johns Hopkins University

Gongqi Huang
Johns Hopkins University

Peng Huang
University of Michigan

Abstract

Modern applications are highly concurrent with a diverse mix of activities. One activity can adversely impact the performance of other activities in an application, leading to *intra-application interference*. Providing fine-grained performance isolation is desirable. Unfortunately, the extensive performance isolation solutions today focus on mitigating coarse-grained interference among multiple applications. They cannot well address intra-app interference, because such issues are typically *not* caused by contention on hardware resources.

This paper presents an abstraction called *pBox* for developers to systematically achieve strong performance isolation within an application. Our insight is that intra-app interference involves application-level *virtual resources*, which are often invisible to the OS. We define *pBox* APIs that allow an application to inform the OS about a few general types of *state events*. Leveraging this information, we design algorithms that effectively predict imminent interference and carefully apply penalties to the noisy *pBox*es to achieve a specified isolation goal. We apply *pBox* on five large applications. We evaluate the *pBox*-enhanced applications with 16 *real-world* performance interference cases. *pBox* successfully mitigates 15 cases, with an average of 86.3% reduction of the interference.

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

Keywords: performance isolation; intra-app interference

ACM Reference Format:
Yigong Hu, Gongqi Huang, and Peng Huang. 2023. Pushing Performance Isolation Boundaries into Application with *pBox*. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613159

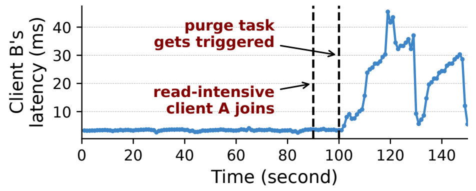

Figure 1. A real-world intra-application performance interference issue from MySQL. Details are described in Section 2.1.

1 Introduction

Applications in production demand strong *performance isolation*—the ability to maintain consistent and predictable performance despite potential sources of interference.

Extensive research [10, 11, 26, 47, 58, 64, 68, 71, 78] has focused on achieving performance isolation among multiple applications running on the same server. They broadly fall into two categories: (1) partitioning hardware resources [10, 47, 48], and (2) dynamically adjusting CPU core assignments [18, 26]. They can mitigate interference between applications because the interference is caused by direct contention on *hardware resources*, *e.g.*, a batch job overuses CPU or network and causes a slowdown to a latency-critical job.

What receives less attention is performance isolation *within* an application, which ensures that distinct activities in an application do not adversely affect each other's performance.

Providing fine-grained performance isolation is increasingly desired by users [3, 23] and developers [19, 72]. Modern applications have a high degree of concurrency with a diverse mix of activities, such as one thread to handle each request and various background tasks, making them susceptible to **intra-application interference**. For example, in processing a query from a client, a thread overuses the UNDO log defined in the application, significantly slowing down another client's requests (Figure 1). Such issues lead to unpredictable performance and poor user experience. They cannot be well addressed by adjusting hardware resources like CPU cores. Indeed, they can occur when hardware resources are sufficient.

The lack of principled solutions for fine-grained performance isolation forces developers to rely on *ad-hoc* code, such as splitting data structures, inserting timeouts, and tuning concurrency levels, which is not only difficult and time-consuming to implement, but also ineffective. Intra-app interference is often triggered by complex interactions among activities, which are difficult to anticipate during coding. There are also many program points that can suffer from interference, so it is almost infeasible to insert isolation code everywhere.

An alternative strategy is using resource quotas. The *resource container* OS abstraction [7] facilitates accurate accounting of resources consumed by an application activity, *e.g.*, functions associated with handling a request. Linux control group [53] supports thread-level resource control. However, production applications exhibit fluctuating resource usage, making it difficult to decide on a suitable quota.

To address the current gaps, this paper proposes an OS abstraction called *pBox* that allows developers to systematically and conveniently achieve performance isolation within an application. *pBox* does *not* enforce resource quotas. Instead, it focuses on the ultimate objective of reducing interference. Developers add *pBox* creation code in the application activity boundaries and specify a high-level isolation goal. At runtime, the kernel monitors if any *pBox*'s isolation goal is in danger of being violated, and reacts to satisfy the goal.

The design of *pBox* is informed by our observation that intra-app interference involves application-level *virtual resources*, such as shared buffers, queues, tickets, and logs. In contrast to hardware resources directly managed by the OS, virtual resources are usually invisible to the OS and exhibit diverse representations. Moreover, applying resource reallocation, the common approach to mitigating interference, poses challenges in this context. Reallocating virtual resources at the system level is non-trivial and can cause side effects to applications.

Fine-grained performance isolation thus requires coordination between the OS and the application, but *how to accommodate the wide variety in virtual resources and their usage among different applications?* Through analyzing real-world intra-app interference issues, our insight is that despite their variety, they can be reduced to a small set of what we call *state events*. By exposing these events, it is feasible for the OS to recognize and mitigate interference effectively and safely.

Based on this insight, we design a few general *pBox* APIs for an application to communicate its state events to the kernel. A kernel manager leverages the state events and other information to provide performance isolation at *pBox* granularity.

At the algorithmic level, we address two key challenges. First, the *pBox* manager needs to proactively detect *imminent* interference. Compared to current cross-app isolation solutions that reactively detect interference from the overall SLO metrics, we face a more strict requirement. This is because our performance isolation targets a finer granularity, namely each *pBox*. Intra-app interference may also occur among a few activities, thereby escaping SLO monitors. Also importantly, since we cannot reclaim a contended virtual resource, we need to detect interference early (ideally before it occurs). This early detection is crucial to minimize a noisy *pBox*'s impact.

To tackle this challenge, we design an algorithm that uses a *worst-case* style analysis to *predict* whether the specified isolation goal of any *pBox* might be violated. If so, the algorithm additionally identifies the victim and noisy *pBox*es.

The second challenge is taking effective action. For safety, the *pBox* manager does *not* reallocate virtual resources. It instead simply applies a delay penalty to the noisy *pBox*. Because we can detect imminent interference early, the penalty typically can prevent the noisy *pBox* from causing more severe contention. We design an algorithm that uses an adaptive penalty length and carefully chooses the penalty timing.

Since *pBox* is activated during regular execution, it should not incur significant overhead. We design the *pBox* detection and prediction algorithms to be lightweight yet effective. We delegate some pre-processing of the application state events in a user-level library and minimize the kernel boundary crossings. We also track certain application state events when the application makes regular system calls.

Like any OS abstraction, using *pBox* in an application code requires developers' involvement. We design the *pBox* APIs to be intuitive and support typical application architectures. Our focus on high-level isolation goals alleviates developers from hard-to-specify resource quotas or reasoning about the complex relationship between virtual resources and end-to-end performance. Developers do need to annotate the state events of a virtual resource. In our experience, such efforts are moderate. We also design a static analyzer that can automatically identify many of the state events in a codebase.

We implement *pBox* in the Linux kernel 5.4 along with a user-level library. For evaluation, we choose five large server applications—MySQL, Apache, PostgreSQL, Vanish, and Memcached—and integrate *pBox* APIs into these complex codebases without significant effort. To test the performance isolation capabilities of the added *pBox* code, we reproduce 16 *real-world* intra-application performance interference issues in the five applications. *pBox* reduces the performance interference for 15 cases, by an average of 86.3% and up to 113.6%. We compare with four start-of-art solutions [10, 18, 48, 53]. They at best only reduce the performance interference for five cases by 38.8% on average, and would make the interference worse in the majority of the cases.

This paper makes the following contributions:

- We propose pBox, an abstraction that pushes the performance isolation boundaries into an application to address the intra-app interference issues facing modern applications.
- We address several design challenges, including identifying a small set of general state events to support diverse virtual resources, and designing algorithms to proactively detect imminent interference and take effective actions.
- We implement *pBox* in the Linux kernel and a library, along with a companion analyzer. We show *pBox*'s effectiveness on real intra-app interference issues in large applications.

pBox is available at https://github.com/OrderLab/pBox.

2 Background and Motivation

Intra-app interference refers to an application activity experiencing severe performance issues due to some other independent activity in that application. We discuss three *real* issues from MySQL to show the characteristics of this problem.

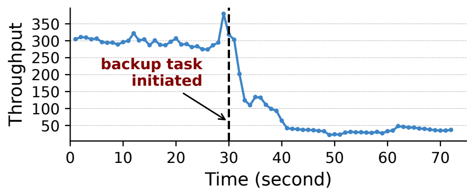

Figure 2. Throughput of all foreground clients

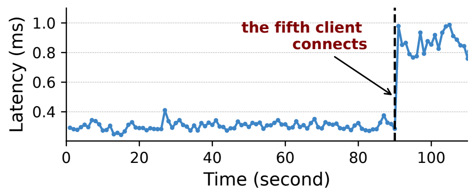

Figure 3. Avg. latency of requests from client 4. A fifth write-intensive client connects around time 90 s.

2.1 Real Intra-App Interference Cases

Case 1: UNDO log. MySQL has different transaction isolation levels. The default setting establishes a snapshot at the first read. While this is convenient, users found it can cause severe performance interference in production [75, 85]. InnoDB is a multi-version concurrency control (MVCC) storage engine, which uses a UNDO log that keeps transaction history. If there are long transactions with old versions, the UNDO log can grow large. As a result, when the old transactions are committed, MySQL's purge thread needs to spend a long time cleaning up the UNDO log, blocking other activities.

To reproduce this case, we create a database with 1 table and run two clients: *A* performs reads and *B* performs writes. *A* issues each read request in a transaction, sleeps for 10 seconds after the request finishes, then commits the transaction. By doing so, we have a long transaction that keeps an old version of the table. Consequently, each write request from client *B* needs to update the UNDO log and causes a large UNDO log.

