Operating System Support for Safe and Efficient Auxiliary Execution

Yuzhuo Jing Peng Huang

Johns Hopkins University

Technical Report*

Abstract

Modern applications run various auxiliary tasks. These tasks gain high observability and control by executing in the application address space, but doing so causes safety and performance issues. Running them in a separate process offers strong isolation but poor observability and control.

In this paper, we propose special OS support for auxiliary tasks to address this challenge with an abstraction called *orbit*. An orbit task offers strong isolation. At the same time, it conveniently observes the main program with an automatic state synchronization feature. We implement the abstraction in the Linux kernel. We use orbit to port 7 existing auxiliary tasks and add one new task in 6 large applications. The evaluation shows that the orbit-version tasks have strong isolation with comparable performance of the original unsafe tasks.

1 Introduction

Applications in production frequently require maintenance to examine, optimize, debug, and control their execution. In the past, maintenance was primarily manual work done by administrators. Today, there are increasing needs for applications to self-manage and provide good observability. Indeed, many modern applications execute *auxiliary* tasks. These tasks are designed for various purposes including fault detection [18, 26, 36, 42], performance monitoring [20, 27, 34], online diagnosis [24], resource management [14, 30], *etc.*

For example, PostgreSQL users can enable a periodic maintenance operation called autovacuum [17] that removes dead rows and updates statistics; MySQL provides an option to run a deadlock detection task [29], which tries to detect transaction deadlocks and roll back a transaction to break a detected deadlock; HDFS server includes multiple daemon threads, such as a checkpointer that periodically wakes up to take a checkpoint of the namespace and saves the snapshot.

Essentially, the structure of applications splits into two logical realms of activities (Figure 1)—the *main* and the *auxiliaries*. Despite being peripheral, the latter tasks are important for the reliability and observability of production software.

*A shorter version of this paper appears in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22), July, 2022

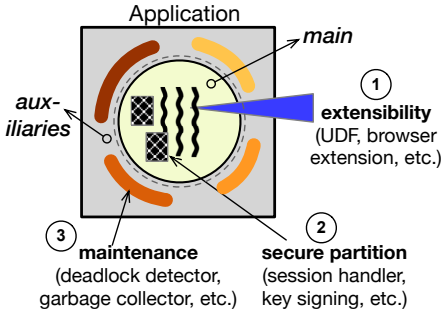

Figure 1: Three protection scenarios for modern applications. This paper focuses on ③.

At the implementation level, though, auxiliary tasks' execution is mixed with the main program's in the same address space, via direct function calls or as threads. Unfortunately, this choice means the auxiliary tasks can incur severe interference to the application's performance, due to unnecessary blocking and contention on CPU, memory, network, and other resources. In addition to costs, bugs in the auxiliary tasks can easily affect the application reliability, *e.g.*, a null-pointer bug inside a checker function can crash the whole process.

The alternative is to execute an auxiliary task externally in another process. This choice, however, would impose significant limitations on what can be observed and what can be changed. If the deadlock detector, for example, is run in a separate process, it would not be able to directly inspect the latest transactions or locks; even if it finds a deadlock it could not apply changes to mitigate the issue.

A fundamental problem is that existing OS abstractions for task execution—processes and threads—are designed for the main activities, but are unfit for auxiliary tasks. They force developers to either choose strong isolation but very limited observability and control (in a separate process), or high observability and control but little isolation (in a thread). In this paper, we advocate direct OS support for the trend of *auxiliary execution* to tackle this tension.

OS support for sub-process protection is not new. The systems and security communities have proposed various mechanisms [10, 12, 16, 23, 28, 39, 41, 46]. However, they are designed for two other different purposes. As illustrated in Figure 1, mechanisms such as SFI [41] are designed for *application extensibility* (①). That is, safely execute some *untrusted* third-party extension code, *e.g.*, browser extensions

and user-defined-functions in database queries. Another category of abstractions such as Wedge [10] and lwC [23] are designed for *secure partitioning* (②), *i.e.*, protecting some sensitive procedures in the main program, *e.g.*, session handler or key signing, in case the application is compromised.

These existing mechanisms are insufficient for the third protection scenario (③)—maintenance. The auxiliary tasks are written by the same developers and are trusted. They are also by nature interactive with the main program and need to constantly inspect the latest states of the main program. They often need to additionally alter the main program execution.

In this paper, we investigate this under-explored protection scenario. We summarize the common characteristics of auxiliary tasks, articulate the unique challenges of protecting such tasks, and advocate for special OS support to close this gap.

We then take the first step to propose a new OS abstraction called *orbit* for auxiliary execution. Orbit enables developers to conveniently add a wide range of auxiliary tasks that execute safely and efficiently while assisting the application.

Orbit has several unique features compared to existing sub-process abstractions such as threads, SFI, and lwC. An orbit is a first-class execution entity with a dedicated address space and is schedulable. Each orbit is bound with a main process but provides strong isolation: (i) if an orbit task is buggy and crashes, it does not affect the main process; (ii) orbit executes asynchronously and can be directly enforced with resource control, thus the main process is isolated from an auxiliary task's performance interference. At the same time, orbit provides high observability. Each *orbit*'s address space is mostly a mirror of the main program's. Thus, when the main process calls an orbit, the orbit can run the task functions with the latest main program states. To meet the need for some auxiliary task to change the main process, orbit provides controlled alteration to safely apply updates.

There are two challenges in designing orbit. First, isolation and observability are difficult to achieve together. Second, isolation is known to be costly. Since the main process often calls auxiliary tasks continuously, orbit can incur large performance slowdown to the main process. Optimizations such as using shared memory conflict with the goal of isolation.

To address the first challenge, we design a lightweight memory snapshotting solution that leverages the copy-on-write mechanism and provides *automatic state synchronization* from the main process' address space to orbit's address space whenever the main process calls the orbit task. To address the second challenge, our insight is that while an auxiliary task may inspect various state variables in the main program, the total size of the inspected state *at each invocation* is often a relatively small portion of the entire program state. Thus, we take a simple approach that coalesces only those state variables that an orbit task needs into what we call *orbit areas*. The kernel dynamically identifies the active memory pages in the orbit areas that an orbit invocation requires and only synchronizes these pages to the orbit side.