Although client A does *not* hold an exclusive table lock and thus would *not* block client *B*, *B*'s latencies are still impacted when *A* commits transactions. As Figure 1 shows, 10 seconds after client A joins, client *B*'s latencies increase by about 4×.

The source of this interference is a **virtual resource**—the UNDO log. The write queries result in the rapid growth of the UNDO log, which increases the cleaning cost. In turn, read queries are severely impacted because the UNDO log is frequently held by the purge thread (iterating log entries).

Case 2: Buffer Pool. MySQL keeps a buffer pool to cache the accessed table and index data. While it generally improves performance, as reported by users [66], a backup task using `mysqldump` can use many blocks in the buffer pool and cause severe interference to other activities.

To reproduce this case, we create a small table (200 MB) that fits in the Innodb Buffer Pool (512 MB), and a larger table (4 GB) that does not fit in the buffer pool. We run four clients with uniform sysbench [42] OLTP on the small table and in time 30 seconds, we run a background mysqldump task on the second table. As Figure 2 shows, the throughput of the four clients is initially around 300 req/sec, but the interference from the backup task causes their throughput to drop by 10×.

The **virtual resources** involved in the interference are the buffer pool and its free blocks. When the dump activity takes many blocks from the buffer pool, it causes other activities for the four clients to frequently evict its old pages, which in turn leads to additional I/O costs for their requests.

```
buf_block_t* buf_LRU_get_free_block(
   buf_pool_t* buf_pool) {          mutex also used in
   ...                               other places, e.g.,
loop:                                returning a block
   buf_pool_mutex_enter(buf_pool);
   block = buf_LRU_get_free_only(buf_pool);
   if (block) {
      buf_pool_mutex_exit(buf_pool);
     return block;       no longer hold the mutex
  }                        after getting the block
  /* If no block was in the free list, search
     from the end of the LRU list */
  freed = buf_LRU_scan_and_free_block(buf_pool,
        n_iterations > 0);
  buf_pool_mutex_exit(buf_pool);
   if (freed)
      goto loop;
}
```

Figure 4. Finding a free block from the buffer pool in MySQL.

Case 3: Tickets. MySQL uses separate threads in its InnoDB engine to process requests from client connections. To minimize context switches, it limits the number of concurrent threads by the `innodb_thread_concurrency` parameter. While such a design is justified, it can cause performance interference among client connections as reported by users [57, 74, 76].

To reproduce this issue, we create a database with 5 tables (10*K* records per table). The thread concurrency is set to 4. We run three clients performing write-intensive workloads and one client performing read-intensive workloads. After around 90 seconds, a fifth client joins and issues write-intensive queries. Each client only queries one dedicated table.

Figure 3 shows the latencies of client 4 (executing read-intensive workload). In the first 90 seconds, this client's average request latency is around 0.3 ms. When the fifth client connects, even though it operates on a different table, the latency of the fourth client increases to around 0.9 ms, which is 3× slower than the non-interference case.

The interference involves two **virtual resources**—an integer `n_active` and tickets. If a thread tries to enter InnoDB, it checks whether the number of threads inside InnoDB has reached the concurrency limit, by comparing an integer `n_active` with the `innodb_thread_concurrency` parameter. If so, it needs to wait and check again. Otherwise, `n_active` is incremented and the thread is given a number of *tickets*. The thread can then enter and leave InnoDB freely until the tickets are used up.

2.2 Observations

Intra-app interference issues are often not strictly a bug but a design trade-off. Even after developers become aware of such a trade-off, they may find the issue difficult to fix and keep the design as is. For example, the InnoDB thread concurrency regulation in *case 3* can reduce context switches and improve scalability. Its limitation has been known for *more than 10 years*, but developers still keep it as a hard-to-tune parameter [76]. As a result, performance interference issues can exist in an application for a long time. We need solutions that can dynamically mitigate performance interference.

In addition, while intra-app performance interference involves contention, the issues are more complex than typical poor synchronization. For example, for case 2, as Figure 4 shows, while a lock is used when accessing the buffer pool, the lock is soon released after a block is obtained. Thus, the real contended virtual resources are the free blocks, which are used by the noisy activities without the lock. Similarly, the core issue in case 1 is the subsection growth of the UNDO log rather than an unfair lock. Thus, simply optimizing lock or other synchronization mechanisms is ineffective.

2.3 Challenges and Gaps

Existing interference mitigation solutions use the allocation of hardware resources as the control mechanism. They are ineffective in addressing the intra-application interference issues shown earlier, which can occur even when many idle hardware resources are available. Blindly adjusting hardware resources may even aggravate the interference. In *case 1*, if we lower the CPU quota for the read requests or purge thread, it would cause even worse write latencies, because the victim activities were waiting for a virtual resource from the noisy activity and would need to wait longer.

Dropping noisy requests is a non-solution either. Production workloads are unpredictable, so it is difficult to know in advance which requests will cause interference. For example, both write (*case 3*) and read (*case 1*) queries can cause interference in MySQL. Moreover, users expect applications to provide strong performance isolation instead of dropping requests. It is also common for the interference to be caused by a background activity instead of a request.

Complex applications may implement custom mechanisms that attempt to mitigate performance interference. For example, MySQL allows limiting resources at the user account level, such as the number of queries an account can issue per hour [19]. However, they are helpful in preventing overload but are ineffective in addressing normal interference, which occurs even when a client sends a small number of requests.

In summary, despite the extensive effort into mitigating performance interference, there is a lack of an effective and systematic mechanism to provide strong performance isolation *within* applications that users expect and desire.

3 Overview of *pBox*

Motivated by the observations from Section 2. We propose a new OS abstraction called *pBox* that pushes the boundary of performance isolation into the application for developers to systematically minimize intra-app interference.

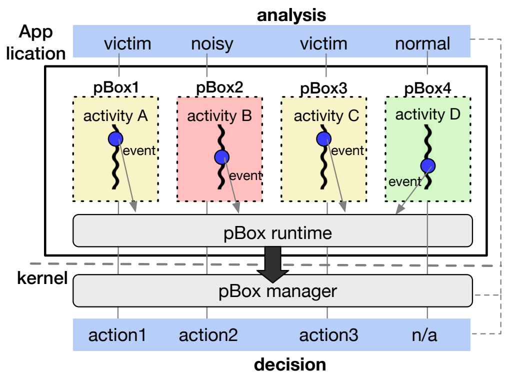

Figure 5. Overview of *pBox*.

Insight. Our insight is that the essence of intra-app performance interference is different application activities contending on *virtual resources*, such as buffers and tickets. Thus, it is invisible to the OS and cannot be simply mitigated through adjusting hardware resources. *pBox* tackles this characteristic by making the OS aware of virtual resource contention.

Abstraction. *pBox* is a *performance isolation domain* within an application that logically divides an application's execution into independent activities, preventing activities within one domain from poor performance due to the execution of activities in other domains. Existing abstractions such as resource container [7] can capture activities within an application. However, they focus on delineating resource principals while treating each activity separately. In comparison, *pBox* focuses on *interactions* across different activities and their *scheduling*. It monitors the interactions to detect contention and applies scheduling actions to achieve a performance isolation goal.

Usage. Developers create a *pBox* around code that represents an application activity boundary. They directly specify a high-level performance isolation goal for this *pBox*, e.g., a maximum interference level *x*. The runtime then aims to achieve the goal for activities executed within this *pBox*.

pBox supports flexible granularity. For example, in a request-based application, developers can define a *pBox* for each request. They can also define a *pBox* for each client connection. In this case, the *pBox* is created when a client connection is established and destroyed when the corresponding connection is closed. Note that one connection may send *N* consecutive requests of different types, *e.g.*, write requests followed by read requests. This *pBox* will be activated *N* times to provide performance isolation for *N* activities—the handling of each request from this connection. For *C* concurrent client connections, there can be *C pBox*es. Besides request handling, developers also create *pBox*es for other activities, *e.g.*, one *pBox* for each background thread.

Architecture. Figure 5 shows the *pBox* system overview. *pBox* exposes a few general APIs (Section 4.1) for application developers to use. A user-level library will be linked with the application. The library traces critical state events (Section 4.2)

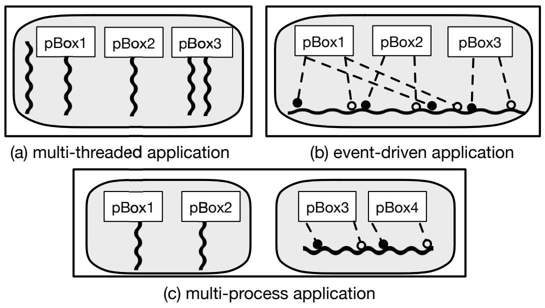
pBox1
pBox2
pBox3
pBox4
(a) multi-threaded application
(b) event-driven application
(c) multi-process application

Figure 6. *pBox* supports common application architectures.

```
/* Create a pbox with an isolation rule, return psid */
int create_pbox(IsolationRule rule)
int release_pbox(int psid)
void activate_pbox(int psid)
void freeze_pbox(int psid)

/* Inform a state event in the current pbox about a
virtual resource named by key (typically an address) */
int update_pbox(size_t key, event_type event)

/* Detach current thread's pbox, associate the pbox with
a key k (not a virtual resource key), return its psid */
int unbind_pbox(size_t k, unbind_flags flags)
/* Find the pbox associated with k, bind it
with the current thread, return its psid */
int bind_pbox(size_t k, bind_flags flags)
```

for event-driven apps

Figure 7. Main *pBox* APIs.

```
void do_handle_one_connection(THD *thd) {
  IsolationRule rule = { .type = RELATIVE,
    .isolation_level = 30 };
  int psid = create_pbox(rule);
  // keep receiving commands
  while (thd_is_connection_alive(thd))
    if (do_command(thd)) break;
  release_pbox(psid);
  close_connection(thd);
}
```

sql/sql_connect.cc

```
bool do_command(THD *thd) {
  command = thd->net.read_pos[0];
  int psid = get_current_pbox();
  activate_pbox(psid);
  // handle a command, e.g., a request
  ret = dispatch_command(command, thd, ...);
  freeze_pbox(psid);
}
```

sql/sql_parse.cc

Figure 8. Example of using *pBox* in MySQL.

about application *virtual resources* and communicates them to a kernel-level manager. The manager monitors the execution of all the *pBox*es. Using the state events along with other information, the manager runs a detection algorithm (Section 4.3) to determine if any *pBox* might suffer from interference soon, and detect the potential noisy *pBox*(es) and victim *pBox*(es). It then carefully applies penalty actions on the noisy *pBox*(es) (Section 4.4) to achieve the isolation goal.

4 Design of *pBox*

In this section, we describe the interfaces of the *pBox* abstraction, the system components for supporting *pBox*, and the algorithms for *pBox* to mitigate performance interference.

4.1 Main APIs and Usage

As an abstraction, *pBox* should be general enough to support a wide range of applications with different architectures and programming paradigms. As Figure 6 shows, there are three common application architectures: (a) multi-threading; (b) event-driven; (c) multi-process. In (a) and (c), one request or task is typically handled by one thread or process. For (b), multiple requests or tasks share the same thread. *pBox* provides a few APIs (Figure 7) that support all three architectures.

An application calls `create_pbox` in a region that represents an activity boundary to be protected. Such boundaries are well-defined. For example, in MySQL, if developers want to create a *pBox* for each client connection, they add a call at the start of function `do_handle_one_connection` (Figure 8). At runtime, the kernel creates a new *pBox* instance and binds it with the current thread that handles an incoming connection.

The `create_pbox` API takes an `IsolationRule` argument for developers to specify an isolation goal. A typical type of isolation rule specifies the *relative* performance behavior, particularly latency increase, compared to the ideal, non-interference execution. For example, a rule of 50% indicates that the *pBox*'s execution latency should not be more than 50% worse than the latency if there was no interference (no other *pBox*es slowing it down). In Section 4.3.1, we discuss how *pBox* enforces a relative isolation rule even though the ground truth of non-interference performance is unknown.

When the application starts a new activity in a *pBox*, developers can activate the *pBox* by calling `activate_pbox`, which causes the manager to start tracing this *pBox* and provide performance isolation for it. Once the activity finishes, the application calls `freeze_pbox`, which stops the tracing for this *pBox*. For example, if a *pBox* represents a client connection thread, the `activate_pbox` and `freeze_pbox` can be called when the thread starts and finishes processing one request from the connection, respectively (Figure 8).

When the condition of a virtual resource changes, the application signals a *state event* (Section 4.2) by calling `update_pbox`. To support diverse virtual resources, the *pBox* names a virtual resource with a generic key, which is typically the address of the resource object. The manager does not need to understand the semantics of a virtual resource. It only needs the key to group recorded information such as the state events.

For event-driven applications, multiple *pBox*es share the same thread and only one *pBox* owns a thread at one time. To support these applications, we provide two *ownership transfer* APIs. The `unbind_pbox` API detaches the *pBox* bound with the current thread and then associates this *pBox* with a key (different from the resource key). The `bind_pbox` API finds the *pBox* associated with a given key and binds it with the current thread. For example, in a typical event-driven application, when a request finishes processing, the connection will be

State event	Description
`PREPARE`	The *pBox* is deferred by a virtual resource that is currently held by another *pBox*.
`ENTER`	The *pBox* is no longer deferred by the resource
`HOLD`	The *pBox* is holding a virtual resource
`UNHOLD`	The *pBox* has released the virtual resource

Table 1. Four state events for application virtual resources. A virtual resource can be mutual exclusive, or exclusive with multiple units. It can also be composed of multiple parts.

put into the event queue. Before the queuing, developers add an `unbind_pbox` call with the connection IP as the key. At the place where a new request from a connection is executed in the worker thread, developers add a `bind_pbox` call.

4.2 State Event

We now introduce a key concept to *pBox*, *state events*.

4.2.1 Rationale. *pBox*'s insight is to make the OS aware of application virtual resources. However, informing the OS of every change in application virtual resource usage is too overwhelming and imposes too much overhead. In addition, application virtual resources have a wide range of semantics and characteristics, the OS lacks the knowledge to transparently manage different virtual resources for an application.

Through analyzing real-world cases, we summarize four general types of conditions that apply to all kinds of virtual resources. Recognizing these conditions is necessary for addressing interference. We call them *state events*: (1) `PREPARE`; (2) `ENTER`; (3) `HOLD`; (4) `UNHOLD`. Table 1 lists their semantics.

An alternative is the traditional resource acquire-release model. However, that model does not capture the key characteristics of performance interference: one activity is causing delay to or deferred by another activity.

The `PREPARE`/`ENTER` events can capture how long an activity is deferred when it tries to acquire a resource or during the usage of the resource. The reason we distinguish the `ENTER` and `HOLD` state events is that a virtual resource may consist of multiple parts and an activity is unblocked from a partial resource but still does not hold the full resource.

4.2.2 Finding state events. A state event is about the usage status of an application virtual resource. Identifying it therefore requires domain knowledge. Developers (not users) possess this knowledge to find code places to call `update_pbox`. Leveraging state events from these API calls, the *pBox* manager *automatically* detects and mitigates performance interference.

One approach to finding state events is based on the types of objects that may cause contention, *e.g.*, queues and buffers. However, applications have many custom implementations of these types, which can be easily missed.

We observe a more robust heuristic. Intra-app performance interference usually comes down to the application using waiting-related syscalls to block a victim task, such as `sleep`, `futex`, or `select`. Thus, developers can first find call sites of

```
195 void srv_conc_enter_innodb_with_atomics() {
204   update_psandbox(&srv_conc.n_active, PREPARE);
207   for(;;) {
209     if (srv_conc.n_active < thread_concurrency) {
213       n_active = os_atomic_inc(&srv_conc.n_active);
216       if (n_active <= thread_concurrency) {
218         update_pbox(&srv_conc.n_active, ENTER);
219         update_pbox(&srv_conc.n_active, HOLD);
221         srv_enter_innodb_with_tickets(trx);
243         return;
244       }
251     }
281     os_thread_sleep(sleep_in_us);
292   }
293 }
299 void srv_conc_exit_innodb_with_atomics() {
305   os_atomic_dec(&srv_conc.n_active, 1);
306   update_psandbox(&srv_conc.n_active, UNHOLD);
307 }
                    storage/innobase/srv/srv0conc.cc
```

Figure 9. Example usage of `update_pbox` API in MySQL, which can mitigate interference issues such as case 3 in Section 2.1.

such a syscall. Then they can check whether a shared variable accessed by multiple activities is used to determine the control paths to a call site. If so, this shared variable is likely a critical virtual resource of interest. Developers can then add the four state events for this resource. In comparison, if the paths to a blocking call site only involve variables that are accessed by one activity, it is likely self-waiting (*e.g.*, a periodic task or retries on I/O errors) that can be skipped.

Figure 9 shows an example of adding the `update_pbox` APIs to the MySQL InnoDB code based on the above heuristics. The shared variable `srv_conc.n_active` is a virtual resource being contended by multiple activities and the sleep call at line 281 represents an activity being blocked.

Note that developers are not expected to do a perfect job in finding state events. As we later show (Section 6.8), *pBox* can tolerate incomplete or inaccurate `update_pbox` calls and still effectively mitigate interference.

We further design a companion static analyzer tool (Section 4.5) to help developers. The tool implements an algorithm based on the above heuristics and automatically analyzes the codebase to find potential virtual resources.

4.3 Prediction and Early Detection of Interference

To achieve strong performance isolation, the manager must monitor each *pBox*'s execution and proactively detect if a violation is *imminent*. Early detection is especially important because of the fine granularity of performance isolation and the fact that we cannot reclaim a contented virtual resource.

A fundamental challenge, however, is that virtual resource usage is low-level information, while the isolation rule is about end-to-end latency. During an activity's execution, we do not know what its final latency will be, nor how much each virtual resource will contribute to the final latency. Given a relative isolation rule (Section 4.1), we also need to know the baseline (interference-free) performance, which is usually unavailable.

4.3.1 Metrics and Algorithm. To tackle this challenge, we use a *pBox*'s *deferring time* on virtual resources as a main

Algorithm 1: Detect potential interference in *pBox*

```
Global: competitor_map - event info by resource key.
Input: (key, event) - arguments in an update_pbox call.
1  $p \leftarrow get\_current\_pbox()$, $now \leftarrow$ get_time();
2  switch $event.type$ do
3      case $PREPARE$ do
4          $competitor\_map[key]$.add($\{p, now\}$);
5      case $ENTER$ do
6          $competitors \leftarrow competitor\_map[key]$;
7          forall $c \in competitors$ do
8              if $c.pbox == p$ then
9                  $competitor\_map[key]$.remove($c$);
10                 $defer \leftarrow now - c.time$;
11                 $p.defer\_time \leftarrow p.defer\_time + defer$;
12     case $HOLD$ do
13         $p.holder\_map$.add($key$);
14     case $UNHOLD$ do
15         if $p.holder\_map.remove(key)$ then
16             $competitors \leftarrow competitor\_map[key]$;
17             forall $c \in competitors$ do
18                 $waiter \leftarrow c.pbox$;
19                 $t_e \leftarrow now - waiter.activity\_start$;
20                 $defer \leftarrow now - c.time$;
21                 $t_d \leftarrow waiter.defer\_time + defer$;
22                 $t_f \leftarrow t_e/(t_d - t_e)$;
23                 if $t_f > waiter.goal$ and $p.time < c.time$ then
24                     $noisy \leftarrow p$, $victim \leftarrow waiter$;
25                     take_action($noisy, victim$);
```

metric. Our rationale is that interference occurs when an activity is deferred for a long time. We define the *deferring time* for one activity to be the additional execution time caused by other activities. Assume an activity uses a set of resources $r_1, r_2, \ldots, r_n$, and a list of `PREPARE` and `ENTER` state events are received for each resource. We denote the time of receiving the `PREPARE` events as $prep_{r_1}, prep_{r_2}, \ldots$ and the time of receiving the `ENTER` events as $ent_{r_1}, ent_{r_2}, \ldots$. The deferring time is calculated as $T_d = \sum_{i=r_1}^{r_n} ent_i - prep_i$.