The lightweight memory snapshotting solution works at page granularity, which has the advantages of simplicity, robustness, and ease of integration with all mainstream OSes without depending on perfect instrumentations as in more complex techniques such as shadow memory. The disadvantage is that the page granularity incurs higher snapshot overhead due to write amplification (snapshot an entire page even if only one small object is changed) and often false sharing (write protection on shared COW pages). We design several optimizations including incremental snapshot, dynamic page mode selection, and delegate objects to reduce the cost.

We have implemented a prototype of orbit in the Linux kernel 5.4.91. To evaluate the generality of the orbit abstractions, we collect 7 auxiliary tasks from 6 large applications including MySQL, Apache, and Redis, and successfully port these tasks using orbit. We also use orbit to write a new auxiliary task for Apache. To demonstrate the isolation capability of orbit, we inject faults to the orbit version of the tasks. Some faults are directly based on real bugs in the task code. The experiments show that the applications are protected from the faults in all cases. We measure the cost of the isolation by comparing the end-to-end application performance. The orbit version applications only incur a median overhead of 3.3%.

In summary, this paper's main contributions are as follows:

- We identify an under-explored category in protection for auxiliary execution and summarize its characteristics.
- We design a new OS abstraction orbit to enable auxiliary tasks that have both strong isolation and high observability.
- We implement orbit in the Linux kernel and evaluate it on real-world auxiliary tasks in large applications.

The source code of orbit is publicly available at:

`https://github.com/OrderLab/orbit`

2 Motivation and Goals

2.1 Auxiliary Tasks

Modern applications often execute various auxiliary tasks designed for assisting reliability, performance, and security. A few typical categories of auxiliary tasks include:

- **Fault detection.** Many applications have checkers to detect faults dynamically. Examples include watchdogs [25] to catch gray failures [19], deadlock checkers, and GC pause detector. Some checkers are instrumented with compilers, such as sanitizers to detect memory leaks.
- **Performance monitor.** It is common for applications to have monitors that collect performance data. For instance, Redis includes a slow log monitor to record queries that take unusually long time.
- **Resource management.** Large applications run resource management routines. For example, Cassandra periodically runs compaction tasks to improve performance for future queries; it also runs a task to asynchronously remove stale records based on past delete requests.

```
const trx_t* check_and_resolve(lock_t* lock, trx_t* trx) {
 do {
  DeadlockChecker checker(trx, lock, mark_counter);
  victim_trx = checker.search();
  if (victim_trx != NULL && victim_trx != trx)
   checker.trx_rollback();
 } while (victim_trx != NULL && victim_trx != trx);
 return victim_trx;
}
```

Figure 2: Deadlock checker function in MySQL.

- **Recovery.** Some routines in an application are designed for assisting active recovery. HDFS continuously scans blocks and schedules tasks to reconstruct blocks with low redundancies. Databases also often employ checkpoint threads that flush modified pages and write checkpoint records.

The workflow of these tasks typically has three steps: (1) read program states; (2) perform inspection work; (3) take actions and modify some states. Depending on their goals, some tasks only read a few program states, while others may inspect lots of states. Some auxiliary tasks are relatively simple that execute synchronously with the main program, *e.g.*, control flow checks [8]. Others are long-running operations that usually execute asynchronously, *e.g.*, in a background thread. Our main focus in this work is the latter type of auxiliary tasks, since they often pose potential issues to the main program.

Note that some existing auxiliary tasks are written in their current forms, *not* because of their inherent nature, but often due to the lack of system support. For example, an existing detection task may execute synchronously, because otherwise the program state may be changed while the task is checking it. However, if an efficient mechanism exists to automatically snapshot the state to be checked, this task could be easily made asynchronous. We aim to provide the support that improves existing auxiliary tasks while enabling novel ones.

2.2 Example: MySQL Deadlock Checker

To make the discussion concrete, we use a representative auxiliary task, the MySQL deadlock checker, as the running example throughout the paper. Figure 2 shows its simplified code snippet. This task is invoked regularly in the main program. Specifically, in handling an update query, MySQL may need to lock a record; if the locking fails, the checking task is invoked. Each checking function invocation takes the blocked lock and the transaction as arguments.

Inside `check_and_resolve`, a deadlock checker instance is created, which runs a search algorithm to inspect the wait-for graph involving the `lock` and `trx` objects as well as other dependent variables. If the checker detects one potential deadlock, it will try to resolve the issue by choosing a victim transaction and rolling it back (modify the state `victim_trx`).

2.3 Safety and Performance Concerns

Developers usually write auxiliary tasks to execute inside the application process. While this choice makes it convenient for the tasks to assist and monitor the main program, their execution poses safety concerns because they execute in the main program's address space. A common issue is a buggy task accessing invalid memory, which crashes the entire application. In other scenarios, a buggy task may cause the main program to get stuck, *e.g.*, a low-priority data gathering thread blocks the high-priority tasks in a similar vein as the infamous Mars Pathfinder incident [35]. Or, the buggy task accidentally modifies some global variables and causes the main program to misbehave. Some issues occur indirectly because of the address space sharing. For example, a defect in HDFS creates too many `SafeModeMonitor` threads and causes the main program to fail with out of memory errors [4].

It might seem that crashing the main program when the auxiliary task is broken is acceptable for some critical auxiliary tasks. For example, since the deadlock detector is important for resolving deadlocks in transactions, if the detector has an invalid memory access, it might be reasonable to crash the main program. However, in practice, crashing the main program is usually too costly (unavailability and slow recovery) and often incurs unintended side effect (inconsistency and data loss), especially considering that the bugs are not from the main program. Alternatively, if we provide strong isolation for auxiliary tasks, we can decouple the fate of the main program from the fates of the auxiliary tasks, which will allow developers to make better choices. For instance, developers can implement a policy that if an auxiliary task dies, it will be automatically restarted and pick up the previous progress, without affecting the main program's execution.

Besides safety, auxiliary tasks can also incur interference to the main program's performance. For instance, we measure the MySQL performance with the deadlock detector task running. The result shows a 3.5%–79.5% drop in the query throughput. This issue was reported by users [1].

In summary, auxiliary tasks are designed to actively improve application reliability and performance, but paradoxically the shared-address-space execution model can cause them to hurt the main program.

2.4 Why Fork or Sandbox Is Insufficient?

To address the safety and performance concerns of auxiliary tasks, two potential alternatives exist: fork and sandbox.