We did not choose holding time as a metric, because holding a virtual resource for long does not mean the *pBox* is noisy.

To connect the deferring time metric to the end-to-end isolation goal, we treat the unknown baseline (interference-free) as an ideal execution with zero deferring time.

Assume the total execution time of an activity in a *pBox* is T_e and its total deferring time is T_d. Its interference level $T_f = \frac{T_e}{T_e - T_d} - 1 = \frac{T_d}{T_e - T_d}$. A given isolation goal λ is violated if $T_f > \lambda$. The problem is that we do not know T_d before this activity finishes, so we need to approximately compute T_f.

To achieve early detection, i.e., *predicting* whether a *pBox*'s execution so far is in danger of violating the performance isolation goal, we use a worst-case analysis inspired by the worst-case execution time (WCET) analysis [82]. In particular, using the *current* defer time t_d and the current execution time t_e, we can compute a simple approximate $t_f = \frac{t_d}{t_e - t_d}$. If $t_f > \lambda$, we can have confidence that if the activity's later execution still maintains the same ratio, the *pBox cannot* achieve its goal. Thus, it would be a good time to take action.

Algorithm 1 shows the core interference detection algorithm. When a `UNHOLD` event is received (line 14), the manager first checks whether the current *pBox* is the holder of the virtual resource. If so, it iterates through all the waiting *pBox*es (line 17). If it finds a *pBox* whose t_d is too long and the current *pBox* is the holder before the waiting *pBox*, we detect potential interference and find both the noisy *pBox* and the victim *pBox*.

The aforementioned detection logic is about one activity executed in a *pBox*. Due to the fundamentally limited information, we may miss detecting and mitigating interference in one activity. Thus, the *pBox* manager also monitors the overall performance of a *pBox* performance and detects interference at the *pBox* level. To do so, it keeps a history of T_d as well as the T_e. It calculates the average interference level $\overline{T_f} = \frac{\overline{T_d}}{\overline{T_e - T_d}}$. If the manager finds one *pBox*'s $\overline{T_f}$ is close (default 90%) to λ, it will also take action at the end of the activity.

Besides calculating the average, the manager supports other metrics including tail and max based on the same principle.

Note that our algorithm does *not* assume a *pBox* accesses only one resource at a time. It uses unique keys to identify state events for different resources, so it tracks them separately and concurrently. It also does *not* have resource dependencies requirements. The deferring time is calculated based on the timing of the state events. A different order of events would change the time but not the accuracy of detection.

4.3.2 Tracking Execution Information.

The manager tracks each *pBox*'s execution information to both support the detection algorithm and facilitate mitigation actions (Section 4.4).

It tracks four statuses for each *pBox*: *start* (*e.g.*, a new client connection is established), *active* (*e.g.*, a new request from the connection is received), *freeze* (*e.g.*, the request handling finishes), and *destroy* (*e.g.*, the connection is closed).

The manager begins to trace state events after a *pBox* is in an *active* status, and ends tracing once it is in a *freeze* status.

When a `PREPARE` event is received, the manager notes this *pBox* in a deferred state about a virtual resource and adds it to a **competitor map** (list of *pBox*es waiting for a resource). When the *pBox* receives an `ENTER` event on the same resource, the deferred state is ended and the manager calculates the deferring time. If a *pBox* receives a `HOLD` event, the manager records it in a **holder map**. The two maps are used in the interference detection and mitigation logic.

4.4 Prevention and Mitigation of Interference

After detecting potential interference, the *pBox* manager needs to take action. Unlike cross-app interference, where the kernel can transparently adjust hardware resources, directly reallocating a contended virtual resource can easily introduce dangerous side effects to an application. For example, directly revoking a lock object from one activity and granting it to another activity can easily violate critical section safety.

4.4.1 Action and Timing. We use penalizing the noisy *pBox* as the main control action so that we can achieve performance isolation without breaking application logic. There are multiple ways to achieve the penalty, such as reducing scheduling slices (giving more to the victim *pBox*), and lowering priority.

We choose a simple type of penalty: adding a delay to slow down the noisy *pBox*. In the Linux kernel, it is done by calling `schedule_hrtimeout`. Compared to other penalties, it introduces a simpler effect, which in turn makes it easier to predict the mitigation effectiveness and make the interference mitigation algorithm (Section 4.4.2) less complex. Also, this simple penalty avoids conflicting with the main OS scheduler.

Applying penalty actions to noisy *pBox* might violate the noisy *pBox*'s isolation goal and trigger additional penalty action. To avoid the cascaded penalty, the detection algorithm 1 only uses the deferring time on virtual resources to determine the interference level. Thus, the violation caused by penalty action would not be considered interference.

The timing of the penalty action requires care. If a virtual resource is still held by a noisy *pBox*, penalizing the noisy *pBox* would cause the victim *pBox* to wait even longer for the virtual resource. Thus, the manager waits until the noisy *pBox* no longer holds the virtual resource to apply the penalty.

Another caveat is nested state events. A noisy *pBox* may hold multiple virtual resources at the same time, so during its penalty, it may still cause interference for other *pBox*es. To avoid this situation, the manager conservatively waits for the noisy *pBox* to release all the virtual resources and takes action at once. As a result, the noisy *pBox* can be penalized without causing more performance interference.

4.4.2 Adaptive Penalty. The mitigation effectiveness depends on the penalty action's length. An improper length may exacerbate the interference. Rather than using a fixed length, which is hard to set, we adaptively adjust the length.

When the manager detects a noisy *pBox* (Algorithm 1), it checks the action history. If this *pBox* has not been penalized for the contended virtual resource before, the manager sets an initial value p_1 as the penalty length. Otherwise, the length is adjusted based on the effect of the previous penalty.

We evaluate whether a penalty is good or not by comparing the *victim pBox*'s performance before and after the penalty. In particular, we calculate $s(i) = \overline{T_d^i}/\overline{T_e^i}$ for the victim *pBox*, where $\overline{T_d^i}$ and $\overline{T_e^i}$ are the victim's average deferring time and execution time until the i-th action, respectively. This ratio reflects the interference level (Section 4.3).

We design two adaptive policies. The first one is score-based. If $s(i+1)$ is larger than $s(i)$, which means the penalty does not reduce the interference level, we increment the score by one. Otherwise, we decrement the score by one if it is positive. The penalty length for the next action is set to $p_{i+1} = p_1 \times (1 + score/\alpha)$, where α by default is 5, so each ineffective action would increase the next penalty time. This policy's convergence to the optimal penalty may be slow.

Thus, we design a second policy inspired by the gradient descent algorithm. We measure the gap from the isolation goal (λ), $gap = s(i+1) - \lambda$, and the delta $\delta = 1 - s(i)/s(i+1)$. The next penalty length is set to $p_{i+1} = p_i \times gap/\delta$. This policy is faster but a step may be too large to reach the optimal value.

The manager dynamically chooses between the two policies. If the deferring time is much larger than the penalty, it chooses the second policy. Otherwise, it chooses the first policy.

To choose the initial penalty p_1, we assume a simple but representative interference model: one noisy *pBox* and one victim *pBox*. The *pBox* manager derives a formula to calculate the optimal penalty length under this model: $p_1 = \sqrt{t_d(victim) \times t_e(noisy)} - t_e(noisy)$. In this way, the p_1 would not be far away from the real optimal result.

4.4.3 Is The Action Too Late? A limitation with using delay as the penalty action is that the action might be too late. For instance, if the interference is caused by a noisy *pBox* holding a virtual resource for a long time near the end of an activity's execution, by the time the *pBox* manager can safely act, the penalty may be useless.

While adding complex transaction mechanisms may address this limitation, our design has the advantages of simplicity and safety. As we later show (Section 6), it is quite effective.

There are several reasons that can explain its effectiveness. First, our detection algorithm (Section 4.3) is proactive, which can find imminent interference before it reaches the level of violating the performance isolation goal. As a result, the penalty action(s) can be applied early on to *prevent* the violation or at least *minimize* the interference impact.

Second, we find that in real-world intra-application interference cases, a noisy *pBox* often creates contention on some virtual resource more than once, either within one activity or across a sequence of activities. Take case 1 in Section 2.1 as an example, the virtual resource is the UNDO log, and the noisy activity keeps adding or cleaning up entries in it. Similarly, in case 2, the virtual resource is the buffer pool, and the noisy activity frequently obtains blocks from it.

Third, lateness in taking action can be more probable when an activity finishes execution quickly. This can impose demanding requirements on the detection and mitigation, but we are targeting performance interference. In such a scenario, a noisy activity typically requires a longer execution time to cause severe interference.

For these reasons, despite the potential limitation, in practice, there are still many opportunities to effectively intervene.

4.5 Static Analyzer

We designed a companion static analyzer to help developers find state events when adding *pBox* to their applications. The analyzer is built on top of the LLVM framework [43], with

Algorithm 2: Identify locations to add state events

```
   Input: program - input application code; wait_funcs - a list of
          standard waiting functions
   Output: locations - locations to add update_pbox calls
1  call_insts ← getCallInstructions(program);
2  foreach inst in call_insts do
3      callee ← getCallee(inst), waitf ← nil;
4      if callee in wait_funcs then
5          waitf ← callee;
6      else
7          foreach f in wait_funcs do
8              if isWrapper(callee, f) then
9                  waitf ← f;
10                 break;
11     if waitf ≠ nil then
12         loop ← getLoop(inst);
13         if loop ≠ nil then
14             shared_vars ← getSharedVars(loop.cond);
15             if shared_vars ≠ nil then
16                 locations.add(<inst, shared_vars>);
```

around 800 SLOC in C++. It designs an algorithm based on the observations described in Section 4.2.

Algorithm 2 lists the core logic. The analyzer takes as input a list of standard library functions or syscalls that perform waiting, such as `semaop`, `pthread_sleep`, `pthread_cond_wait`, `pthread_yield`, and `apr_sleep`. Many applications also implement custom waiting functions that are wrappers of a standard function. The analyzer identifies such a wrapper (`isWrapper` at line 8) by checking whether a function calls some waiting function in *all* the paths. Specifically, it checks the Control Flow Graph (CFG) to see if this call instruction's basic block is a *post-dominator* [14] for the function's entry basic block.

The analyzer then finds all callsites for these waiting functions and the wrappers. Next, it checks whether a callsite is in a loop (line 13). If so, it checks whether the loop condition uses some variables shared by multiple activities (line 15).

A callsite that matches these conditions is a candidate location to add a state event. The analyzer outputs all locations and the associated shared variables (likely virtual resources). The output guides developers to add `update_pbox` calls.

5 Implementation

We have implemented a prototype of *pBox* in Linux kernel 5.4.1 and a user-level runtime library.

Lightweight Tracing. Since a *pBox* is activated during an activity's execution, we need to minimize the overhead of tracing and management. To make the tracing lightweight, we optimize the cost of each *pBox* operation. A major cost is allocating bookkeeping data structures such as the state event hash table and competitor map. We reduce this cost by using pre-allocation. For example, for the `bind_pbox` operation, one *pBox* would normally only bind to one key (variable) at a time. Thus, we allocate a small array in the *pBox*'s struct during its creation phase. In later binding operations, we just find a free slot in the array and allocate only if all slots are used.

After optimizing the core operation itself, the syscall overhead dominates. We further reduce the number of syscalls, especially for `update_pbox`. The user-level library checks whether `HOLD` has a matched `UNHOLD` event and only calls `update_pbox` when there is a match. Since the two events are used to locate the noisy *pBox* and take action if needed, we can skip the syscall upon redundant events. This decision needs to find the associated *pBox*, which still requires a syscall. To avoid this syscall, we use thread local storage to record the *pBox* id when it is created. Then we keep an array in each *pBox* to check the ownership of the virtual resource.

Supporting Event-driven Model. Event-driven applications can be single-threaded or multi-threaded (thread pool). The bind and unbind *pBox* APIs take a `flags` argument that can indicate whether the currently bound thread is a shared thread or a dedicated thread. If a noisy *pBox* is bound with a dedicated thread, the manager takes action immediately. If the bound thread is shared, penalizing the noisy *pBox* with a delay would prevent other *pBox*es from using this thread. The manager instead makes the following activities from the noisy *pBox* wait in the task queue for a while. Specifically, the manager keeps a penalty timestamp. If an activity from the noisy *pBox* selected to execute next happens within the timestamp, the activity is put back to the task queue.

One challenge is how to manipulate the application task queue without causing side effects. We observe that event-driven applications commonly leverage kernel-level queues for task management by using syscalls such as `accept` and `epoll`. In such cases, the *pBox* manager traces state events at the application level but take action in the kernel queues by modifying the syscall implementations to achieve transparent mitigation. If applications do not leverage kernel-level queues, developers need to annotate the task queues.

Lazy Unbind. In high-performance event-driven applications, we observe that the same thread might frequently bind and unbind to the same *pBox*. We introduce a lazy unbind optimization to reduce the number of syscalls. Under this mode, when the library receives a `unbind_pbox` call, it marks the *pBox* as detached and pauses its state event tracing, but does not make a syscall. At the kernel side, the *pBox* is still bound with the current thread. In the next `bind_pbox` call, the library checks if it is about the same detached *pBox*. If so, the library removes the detached flag, also without making a syscall. Otherwise, it makes a syscall for the manager to unbind the last *pBox* and bind the new one.

6 Evaluation

We evaluate *pBox* to answer several questions: 1) Can *pBox* reduce intra-app interference? 2) How does *pBox* compare to state-of-the-art solutions? 3) Is *pBox* robust? 4) What is the overhead? 5) How much effort is needed to use *pBox*?

Software	Desc.	Arch.	Version	SLOC
MySQL	Database	Multi-thd	5.6.22	1.74 M
PostgreSQL	Database	Multi-proc	9.2.0	629 K
Apache	Web server	Multi-thd	2.4.38	198 K
Varnish	Proxy server	Event-driven	4.0.0	59 K
Memcached	Key-value store	Event-driven	1.4.29	19 K

Table 2. Evaluated software.

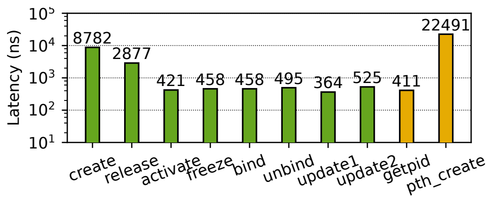

Figure 10. Latency (ns) of *pBox* operations. *update2* (*update1*): call `update_pbox` under (no) interference; *pth_create*: `pthread_create`.

Setup. The experiments are conducted on servers with 10-core (20 hyper-threads) Intel Xeon E5-2640 CPUs at 2.4 GHz, 64 GB DRAM, and a 480 GB SSD, running Ubuntu 20.04. We evaluate *pBox* on five large, open-source applications (Table 2): MySQL, PostgreSQL, Apache, Varnish and Memcached. We choose them because they are widely used, and cover different functionalities and architectures. They are complex enough to test *pBox*'s generality. To measure the performance of these applications, for MySQL and PostgreSQL, we use sysbench as the benchmark tool [42]. For Apache and Varnish, we use the official Apache benchmark tool [25]. For Memcached, we use Mutilate [44] as the benchmark tool.

6.1 Microbenchmark

We measure the costs of *pBox* operations with microbenchmark. We write a test app that invokes different *pBox* APIs for 10 million times. We run the app 10 times and calculate the average latency for each operation.

Figure 10 shows the results. The *pBox* creation on average takes 8.8 μs, which is much faster than the pthread creation. For the other operations, the latency is around 420 ns to 500 ns, which is close to the `getpid` syscall latency.

6.2 Mitigating Real-World Issues

To evaluate the effectiveness of *pBox*, as Table 3 shows, we collect 16 *real-world* intra-application performance interference issues in the five software. All cases are collected from blog posts, ServerFaults [67], and application bug trackers. Only 4 cases are marked as bugs by developers. The rest do not have associated bug reports and are usually design trade-offs.

We reproduce these cases, measure their performance on vanilla Linux, and compare it with running them on the *pBox* versions. In the *pBox*es creation APIs, we use a relative isolation rule (interference tolerance level) of 50%. We choose 50% because contention is inevitable in modern applications and this goal is more realistic to consider the complexities of performance behavior in our evaluated applications. We evaluate the impact of different rule settings in Section 6.5.

Figure 11 shows the normalized latencies of the activities (threads or processes) that originally suffered from performance interference. *pBox* successfully mitigates (reduces the latencies) 15 of 16 cases.

The degree of mitigation matters. Let T_i denote the performance with interference, T_o denote without interference, and T_s denote the performance under a solution. Then, the original interference level is $p = \frac{T_i}{T_o} - 1$. The last column in Table 3 lists p for each case. Most cases experience severe interference. The interference level under a solution is $q = \frac{T_s}{T_o} - 1$. Thus, the interference reduction ratio $r = \frac{p-q}{p} = \frac{T_i - T_s}{T_i - T_o}$.

pBox significantly reduces the interference, by an average of 86.3% and as large as 113.6%. For case c16, *pBox* does not achieve effective mitigation, because the contention on the particular application resource is not heavy. In addition, since Memcached is a high-performance in-memory system, even one or two additional syscalls can be costly. The overhead of *pBox* exceeds the overall performance benefit from its mitigation actions. Note that *pBox*'s improvements for cases c2 and c15 are not negligible as Figure 11 might suggest. For readability, the normalization in Figure 11 is calculated as $\frac{T_s}{T_i}$, which does not always reflect r. For instance, in case c2, T_i is 23.95 ms; T_o is 21.67 ms; T_s for *pBox* is 21.99 ms. The normalized latency in Figure 11 is 0.91 ($\frac{21.99}{23.95}$), which seems a small improvement. However, *pBox* improves the victim activity's latency to be close to its non-interference latency (21.67 ms vs. 21.99 ms), achieving an 86% reduction ratio.

For tail latency, *pBox* reduces the 95th percentile for 13 cases (Figure 12), with an average reduction ratio of 54.6%.

In terms of the impact on the noisy *pBox*. The latency of the noisy *pBox* is only increased by an average of 34.1%.

pBox does not guarantee that the specified isolation goal can always be achieved. From measuring the first five cases, we observe that 94.6% of the activities meet the goal with *pBox*, whereas this number drops to 48.2% without *pBox*.