Fork-based Execution Model In this approach, the application makes a `fork()` system call before an auxiliary task executes and switches to run the task functions in the child process. The separate address space provides strong memory isolation. In addition, the task has a copy of address space and thus can inspect any main program states easily. Once `fork()` completes, the main program can continue, while allowing the auxiliary task to execute asynchronously.

Unfortunately, there are several issues. First, the cost is substantial, which includes the creation of a heavy-weight execution entity, as well as the copying of an address space. Even with the copy-on-write optimization, the main program may modify many pages afterward and trigger excessive copying. Moreover, for auxiliary tasks that execute frequently, the

fork overhead will be incurred at each task invocation.

Besides overhead, with the auxiliary task running as a child process, it is difficult for the task to perform maintenance work that requires modifying the main program states. For instance, the MySQL checker can identify a victim transaction and perform a rollback, but the resolution only affects the child process and would not help the parent process.

Sandbox-based Execution Model Another solution is to execute an auxiliary task in a sandbox, which is well-suited to execute untrusted code, *e.g.*, browser renderer. A sandboxed process has reduced privileges in accessing resources including file systems and system calls, and may reside in a separate fault domain using SFI techniques [41].

However, auxiliary tasks are not untrusted codes that sandboxes are designed for. They are written by the application developers and are trusted. Their safety issues arise because of bugs or unintended side effects such as invalid memory access, infinite loops, using too much CPU, etc., rather than accessing unwanted system calls or files. A sandboxed process in a separate fault domain can access only the memory segment allocated to them. It thus gains little observability of the main program and cannot change the main program state.

RPCs or Shared Memory In principle, some aforementioned limitations can be circumvented using RPCs or shared memory. In practice, such workarounds are not favored by developers, because neither model matches with how developers write auxiliary tasks. Developers currently add auxiliary tasks directly in the application codebase and can easily refer to variables in the main program or invoke its functions. To use the RPC model, developers need to convert many variables and functions to be amenable to RPCs. Variables such as `lock` and `trx` in MySQL are difficult to marshal and unmarshal across calls. Frequent RPCs also add large overhead.

The shared memory model similarly requires cumbersome setup and coordination. In addition, the main process would have to wait until the auxiliary task finishes before continuing. Otherwise, the task would inspect inconsistent states. Another issue is that shared memory defeats the isolation purpose. An auxiliary task may need to access variables that scatter across the main program's address space. As a result, the main process may share a large portion of its address space, posing significant safety issues like a thread-based auxiliary task.

3 Orbit: OS Support For Auxiliary Executions

The aforementioned challenges are largely because existing OS abstractions for execution are designed for activities that have clear modularity and isolation boundaries. Auxiliary tasks are inherently interactive with the main program, but it is also desirable to isolate their faults and avoid interference. Developers are forced to choose either an abstraction that offers high observability but weak isolation (*e.g.*, thread), or one with strong isolation but low observability (*e.g.* process).

To address this gap, we propose direct OS support for auxiliary execution with a new abstraction called *orbit*. Orbit offers high observability of another execution entity, while providing strong isolation. Its end goal is to enable developers to create a variety of auxiliary tasks that assist applications in production to enhance the applications' reliability and performance.

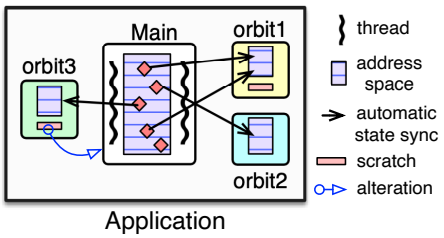

Figure 3: Multiple *orbit*s co-exist with the main program at runtime to provide observability and maintenance support.

3.1 Overview

An orbit task is a lightweight OS execution entity. Each task is bound to "watch" one *target* process. A process can have multiple orbit tasks as shown in Figure 3. They inspect different parts of the target's states for different maintenance purposes. Compared to existing abstractions, *orbit* has several major unique properties:

- **Strong Isolation.** Each orbit task has its own address space. Faults in an orbit would not jeopardize the main program or other orbit tasks. Most orbit tasks execute asynchronously without blocking the main program for a long time.
- **Convenient Programming Model.** The orbit abstraction preserves the current way of how developers write auxiliary tasks. Developers write the orbit task functions within the main program and directly refer to almost any state variables of the main program. They can also easily convert existing functions into orbits. This programming model is close to the thread model that developers are familiar with.
- **Automatic State Synchronization.** A defining characteristic of the orbit task's address space is that it is mostly a mirror of fragments in the target's address space. The fragments are those states that the orbit task needs to inspect. The underlying OS will *automatically* synchronize the specified states to the orbit address space in *one direction*, which occurs before each task invocation in the main program.
- **Controlled Alteration.** A regular orbit only observes the main program, while a privileged orbit is allowed to alter the main program state. However, it cannot change arbitrary state at arbitrary times. The modification has to be made using *scratch space* and well-defined interfaces.
- **First-class Entity.** Orbit tasks are first-class OS entities. They are schedulable like a normal process or thread. This property differs from existing sub-process abstractions such as SFI-based sandboxes and lightweight-context [23], which are subordinates to the main program and *not* schedulable. These abstractions typically have to execute synchronously. An orbit task can be also directly enforced with various limits such as CPU quota.

API	Description
`orbit *orbit_create(const char *name, orbit_entry entry, void* (*init)(void))`	create an orbit task with a name, an entry function, and an optional initialization function
`int orbit_destroy(orbit *ob)`	destroy the specified orbit task
`orbit_area *orbit_area_create(size_t init_size, orbit *ob)`	create an orbit memory area with an initial size
`void *orbit_alloc(orbit_area *area, size_t size)`	allocate an object of `size` from the orbit area
`long orbit_call(orbit *ob, size_t narea, orbit_area** areas, orbit_entry func_once, void *arg, size_t argsize)`	invokes a synchronous call to the orbit task function with the specific area(s) and arguments, blocks until task finishes
`orbit_future *orbit_call_async(orbit *ob, int flags, size_t narea, orbit_area** areas, orbit_entry func_once, ...)`	invokes an asynchronous call to the orbit task function, returns an `orbit_future` that can be later retrieved
`long pull_orbit(orbit_future *f, orbit_update *update)`	main program waits and retrieves update from orbit future `f`
`long orbit_push(orbit_update *update, orbit_future *f)`	orbit passes update to an existing orbit future `f`

Table 1: Main orbit APIs.