6.3 Comparisons with Existing Solutions

We choose four state-of-the-art performance interference mitigation solutions to compare with *pBox*: Linux cgroup [53], PARTIES [10], Retro [48] and DARC [18].

For standard cgroup, we use a script to dynamically identify threads that handle different types of workloads and put them into different cgroups. It also identifies background task threads and assigns them into one cgroup. Then the script configures an even CPU usage quota among the cgroups. In this way, a noisy workload or background task would not impact the CPU usage in other groups. For PARTIES, we modify its monitoring component to trace each client's latency. We use a script to identify threads that handle each client and configure them as PARTIES' control targets. PARTIES can then control resource usage at the client level. Retro is designed for Java-based distributed systems. We use the *pBox* codebase to re-implement its core design to apply to C/C++ programs. We

Id.	**Application**	**Bug**	**Virtual Resource**	**Description**	**Interf. Level**
c1 (link)	MySQL	N	custom lock	SELECT FOR UPDATE query blocks other clients' insert query	8.76
c2 (link)	MySQL	N	custom mutex	Inserting to tables without primary key would cause contention on global mutex	0.11
c3 (link)	MySQL	N	integer and tickets	Slow query blocks other clients' requests when concurrency limit is reached	10.70
c4 (link)	MySQL	Y	integer variable	SERIALIZABLE isolation model causes significant overhead to SELECT locking	6.61
c5 (link)	MySQL	N	UNDO log	Background purge task blocks the client's request when purging the UNDO log	15.35
c6 (link)	PostgreSQL	Y	table index	In-progress INSERT causes other queries to spend time on MVCC	39.16
c7 (link)	PostgreSQL	N	table-level lock	Select for update query blocks the request on other tables	1204.28
c8 (link)	PostgreSQL	N	table-level lock	LWlock waiters for exclusive mode are blocked by shared mode locker	1727.95
c9 (link)	PostgreSQL	N	dead table rows	Vacuum full process blocks other requests	419.14
c10 (link)	PostgreSQL	N	write-ahead log	A large WAL causes the group insertion blocking other requests	3.69
c11 (link)	Apache	Y	fcgid request queue	slow request in mod_fcgid blocks other fast connections	1621.12
c12 (link)	Apache	N	apache thread pools	Apache locks server if reaching maxclient	1429.21
c13 (link)	Apache	N	php thread pool	Apache server suddenly slows when the connection reaches pm.maxchildren	352.38
c14 (link)	Varnish	N	varnish thread pool	Slow request on visiting big objects blocks the requests on small objects	18045.79
c15 (link)	Varnish	Y	system lock	WRK_SumStat lock contention with high number of thread pools	0.68
c16 (link)	Memcached	N	system lock	lock contention in the cache replacement algorithm	0.73

Table 3. Description of 16 *real-world* intra-application interference cases we collected and reproduced in the five evaluated software. *Bug*: *Y* means the interference case is from a bug report; *N* means the interference case is from some user post without a corresponding bug report.

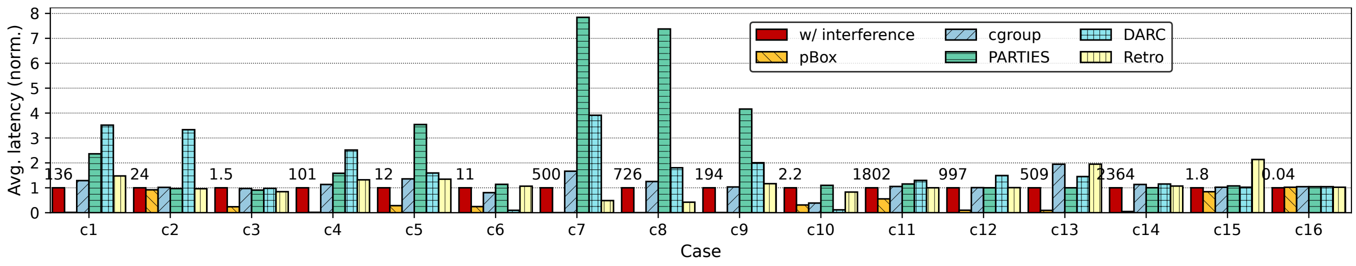

Figure 11. Avg. latency for each case normalized by the interference performance in the original application, compared to running the application with (1) *pBox* (2) cgroup (3) PARTIES [10] (4) DARC [18] (5) Retro [48]. The interference is reduced if the normalized latency is below 1. The lower it is, the higher the interference reduction ratio. Normalized latency above 1 means the interference becomes worse. The numbers above the red bars are the absolute latencies (in *ms*) for the interference performance.

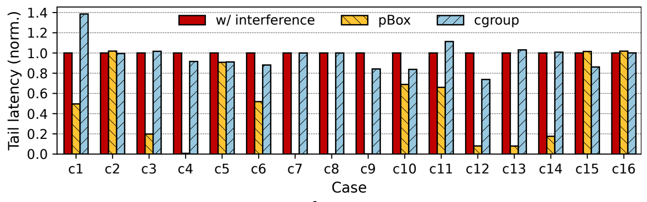

Figure 12. Normalized tail (95th percentile) latency for each case.

trace each activity's resource usage including lock and CPU, calculate the slowdown and load factor, and run Retro's BFAIR policy to throttle noisy requests. DARC provides request-level scheduling. We extend its request classifiers to support four request types for MySQL/PostgreSQL (Read, Write, Insert, Delete) and two request types for Apache/Varnish/Memcached (Post, Get). We implement a worker for each application to translate a PSP request into an app request.

Figure 11 shows the result. Cgroup reduces the interference for 3 cases by 33.6% on average and a max of 77.8%. In the remaining 13 cases, it makes the interference worse by -22.5% on average and worst by -94.6%. DARC helps 3 cases by 61.6% on average and a max of 90.8%. In the remaining 13 cases, it makes the performance worse for 535.8% and a max of 5716.5%. The reason is that DARC and cgroup limit a noisy activity's hardware resources, but the victim activities

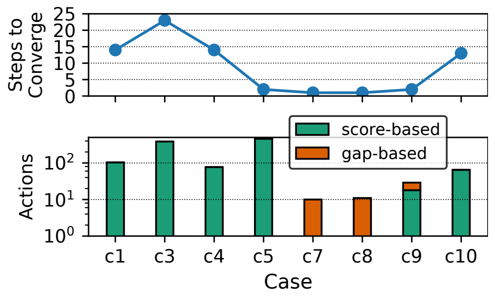

Figure 13. The number of penalty actions, interference level, and steps for the penalty length to converge to a fixed point. The score-based and gap-based adaptive policies are dynamically chosen.

are waiting for virtual resources from the noisy activity and need to wait longer. Retro helps 5 cases by 38.8% on average and a max of 57.8%. In the remaining 11 cases, it makes the performance worse by -48.6% on average and worst by -280.8%. The reason that Retro can help most cases is partially because we implemented its control points and throttling on top of the *pBox* abstraction and our *pBox* calls would avoid bad penalty timing. PARTIES helps 3 cases by 13.5% on average and a max of 28.6%. It makes 13 cases worse by -176.2% on average and worst by -716.7%.

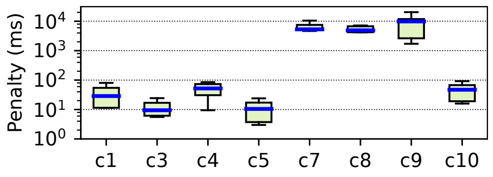

Figure 14. Penalty lengths *pBox* applied in interference mitigation.

	c1	c3	c4	c5	c6	c7	c8	c9	c10
Fixed (10 ms)	123.63	0.36	81.52	9.37	2.18	104.16	16.28	35.74	0.49
Fixed (100 ms)	49.50	0.56	29.72	3.65	2.41	13.71	8.15	13.42	0.62
Adaptive	1.28	0.35	1.24	3.29	2.64	0.65	0.70	1.28	0.56

Table 4. Average latency (ms) for nine evaluated cases using a fixed penalty versus using the default adaptive penalty design.

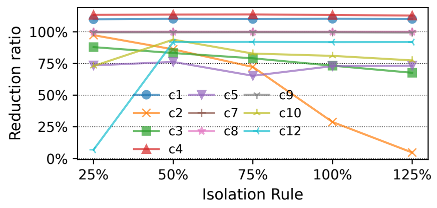

Figure 15. Interference reduction ratios for ten cases under different isolation rules from 25% to 125%. The default is 50%.

6.4 Penalty Action

To understand the internals of *pBox*'s mitigation, we measure the number of penalty actions in 8 cases. Figure 13 shows the result. In general, *pBox* takes more penalty actions under a high interference level (listed in Table 3). However, if the interference level is too high, fewer penalty actions may occur, because that can cause *pBox* to choose the gap-based adaptive policy, which increases the penalty length for each action and thus decreases the number of actions.

Figure 13 also shows the average number of steps that the adaptive penalty policy takes to converge (the penalty length reaches a fixed point). In cases where the gap-based policy is chosen (primarily), the convergence step is 10 times smaller than the step in cases where the score-based policy is chosen.

Figure 14 shows the penalty length distribution in the 8 cases. The cases choosing the gap-based policy have longer penalty lengths than the cases choosing the score-based policy.

6.5 Adaptive Penalty and Rule Sensitivity

We compare our adaptive penalty design (Section 4.4.2) with using fixed penalties of 10 ms and 100 ms. Table 4 shows that the adaptive penalty performs better for 7 out of 9 cases.