3.2 Design Challenges and Insight

There are two core challenges that we need to address. First, *how to enable orbit tasks to continuously inspect the main program states conveniently*, given that observability and isolation are difficult to achieve together? Second, *how to minimize the performance cost while providing strong isolation?* Isolation inevitably incurs cost. A straightforward design can incur excessive performance slowdowns. Optimizations that can potentially reduce costs, such as using shared memory, are often in conflict with the goal of fault isolation.

Our observations about the characteristics of typical auxiliary tasks reveal insight to address the challenges. While an auxiliary task may inspect various states in an execution, the total size of the inspected state *at each invocation* is often a relatively small portion of the entire program state. In addition, an auxiliary task often performs work incrementally: once the task inspects some state instance in one invocation, the task may not inspect that instance in the next invocation.

4 Orbit Designs

In this section, we describe the designs of the orbit abstraction and how to achieve the properties described in Section 3.

4.1 System Interfaces

The orbit abstraction is exposed through system calls accompanied by a user-level library. Table 1 shows the major APIs.

Developers create an orbit task in place in the application codebase using `orbit_create`, specifying the task entry function. The entry function pointer is defined as `long(*orbit_entry)(void *argbuf, void *store)`, which is similar to the entry function definition in `pthread_create`. However, the orbit entry function executes in a separate address space. This function is also only invoked later by the main program through explicit orbit calls. In other words, the orbit task invocation is decoupled from the orbit creation and can occur *repeatedly*. The `void *argbuf` points to a buffer in the orbit's address space, which is used later during each task invocation to hold the arguments. An optional initialization function can be passed to `orbit_create`. It is useful when some orbit task needs to allocate structure in its address space to keep bookkeeping information. The `orbit_create` returns an orbit handle for the main program to use in later invocations.

```
1  + struct orbit *dlc;
2  + struct orbit_area *area;
3
4  int mysqld_main() {
5  + dlc = orbit_create("dl_checker",check_and_resolve,NULL);
6  + area = orbit_area_create(4096);
7  }
8
9  lock_t* RecLock::lock_alloc(trx_t* trx) {
10   lock_t* lock;
11 - lock = (lock_t*) mem_heap_alloc(heap, sizeof(*lock));
12 + lock = (lock_t*) orbit_alloc(area, sizeof(*lock));
13   return lock;
14 }
15
16 dberr_t lock_rec_lock() {
17   if (status == LOCK_REC_FAIL) {
18 -   check_and_resolve(lock, m_trx);
19 +   dlc_args args = {lock, m_trx};
20 +   orbit_call(dlc, 1, &area, &args, sizeof(dlc_args));
21   }
22 }
```

Figure 4: Using `orbit` to enhance the MySQL deadlock detector. The core logic `check_and_resolve` in Figure 2 remains the same.

The orbit task invocations are done through either the synchronous `orbit_call` or asynchronous `orbit_call_async`. The latter would be particularly common to use. The semantics of the `orbit_call_async` guarantee that the states needed for the task are snapshotted before the API returns. As a result, the main program can continue executing other logic while the orbit task runs concurrently.

This API will return an `orbit_future f`. The main program can wait on `f` later through `orbit_future_get` when it requires knowing the update from the orbit task, just like the typical asynchronous programming models that developers are familiar with. Asynchronous orbit task execution along with the automatic state synchronization feature allows developers to exploit concurrency in the system.

Figure 4 shows an example of using orbit for the MySQL deadlock detector. The task core logic remains the same, but the invocation is split into two steps. Developers use `orbit_create` to create an orbit at the beginning (line 4), which specifies the entry function `check_and_resolve`. An orbit area is created. The allocations of the `lock` (line 12) and `trx` objects are changed to allocate from the orbit area. The

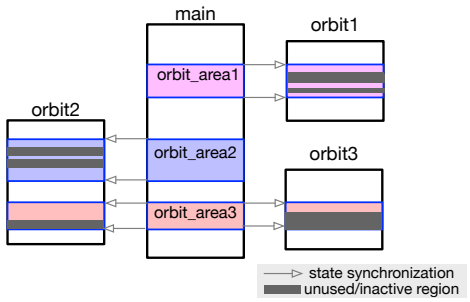

Figure 5: Orbit areas in the main program to be monitored.

original function call (line 19) is replaced with an `orbit_call` to invoke the previously created orbit with the `area` and arguments. Alternatively, developers can use `orbit_call_async` to asynchronously perform deadlock checking.

4.2 Managing Orbit

When a process creates an orbit using `orbit_create`, the kernel internally represents the orbit with a control block and records the target process the orbit is bound with. To avoid intrusive code changes to the Linux kernel function interfaces, we currently re-use the existing `task_struct` (with new fields and a subset of existing fields) to represent the orbit entity.

The main program maintains a `orbit_children` list in its `task_struct`, mapping orbit IDs to the orbit's `task_struct`. Each orbit maintains a `orbit_info` structure in its `task_struct`, that contains the basic execution states of orbit and a FIFO queue of orbit calls.

The kernel also allocates a dedicated address space for the orbit, which is initially kept to a minimum (mostly code pages of the main program). As a first-class OS abstraction, orbit is a schedulable entity and can be enforced with resource limits like a regular process. At the creation time, the orbit is in an *idle* state, waiting for the task invocations. If an orbit task is terminated (*e.g.*, because of its own bugs), it can be configured to be automatically restarted. In that case, after a restart, the orbit task will be reattached to the main program. The main program can explicitly destroy a specific orbit task.

4.3 Synchronizing States to Orbit

Each orbit executes in a separate address space but regularly inspects the state in the main program. To facilitate convenient inspection, the orbit abstraction provides a key feature of automatic synchronization for the referenced state. This automatic synchronization is one-way from the address space of the main to the orbit's. We propose a lightweight memory snapshotting solution for providing this feature.

Determining States One challenge is that an orbit task often inspects state variables that scatter across the main program's address space. Therefore, coarse-grained snapshotting would include too many unneeded objects in the snapshot memory regions, which would not only waste significant memory but also incur large overhead to the application. In addition, while the set of variables an orbit task inspects may be fixed and known at the static compilation time, the dynamic addresses and sizes of these variables can change over time. For example, the MySQL deadlock detector checks different `lock` and `txn` objects in different invocations.