Users specify an isolation rule (goal) when creating *pBox* (Section 4.1). This setting can affect the detection and mitigation decisions. The experiment in Figure 11 uses the default 50%. We test 10 cases under different settings. Figure 15 shows the result. In general, a larger (more relaxed) isolation level can decrease the mitigation effectiveness.

The case c2 shows higher sensitivity to the rule settings. This is because the interference in this case is less severe—the interference level T_f is lower than two for c2 but greater than five for other cases. More relaxed isolation rules would cause fewer penalty actions, resulting in a lower reduction ratio.

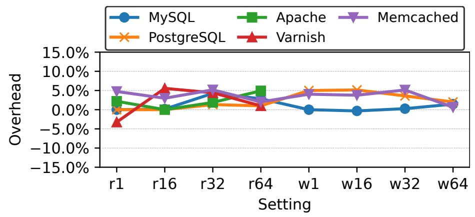

Figure 16. Overhead under different workload settings. r1 to r64: read-intensive workloads with one to 64 clients. w1 to w64: write-intensive workloads with one to 64 clients.

6.6 Overhead

We measure the end-to-end overhead of *pBox* to an application's performance in normal conditions. We use the same application versions and configurations as Section 6.2, but we run normal workloads instead, which are assumed to not introduce significant performance interference. Specifically, we generate OLTP read-only and write-only workloads for MySQL, PostgreSQL using sysbench [42], with an initial database of 64 tables and 1 K records per table. For Memcached, we generate read-intensive and write-intensive workloads based on Facebook's USR and VAR request distribution [5]. Each workload has eight settings with varying numbers of clients (Figure 16). We run Apache and Varnish under settings r1 to r64. The workload is serving HTML pages based on Varnish high-availability benchmark [77]. We run each setting for 90 s and compare the average latency with and without *pBox*.

Figure 16 shows the results. *pBox* introduces an overhead of 1.1%, 2.3%, 2.2%, 1.9%, and 3.6% on average to MySQL, PostgreSQL, Apache, Varnish, and Memcached, respectively. Interestingly, in several settings, *pBox* reduces the latency because of minor interference being mitigated.

The overhead does not significantly increase as the concurrency level increases. We use hashtables to store virtual resources. For each resource, we use a list to store the current waiters. Adding a *pBox* to this list has a constant cost. Removing a *pBox* and finding a victim *pBox* have costs linear to the number of waiters. If many *pBox*es are waiting on a virtual resource, it is likely that performance interference already occurs. In this case, the cost of finding a victim is shadowed by the gains of mitigating the interference.

We also measure the 99^{th} percentile latency. The overhead is 2.0%, 2.3%, 1.0%, 5.3%, and 2.6% on average to MySQL, PostgreSQL, Apache, Varnish, and Memcached, respectively.

6.7 Usage Effort

Table 5 shows the SLOC we add to the five applications for using *pBox*. MySQL's changes are the largest, mainly because it defines a number of custom virtual resource types that we need to cover. But the changes overall are small, especially considering the applications' large codebase sizes (Table 2).

Software	Inspected Functions	State Events		SLOC Added
		Manual	Detected	
MySQL	83	57	40 (70%)	192
PostgreSQL	71	40	44 (110%)	127
Apache	43	12	8 (66%)	71
Varnish	53	16	12 (75%)	77
Memcached	22	14	12 (85%)	70

Table 5. Functions we *inspected* to use *pBox*, state events we manually found to add `update_pbox` calls, and total SLOC added to the app code. *Detected* is the number of state events found by our analyzer.

Since we are not the application developers, we need to read the source code first. Table 5 shows the number of functions we inspected to determine the places for using *pBox*. It takes a graduate student a few days to complete the task for each application. Developers can likely use *pBox* more quickly.

We also test our static analyzer (Section 4.5). Table 5 reports the state events detected by the static analyzer. On average, the analyzer detects 81% of our manually found state events. For PostgreSQL, the analyzer detects four more points that we did not find during our manual porting.

For the remaining 19% of state events, they have the same heuristic as the others. The reason our static analyzer failed to identify them is that it only checks direct wrappers of waiting functions, but in these cases the callchain to a waiting function is deep. Additionally, some loop condition variable is the return value of a function call, and our current analyzer does not support checking if a returned variable is shared or not.

When an application evolves, if its activity boundaries and virtual resource usage code are changed (typically in a major upgrade), developers need to update the *pBox* calls accordingly. This is similar to how developers need to adjust the synchronization points when they make major changes to a multi-threaded program. Developers can re-run our static analyzer to assist them with updating the *pBox* calls.

6.8 Mistake Tolerance

We evaluate whether *pBox* can tolerate mistakes in using *pBox* APIs. We randomly remove 10% of the `update_pbox` calls in our *pBox*-version MySQL and rerun the experiment in Section 6.2. This process is repeated five times. On average, 4 cases (out of 5 cases) show positive mitigation, with an average interference reduction ratio of 92.1%, which is slightly lower than the result (93.9%) under correct usage.

7 Discussions

Kernel vs. user level. Where to implement *pBox* (application, library, and kernel) has performance, transparency, control, and flexibility trade-offs. Our current kernel-heavy implementation is motivated by several considerations:

- *pBox* is essentially an effort to improve the scheduling of application activities for performance isolation, which is an important property that the OS should provide to applications. Many works [26, 29, 38, 51, 53] have been implemented in the kernel to achieve performance isolation. However, they are insufficient to address the prevalent intra-application performance interference issues.
- Intra-app performance interference can occur due to system-level resources contention like `futex` and network queues. In Table 3, five cases are contending on such resources. Certain application virtual resources are proxies for system resources. For example, the table lock in MySQL is implemented using `pthread_mutex`, which relies on the `futex` syscall. The kernel-level *pBox* can directly modify the corresponding kernel code (e.g., futex implementation), allowing us to transparently trace state events without requiring developers to add `update_pbox` calls in application code.
- The timing of *pBox* actions is easier and more effective to enforce in the kernel. The actions' impact on the Linux scheduler is more predictable than user-level actions. The kernel-level implementation also allows future extensions using other scheduling actions, such as changing priorities.

However, for certain applications that use pure user-level queues for task management, it is beneficial to provide a library-heavy *pBox* implementation or design upcall APIs similar to scheduler activations [2].

Testing. To test whether the added *pBox* API calls are effective, developers can create performance benchmarks that reproduce past interference issues. Another testing strategy is to use a strict isolation goal in performance testing. Large software often experiences minor forms of intra-app performance interference. The *pBox* traces should show that some mitigation actions have been taken. Additionally, since the *pBox* APIs are designed to be simple, developers can easily add *pBox* code to missed code regions during or after a production performance issue, which will benefit future performance isolation. This flexibility enables iterative instrumentation.

Future Work. Like other performance interference mitigation work, *pBox* is a best-effort solution. It only reduces interference and does not guarantee that a given isolation goal will always be satisfied. How to provide strict performance isolation for large software is an open challenge. A related area of improvement is to provide a more rigorous analysis of the *pBox*'s actions, such as applying queuing theory [31]. *pBox* currently does not support distributed systems. Extensions to the tracing and detection algorithm as well as coordination on mitigation actions are needed for the support.

8 Related Work

Performance Interference. Performance interference has been extensively investigated in the contexts of cloud and virtualization environment [1, 4, 17, 41, 70, 71]. The interference occurs due to contentions on various shared hardware resources including CPU [27, 34, 47, 86], storage [45, 62, 78], memory [26, 49, 58, 83], and network [28, 69].

Numerous solutions [10, 11, 26, 34, 35, 47, 54, 58, 64, 68, 86] are proposed to mitigate interference by adjusting hardware resources. For instance, PerfIso [34] dynamically restricts the cores for batch jobs to protect the performance of latency-sensitive jobs. PARTIES [10] boots allocation of hardware resources for latency-critical services upon detecting QoS violations. Caladan [26] uses memory bandwidth and request processing times as the control signals to detect memory and CPU interference, and restricts CPU cores for antagonist jobs.

We focus on intra-application performance interference, which is caused by internal activities contending on application-level resources such as buffers or tickets. The contention can be invisible to existing solutions.

Fine-grained Resource Management. A long history of supporting fine-grained resource management exists in the context of real-time and multimedia operating systems [13, 24, 37, 55, 56]. Much of the work focuses on charging resource consumption to an application activity that is across the process or thread boundary. Similar efforts exist in general-purpose operating systems and software [7, 36, 46, 48, 53, 65]. A representative work is the resource container abstraction [7], which allows developers to limit an application activity's resource usage. It is modernized by Linux cgroup [53]. All these efforts still mainly target hardware resources, while *pBox* is about contention on virtual resources. Moreover, *pBox* focuses on cross-activity interference instead of managing each activity independently. It uses virtual-resource-aware scheduling to minimize the interference.

Retro [48] attributes the resource usage to different workflows and allows developers to write their own scheduling policies to control resource allocations. While Retro can trace some application resources (locks and thread pools), it mainly targets conventional interference due to multi-tenancy. *pBox* covers a wide variety of virtual resources. It does not target resource allocation, but instead focuses on fine-grained performance isolation and may take mitigation action at any time during an application activity's execution.

Server Overload Control. Applications may experience performance overload due to excessive requests. Solutions typically use admission control techniques [9, 11, 12, 20, 22, 30, 40, 79] that apply rate limiting on the client side or drop requests at the proxy or server side. Intra-application performance interference is an orthogonal problem. It can happen even when the server is not overloaded. *pBox* does not throttle requests in providing performance isolation.