To address this challenge, we take a simple approach that coalesces only those state variables that the orbit tasks need into what we call *orbit areas*. Orbit areas are fragments of the *main program*'s address space. Each orbit area is composed of contiguous virtual pages. An orbit's address space is mostly a mirror of orbit areas (Figure 5). The main program creates an orbit area through `orbit_area_create` with an initial size that is dynamically expandable. This API takes an `orbit` argument. If specified, the kernel will create a memory region in the orbit's address space and ensure it has the same virtual address of the orbit area in the main program before the API returns. Otherwise, this mapping mirroring will be done when an orbit later binds to an orbit area.

For the state variables that may be accessed by some orbit task, their allocation points need to be replaced to allocate from an orbit area through the `orbit_alloc` API. Similarly, these variables can be freed using the `orbit_free` API. The main program can still use these variables like before.

Taking a Snapshot Dynamically, when the main program makes a call to an orbit task function, the kernel identifies the memory pages in the orbit area that contain the variables the orbit task requires. Then the kernel updates the page table entries (PTEs) of these pages to mark them as write protected for copy-on-write (COW). The PTEs are also copied to orbit task's page table with write-protected bit set. For consistent snapshotting, the orbit call will return only after all needed mappings are updated. Afterward, as long as the main program and orbit task do not modify a page, no copying is incurred; otherwise, they will have separate copies of the page. Note that the above snapshotting process occurs on each orbit call, so the mappings in the orbit address space constantly change, but the orbit task is not re-created.

Concurrency To ensure safety under concurrency, the kernel acquires necessary locks (*e.g.*, `mmap_sem` in Linux) while accessing the PTEs in the main program and the orbit. In one orbit call, multiple pages may need to be snapshotted. To provide a consistent snapshot for multi-threaded applications, a conservative solution is to pause all the application threads so that these pages are not modified during the snapshotting. This pausing will incur a significant performance penalty.

We instead rely on application-level synchronization to handle this situation properly. Indeed, if the objects needed in an orbit call may be concurrently modified by some other thread, the application would add proper locks in the original call site to prevent race conditions. For example, the MySQL deadlock checker invocation (Figure 4) is already inside a critical section. Thus, when we port it to an orbit call, the snapshot of the `lock` and `m_trx` objects is consistent.

Locks are intentionally not shared between orbit and the main program, and thus orbit cannot directly alter the main program's lock states. It is possible that a complex orbit task

function acquires and releases locks during its execution. In such cases, acquiring locks can be moved upfront before the orbit call. From our experience of porting tasks that require synchronization (MySQL and Apache), we find that the original auxiliary functions only run within a single global critical section, which makes it straightforward to guarantee consistency. Also, since a consistent snapshot is obtained under a global lock, the orbit task can omit lock acquires in these cases, since it runs singled-threaded in another address space.

Concurrent Orbit Calls Another challenge is to handle state synchronization when some orbit tasks may be invoked concurrently. For example, the MySQL deadlock detector is invoked during request handling. Since MySQL uses multiple threads to handle concurrent requests, the main program may make another orbit call while the previous call is ongoing.

To address this challenge, the kernel maintains a task queue for each orbit (Section 4.4 will describe this part). After introducing the task queue mechanism, we need to ensure `orbit_call(_async)` preserves the semantics that the task invocation will get a consistent snapshot of relevant objects at the time of the API call. The kernel does so by marking COW for the *main program*'s PTEs of relevant orbit area pages, storing the marked PTEs, and returning. The stored PTEs will be installed to the orbit's page table *later* when the queued task executes. This works because, assume that the main program has modified some page in the orbit area while this invocation is in the task queue, COW will be triggered in the main program side and the main program will get a new page. The stored PTEs still point to the old physical page containing the data at the time of the invocation.

Design Choice Rationale Our memory snapshotting leverages the page protection and COW mechanism. Although snapshot at the page granularity can be costly, it integrates well in mainstream OSes and works reliably. Through several optimizations (Section 4.6), we can effectively reduce its performance costs. An alternative solution is to use fine-grained object-level shadow memory, which allocates shadow memory region, uses static analysis to identify and instrument memory writes to the target objects, and checkpoints these writes to the shadow memory region. We did not choose this approach for several reasons. First, the shadow memory consumes significant (often half) of the main program's address space, and because it is in the same address space, the isolation is weak. Second, there can be many objects repeatedly and unnecessarily checkpointed even when the orbit task does not need them. Third, handling concurrency is challenging. Lastly, it makes strong assumptions about the target application and instrumentation accuracies, which are fragile to apply to many complex applications.

4.4 Orbit Task Execution

When an orbit is created, it waits for the main program to make orbit calls. Implementing the task execution is non-trivial, because each call crosses two address spaces. In addition, the orbit may receive different styles of orbit calls, including concurrent calls. The kernel side needs to support these different styles together.

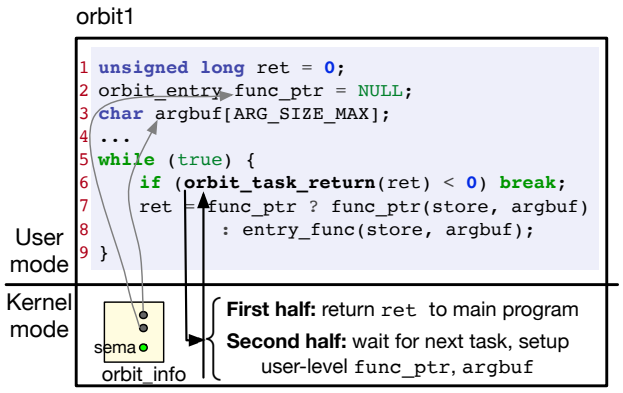

Figure 6: Orbit execution loop waiting for task invocations from main, facilitated by the helper system call `orbit_task_return`.

For supporting potential concurrent calls, the kernel maintains a task queue for each orbit. For each invocation from the main program, the kernel assigns a call id with an internal call struct and inserts it into the queue. The orbit task execution workflow processes the pending invocations in FIFO order. Serializing the task invocation processing makes it much simpler to ensure the correctness of the state synchronization.

To properly implement orbit task execution, we introduce a helper system call `orbit_task_return`. As Figure 6 shows, each orbit is a single-threaded worker executing this loop, and invokes this system call in each iteration. When trapped into the `orbit_task_return` syscall, the kernel knows which main program this orbit corresponds to by looking up the information in its `orbit_info`.

Internally, this kernel function consists of two halves. In the first half, it returns the return value of the *last* orbit call to the main program. Specifically, the kernel stores the passed `ret` value into an internal struct corresponding to the last orbit call, and then *signals* the thread that was executing the last orbit call and blocked waiting for the call to finish. If no orbit call has been made, this first half is skipped.

In the second half, the function waits for the next task from the main program. This is done by waiting on a semaphore in the orbit control block. Once the orbit tasks queue is non-empty, the `orbit_task_return` proceeds and dequeues an invocation. Recall that state snapshotting stores the marked PTEs (Section 4.3) in an array for the pending invocation. The kernel function at this point applies the snapshot by installing the PTEs to the orbit's page table. It then sets up the user-space `argbuf` and `func_ptr`, and returns.

The kernel setups the user-level `argbuf` by copying the orbit call arguments into it. The arguments are typically pointers (*e.g.*, `lock` and `m_trx` in Figure 4), thus only the address values are copied. The actual objects to be referenced in the task are in the orbit area. With the mirroring setup of the orbit area (Section 4.3), the addresses map to equivalent objects. The `func_ptr` is set to either the task entry function or the

function pointer specified in the pending `orbit_call`. The latter is particularly useful for an orbit to provide *query* functionalities. For example, if an orbit stores some bookkeeping information, the main program may want to query the orbit about this information occasionally. Finally, the orbit execution loop invokes the appropriate task function with the prepared `argbuf` (line 7 in Figure 6) *at the user level*.

The major task execution workflow described earlier applies to the asynchronous orbit calls as well. The `orbit_call_async` returns an `orbit_future`, which is a reference to the asynchronous task. The main program can later wait on this reference and retrieves updates from the completed asynchronous task, just like the typical asynchronous programming models that developers are familiar with.

4.5 Controlled State Alteration

A privileged orbit is allowed to modify the main program states. One solution is to identify pages in the orbit area that the orbit has modified in its private copies and transparently update the corresponding copies in the main program. The updates are restricted to states belonging to an orbit area. A complication arises if the main program also has since made modifications to some pages in an orbit area. Automatically merging the updates could introduce accidental changes.

To avoid introducing such accidental incorrectness, we instead use a more controlled alteration mechanism by exposing the `pull_orbit` and `orbit_push` system calls. Developers call the `orbit_push` API in the orbit task functions to explicitly decide which updates to push to the main program side. A corresponding call of `pull_orbit` in some main program function will retrieve the updates and explicitly apply the updates to the appropriate state variables. The `orbit_push` API supports pushing flexible data types including raw bytes.

A *scratch* space is backed by some memory region holding the data. The pushing is done efficiently by moving the PTEs of the scratch space pages in the orbit page table to the main program's page table. Besides data, `orbit_push` also supports pushing some operation (function pointer). This is useful if the maintenance operation is difficult to conduct in the orbit side, such as killing some main program's thread.

Example Figure 7 shows an example for the MySQL deadlock detector, which represents a relatively complex use case. Function `trx_rollback` creates a *scratch* `orbit_update` and then pushes a `TrxVersion` by calling `add_data`. This data can later be used to check whether the `victim` transaction is still alive. A following `add_modify` call records the modification of a single field. The next `add_operation` pushes a function with its argument, which will later be invoked in the main program side when the updates are applied and will signal the specified conditional variable. The function pointers are valid for both sides, since the code pages mapping are preserved. The updates are then sent in a batch by calling `orbit_push`.

The `handle_rollback` function then pulls updates from the future. If the task fails, the orbit task is recreated (omitted in the figure). When the main program retrieves an update, it applies the update if the transaction's `version` is still alive.

```
void trx_rollback(trx_t *victim) { // within orbit task
 orbit_update *scratch = orbit_update_create();
 orbit_update_add_data(scratch, &victim->version);
 victim->lock.cancel = true;
 orbit_update_add_modify(scratch, &victim->lock.cancel, true);
 orbit_update_add_operation(scratch, pthread_cond_signal,
    &trx->slot->condvar);
 ...
 orbit_push(scratch);
}
void handle_rollback(orbit_future *future) { // in main program
 orbit_update update;
 long ret = pull_orbit(future, &update);
 TrxVersion *version = orbit_update_first(update)->data;
 if (trx_is_alive(version))
  orbit_apply(update);
}
```

Figure 7: Controlled state alteration for MySQL deadlock detector.

4.6 Optimizations

We design several optimizations to further reduce the cost of our memory snapshotting. There are two main overhead sources: (1) iterate the PTEs for the active pages in an orbit area, update COW flags, and create mappings in the orbit's address space; (2) page faults when an orbit area is modified.

4.6.1 Incremental Snapshotting

Overhead source (1) is incurred upon each `orbit_call`. In addition, we tear down the orbit's mappings and reset the COW flags of relevant PTEs in the main program when the orbit runs finishes to avoid unnecessary page faults. For orbit areas that have many active pages, this overhead can be significant.

We introduce an incremental snapshotting optimization to reduce this overhead. We keep the mappings after an orbit run finishes. Upon the next `orbit_call`, we iterate through each remained PTE and check if it is the same as the main program's counterpart. If so, we keep it. Otherwise, we recreate the mapping or discard it if the orbit area page is no longer active. Thus, we only pay the mapping cost for the orbit area's pages that are modified by the main program since the last run. One caveat is that keeping the mappings may incur unnecessary page faults. This optimization helps when the main program is not intensively updating the orbit area. We allow developers to pass a flag in an `orbit_call` to indicate whether to enable this mode (keep the mappings).

A second part of this optimization is a region-based marking scheme that aims to reduce the cost of looping through each PTE in an orbit area. We track the PTEs by regions. Specifically, we maintain a bitmap for each range of 512 PTEs (one PMD entry) in the orbit area. A 64-bit bitmap partitions the 512 entries into 64 groups of 8 PTEs. Each bit represents whether the consecutive 8 PTEs have faulted since the last snapshot. During a page fault, the corresponding bit is set to 1. After a snapshot, the snapshotted groups's bits are set to 0. In this way, we can jump to the next group of PTEs that have changed by using bit-wise operation on the bitmap.

```
// allocate with normal malloc
struct trx_t {
  struct {
    ...
-   lock_t* wait_lock;
+   lock_t*& wait_lock;
    ...
  } lock;
+ trx_t_delegate *delegate();
};
```

(a) original full object

```
// allocate with orbit_alloc
struct trx_t_delegate {
  struct {
      lock_t* wait_lock;
  } lock;
};
```

(b) delegate object

Figure 8: Delegate object for the struct `trx_t` in MySQL.

```
// new constructors
trx_t::trx_t(trx_t_delegate *d) : lock(d) {}
trx_lock_t::trx_lock_t(trx_t_delegate *d)
 : wait_lock(d->lock.wait_lock) {}
// creating and binding delegate objects
void trx_init(trx_t *trx) {
 auto delegate = (trx_t_delegate *)orbit_alloc(area,
  sizeof(*trx_delegate));
 new(trx) trx_t(delegate);
}
```

Figure 9: Create and bind delegate object for `trx_t` in MySQL.