Application-Specific Scheduling. Customizing scheduling based on an application's workload characteristics can greatly improve performance, thus motivating works to provide this capability [18, 33, 38, 39, 50, 59, 61]. For example, Syrup [39] allows developers to easily write application-specific scheduling policies. DARC [18] profiles application requests and leaves some cores idle when there are no short requests. *pBox* is orthogonal to these efforts. It is *not* a scheduler to allocate CPU and other hardware resources. It only takes action when an activity's isolation goal is in danger of being violated. Also, these solutions often assume independent requests, so hardware resources can be arbitrarily scheduled. But requests (and background tasks) involved in intra-app interference have dependencies on virtual resources, thus simply adjusting hardware resources do not help and can worsen the interference.

SLO Guarantees. Some projects target SLO enforcement in multi-tenancy. PSLO [45] enforces tail latency and throughput for consolidated VM storage by controlling I/O concurrency level and arrival rate for each VM. FIRM [63] uses machine learning methods to detect SLO violations in microservices, upon which it adjusts the hardware resource provisioning. *pBox* aims for fine-grained performance isolation. Overall SLO may not detect interference among application activities. Reacting after SLO violation can also be too late, because the contended virtual resources cannot be directly reclaimed.

Synchronization Optimization. Extensive efforts optimize locks and other synchronization primitives, *e.g.*, scalable spin locks [52], NUMA-aware locks [16], user-defined kernel locks [60]. Many intra-app interference issues are not simply due to poor synchronization or scalability bottlenecks. While locks often appear in them, the virtual resources that cause the interference are diverse and involve complex interactions among application activities. Thus, optimizing locks is insufficient. *pBox* focuses on end-to-end performance and isolation for an activity instead of an individual lock.

Performance Debugging. It is notoriously difficult to debug complex performance issues in large software. Many profilers and analyzers [6, 8, 15, 32, 73, 80, 81, 84] are therefore proposed to help developers with this task. pBox targets performance issues caused by interference among application activities and provides performance isolation at runtime. The log traces from pBox can provide useful insights for developers to understand a performance interference issue.

9 Conclusion

This paper explores pushing the performance isolation boundaries into an application. We propose an abstraction called *pBox*. *pBox* captures general state events about diverse virtual resources, detects imminent interference among application activities, and carefully chooses actions to achieve the performance isolation goal. We apply *pBox* on five large applications and evaluate it with 16 real-world intra-app interference issues. *pBox* significantly reduces the interference for most cases.

Acknowledgments

We thank our shepherd, Emmett Witchel, and the anonymous reviewers for their valuable and detailed feedback that improved our work. We thank CloudLab [21] for providing the environment for running our experiments. This work was supported in part by NSF grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757, and a Meta research award.

References

[1] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. ACM, 95–109.

[3] Andrew. 2017. How to keep one query from slowing down entire database? https://dba.stackexchange.com/questions/11920/how-to-keep-one-query-from-slowing-down-entire-database.

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-End Performance Isolation through Virtual Datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, 233–248.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. 40, 1 (jun 2012), 53–64.

[6] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (Hollywood, CA, USA). 307–320.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, 45–58.

[8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (San Francisco, CA, USA). 259–272.

[9] Huamin Chen and Prasant Mohapatra. 2002. Session-Based Overload Control in QoS-Aware Web Servers. In *Proceedings of The 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '02)*. IEEE Computer Society, 516–524.

[10] 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* (Providence, RI, USA) *(ASPLOS '19)*. ACM, 107–120.

[11] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 299–314.

[12] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2023. Protego: Overload Control for Applications with Unpredictable Lock Contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*.

[13] Raymond K Clark, E Douglas Jensen, and Franklin D Reynolds. 1992. An architectural overview of the Alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*. 27–28.

[14] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. 2001. *A Simple, Fast Dominance Algorithm*. Technical Report. Rice University.

[15] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. 2017. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, CA, USA). 641 – 652.

[16] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. CLoF: A Compositional Lock Framework for Multi-Level NUMA Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. ACM, 851–865.

[17] Christina Delimitrou and Christos Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC '13)*. 23–33.

[18] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. 621–637.

[19] MySQL Developers. 2023. Setting Account Resource Limits. https://dev.mysql.com/doc/refman/8.0/en/user-resources.html.

[20] Peter Druschel and Gaurav Banga. 1996. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (Seattle, Washington, USA) *(OSDI '96)*. ACM, 261–275.

[21] 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)*. USENIX Association, 1–14.

[22] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure as Interrupts for Highly Scalable Data-Parallel Programs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, 394–409.

[23] Tim Ferrell. 2010. How to throttle or prioritize a query in MySQL. https://stackoverflow.com/questions/3134350/how-to-throttle-or-prioritize-a-query-in-mysql.

[24] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (San Francisco, California) *(WTEC '94)*. USENIX Association, 9.

[25] The Apache Software Foundation. 2023. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[26] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 281–297.

[27] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying Effects of Shared on-Chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SoCC '11)*. ACM, Article 22, 14 pages.

[28] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, 1–14.

[29] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 401–417.

[30] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S.

Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, 168–183.

[31] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[32] Yigong Hu, Gongqi Huang, and Peng Huang. 2020. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Online). 719–734.

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

[34] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX Association, 519–532.

[35] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 567–584.

[36] Yuzhuo Jing and Peng Huang. 2022. Operating System Support for Safe and Efficient Auxiliary Execution. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. USENIX Association, 633–648.

[37] M. B. Jones, P. J. Leach, R. P. Draves, and . Iii Barrera, J. S. 1995. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) (HotOS '95)*. IEEE Computer Society, 12.

[38] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '19)*. USENIX Association, 345–359.

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

[40] Marios Kogias and Edouard Bugnion. 2020. Tail-Tolerance as a Systems Principle Not a Metric. In *4th Asia-Pacific Workshop on Networking* (Seoul, Republic of Korea) *(APNet '20)*. ACM, 16–22.

[41] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. 2007. An Analysis of Performance Interference Effects in Virtual Environments. In *2007 IEEE International Symposium on Performance Analysis of Systems Software* (San Jose, CA, USA). IEEE Computer Society, 200–209.

[42] Alexey Kopytov. 2021. Sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[43] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, 75–.

[44] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. ACM, Article 4, 14 pages.

[45] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. 2016. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. ACM, Article 28, 14 pages.

[46] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 49–64.

[47] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 450–462.

[48] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, 589–603.

[49] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. ACM, 248–259.

[50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, 399–413.

[51] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 1–18.

[52] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (feb 1991), 21–65.

[53] Paul Menage. 2004. Linux Control Group. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html.

[54] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. ACM, 80–88.

[55] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. 1994. Processor capacity reserves: operating system support for multimedia applications. In *1994 Proceedings of IEEE International Conference on Multimedia Computing and Systems*. 90–99.

[56] David Mosberger and Larry L. Peterson. 1996. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Seattle, Washington, USA) *(OSDI '96)*. ACM, 153–167.

[57] MYoussef. 2014. Hyperthreading and MySQL InnoDB Thread Concurrency Performance. https://dba.stackexchange.com/questions/81204/hyperthreading-mysql-innodb-thread-concurrency-performance.

[58] Jinyoung Oh and Youngjin Kwon. 2021. Persistent Memory Aware Performance Isolation with Dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. ACM, 97–105.

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th*

USENIX Conference on Networked Systems Design and Implementation (Boston, MA, USA) *(NSDI'19)*. USENIX Association, 361–377.

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

[61] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, 325–341.

[62] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments. In *2010 IEEE 3rd International Conference on Cloud Computing*. 51–58.

[63] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 805–825.

[64] Francisco Romero and Christina Delimitrou. 2018. Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. ACM, Article 19, 13 pages.

[65] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 1409–1427.

[66] ServerFault. 2017. How to deal with mysqldump and innodb_buffer_pool_size? https://serverfault.com/questions/852323/how-to-deal-with-mysqldump-and-innodb-buffer-pool-size.

[67] ServerFault. 2023. ServerFault. https://serverfault.com.

[68] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, Article 33, 17 pages.

[69] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI '11)*. USENIX Association, 309–322.

[70] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433.

[71] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI '12)*. USENIX Association, 349–362.

[72] Rajini Sivaram. 2017. Kafka proposal 124: Request rate quotas. https://cwiki.apache.org/confluence/display/KAFKA/KIP-124+-+Request+rate+quotas.

[73] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, OR, USA). 561–578.

[74] Alexey Stroganov, Laurynas Biveinis, and Vadim Tkachenko. 2016. Percona Server 5.7 performance improvements. https://www.percona.com/blog/2016/03/17/percona-server-5-7-performance-improvements.

[75] MySQL team at Facebook. 2014. Purge should remove intermediate rows between far snapshots. https://www.facebook.com/MySQLatFacebook/posts/10152610642241696.

[76] Vadim Tkachenko. 2006. Mess with innodb_thread_concurrency. https://www.percona.com/blog/2006/05/12/mess-with-innodb_thread_concurrency/.

[77] varnish software. 2016. PVARNISH HIGH AVAILABILITY BENCHMARKS. https://info.varnish-software.com/blog/varnish-high-availability-benchmarks.

[78] Matthew Wachs and Michael Abd-El-Malek. 2007. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies (FAST '07)*. USENIX Association.

[79] Matt Welsh and David Culler. 2003. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Seattle, WA) *(USITS '03)*. USENIX Association.

[80] Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang. 2023. Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. ACM, 1–17.

[81] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. 2021. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference*. 193–207.

[82] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (may 2008), 53 pages.

[83] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 112–125.

[84] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China). 67–76.

[85] Liran Zelkha. 2011. Database Isolation Levels And Their Effects On Performance And Scalability. http://highscalability.com/blog/2011/2/10/database-isolation-levels-and-their-effects-on-performance-a.html.

[86] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. ACM, 379–391.