4.6.2 Dynamic Page Mode Choice

Overhead source (2) is inherent in the COW mechanism. This cost becomes significant when the orbit area pages are frequently updated by the main program. In this case, COW may perform worse than directly copying the page, which eliminates later page fault penalty to the main program. COW is effective if an orbit area page is infrequently updated.

We support page mode choice (COW or COPY) for an entire orbit area and each page in the orbit area. The former is specified by developers when creating an orbit area. The (likely) update-intensive objects can then be allocated from a COPY-mode orbit area, which will use copying during snapshot. For page-level mode choice, the kernel tracks the statistics of fault rate as `# of faults/# of snapshots` for each page. If the percentage exceeds a heuristic threshold of 30%, we determine the page mode as COPY. Besides, we also impose a limit of 32KB on the total size of COPY pages, and we choose the pages with the highest scores. This is used to prevent exhausting too much memory, and achieve a relatively balanced performance between COPY and COW (because copying large memory region is slower than snapshotting).

4.6.3 Delegate Objects for Large Structs

Complex applications may define large structs, while the states that an orbit is concerned with may be only a small subset of the fields in a large struct. If we allocate the entire large struct from the orbit area, it can incur unnecessary snapshot and page faults due to false sharing.

We use *delegate objects* to mitigate this issue. The basic strategy is to define a delegate struct for the large struct and keep only the fields that are needed in the orbit task functions. Then we allocate the delegate struct from an orbit area but preserve the normal allocation (*e.g.,* `malloc`) for the underlying large struct. Each delegate object has a one-to-one binding to its original struct. It is created at the same time of the original struct as an additional argument to its constructor. To connect the two structs, the relevant fields in the large struct are changed to reference types (*e.g.*, `int` to `int &`, `int *` to `int *&`), and the struct constructor is modified to bind the references to the delegate struct argument. The main program still uses these fields like before without changes.

Figures 8 and 9 show an example of defining and using delegate object for the `trx_t` struct in MySQL. After introducing this delegate object, the main program does not need to change its usages, *e.g.*, `trx->lock.wait_lock` still works. The orbit task function uses the delegate object from `trx->delegate()`.

In our ported systems, we pick those large structs whose total size of accessed fields is smaller than the size of the remaining fields as the target for optimization. Developers can have their own choices to determine what are large structs for delegate object optimization.

4.7 Compiler Support

Our current design requires replacing allocation points for needed state variables (Section 4.3). Some applications already use custom functions to allocate their main objects. In these cases, developers may only need to make minor changes in the custom allocation function to use `orbit_alloc`.

In other cases, developers may need to find individual allocation points and replace them. To help developers with this task, we build an analyzer on top of LLVM [22].

Given an entry function to be converted to an orbit task, *e.g.*, `check_and_resolve` in Figure 2, the analyzer runs forward data-flow analyses to locate all relevant definition and allocation points. Specifically, the analyzer first identifies heap allocation calls in the main program. For each call, it constructs a use graph with the return value variable as the root. Nodes in the use graph include both direct and indirect usage points of the root based on the standard *def-use* chain analysis.

After constructing the use graphs, the analyzer checks whether any use graph can reach the arguments in a callsite of the target function. If so, the allocation point associated with the use graph is included in the result. Besides arguments, the compiler also analyzes the non-local variables referenced in the target function body and leverages the use graphs to identify their allocation points. If no allocation points are found for an argument or non-local variable, the analyzer identifies the definition point (*e.g.*, it is a static global variable) using reaching definition analysis and includes it in the result.

Currently, the analyzer only outputs a list of candidate allocation or definition points. It does not replace these points with `orbit_alloc` automatically, although that is feasible.

5 Evaluation

Our evaluation aims to answer several major questions: (1) *Is orbit general to (re)write auxiliary tasks in complex applications?* (2) *Can orbit-based tasks provide strong isolation?* (3) *How much overhead does orbit incur for achieving isolation?*

No.	Application	Auxiliary Task	Source	Description
t1	MySQL	deadlock detector	port	Automatically detect transaction deadlocks & rollback transaction(s) to break deadlock
t2	Apache	proxy balancer	port	Load balancing to determine suitable proxy backend worker for request
t3	Apache	lock watchdog	new	Periodically check for long mutex lock waits and output notifications to the log
t4	Nginx	WebDAV PUT handler	port	File upload handler for WebDAV PUT requests
t5	Varnish	pool herder	port	Dynamically adjust thread pool sizes
t6	Redis	Slow log	port	A system to log queries that exceeded a specified execution time
t7	Redis	RDB persistence	port	Performs point-in-time snapshots of dataset at specified intervals
t8	LevelDB	background compaction	port	Compact sorted table files to maintain level size limit and improve performance

Table 2: Evaluated auxiliary tasks in six large software.

	small (32 MB)	medium (1 GB)	large (8 GB)
orbit	56.71 (2.22)	58.64 (2.97)	70.89 (2.17)
fork	270.20 (3.77)	7083.89 (36.20)	56194.83 (241.62)

Table 3: Mean latencies (in microseconds) of creating orbit versus process. Numbers in parentheses are standard deviations in 100 runs.

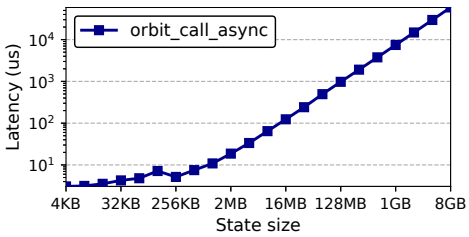

Figure 10: Orbit call latencies with different sizes of snapshot state.

5.1 Evaluation Setup

The experiments are performed on a 8-core Intel i7-11700 CPU (2.50GHz), 32GB memory and 1TB NVMe running Debian 10. We run all experiments using x86-64's default 4KB-sized pages, with huge page disabled.

5.2 Microbenchmark

We first evaluate the performance of creating and invoking orbit with microbenchmarks. We measure the orbit creation under different memory footprint settings of the main program. For a given memory setting, the benchmark program allocates the size, fills it with non-zero data to ensure the kernel actually allocated a physical page for it before running the measured action. It then calls `orbit_create` and measures the latency. We compare the orbit creation with `fork`.

Table 3 shows the result averaged over 100 runs, with test program pinned to a single core using `numactl`. The initial address space for orbit is minimum with mostly code and stack pages (Section 4.2). Compared to fork, this gives performance benefits for creating isolated address spaces even with a large memory footprint, as most unneeded data are not copied. When the main program has an 8 GB memory footprint, fork is 793× slower than creating an orbit.

We also measure the latency of `orbit_call_async`. Figure 10 shows the result averaged over 100 runs, with test program pinned to a single core using `numactl`. In general, orbit call time increases almost linearly with the size of orbit area, because it is dominated by the snapshotting cost. For example, making an orbit call with 32MB memory snapshotted takes 241.1 μs, which is comparable to the performance of forking a process with 32MB data shown in Table 3. Orbit call with 8GB snapshotted takes 59.0 ms, which is slightly higher than forking 8GB memory. This is due to the more complicated implementation of snapshotting, such as incremental snapshotting and support for several snapshotting modes.

5.3 Applying Orbit on Large Applications

To evaluate the generality of the orbit abstraction, we apply orbit on 6 large applications, MySQL, Apache, Nginx, Varnish, Redis and LevelDB, which have complex codebases and use diverse programming paradigms.

We use orbit to port 7 existing, representative auxiliary tasks in the applications (Table 2). They cover typical auxiliary tasks ranging from fault detection, debugging, resource management, and performance optimization. Two tasks, the Apache proxy balancer and the Nginx WebDAV handler, can be also considered main features. We evaluate them to test the boundaries of tasks that orbit can support. We successfully port all 7 tasks. We run each application's unit tests to verify the ported tasks preserve the original functionalities, even though the tasks now execute the separate address spaces.

We also use orbit to write a new auxiliary task, a lock watchdog, in Apache as an exercise. This task periodically checks if some thread in Apache is stuck and pinpoints the long-holding locks. We add a counter and held locks in thread-local storage. For every lock operation, the main program threads increment the counter, and the number of held locks. A background thread makes an `orbit_call` to the watchdog every second with all threads' counters and held locks. The orbit resets all counters. It also stores historic data of the last held locks and the number of iterations that there is no activity for each thread. When the orbit finds that some thread has has no activity over a threshold (60s), it `orbit_pushes` a return value to inform the main program, which triggers another `orbit_call` to the orbit's diagnosis function that finds the root cause. Figure 11 shows the watchdog thread function.

```
void watchdog_loop() {
 long next_op = WATCHDOG;
 while (true) {
  if (next_op == WATCHDOG)
   next_op = orbit_call(..., wd_areas, wd_func, ...);
  else if (next_op == DIAGNOSIS)
   next_op = orbit_call(..., diag_areas, diag_func, ...);
  ...
 }
}
```

Figure 11: The Apache lock watchdog thread

5.4 Fault Isolation

5.4.1 Fault Injection Testing

We evaluate the isolation capability of orbit by performing fault injection testing on all 8 auxiliary tasks. We inject null pointer deference faults at different times during a task's execution. In all cases, the system successfully isolates the faulty orbit without causing impact to the application and restarts the task gracefully to reattach to the running main process. In some systems, graceful failure handling is implemented by returning an application-specific error code after witnessing an error return code from `orbit_call`. For example, in Apache proxy handler, we return a `HTTP_SERVICE_UNAVAILABLE` after checking the orbit state in main program.

As a first-class OS entity, orbit also provides isolation of performance interference and resource overuse faults in auxiliary tasks. We inject two such faults in Redis slowlog (t6), and mitigate them with `cgroup`. We enforce a memory limit of 256 MB on the orbit task, and inject a memory allocation of 512 MB in orbit task, which this task would never use up. `Cgroup` triggers an OOM kill immediately when the task goes over the memory limit, and the main process gracefully restarts the orbit task. We also inject one CPU hogging for 10 seconds, and modify `cfs_quota` scheduler parameter with `cgroup` to bring CPU usage from taking up one whole core down to 10% of single-core CPU time shown in `top`.

For our newly implemented task in Apache (t3), we inject a long sleep right after one thread has acquired a lock. The watchdog immediately triggers a diagnosis once it finds the counter has not been updated for 60s. The diagnosis function pinpoints the thread ID that holds the lock, along with the location where the lock is acquired.

5.4.2 Real-world Bug Testing

We reproduced 4 *real-world* bug cases from MySQL, Apache, Redis and Nginx that involve the four tasks.

MySQL assertion failure We reproduced the MySQL Bug #28523042 [7]. This bug is introduced in MySQL 8.0 and adds incorrect assertions, which result in assertion failures. We reintroduced this bug into our orbit-enabled MySQL 5.7.31. For demonstration purposes, we modified some part of the expressions that touch the new variables in the 8.0 version, to make the backported code run on the 5.7.31 version.

When a deadlock occurs in the original buggy version, the whole MySQL server crashes, and all clients' connections are dropped. With the orbit-protected deadlock detector, even though the orbit task crashed, the MySQL server is still alive. After the default MySQL lock wait timeout is exceeded, one transaction is chosen as the victim, and all other transactions can continue to finish successfully.

Apache proxy balancer segfault We reproduced Apache Bug #59864 [6]. The user reported that under a proxy balancer configuration with a pair of unavailable fail-over backends pointing to each other, Apache entered infinite recursion when it searched for suitable backend, resulting in stack overflow. We isolate the backend selection in orbit, and successfully catch such failure. Instead of dropping connection, the main program now returns a more meaningful *"Temporary Unavailable"* message when it finds that orbit task has failed.

Furthermore, although web servers like Apache and Nginx often use fault-tolerance mechanisms like multi-process workers, such mechanisms cannot provide fault isolation for concurrent requests within the same worker. When one of the requests triggers a fault, all other connections to this worker also gets disconnected. This applies to both multi-threading (Apache) and event-driven architecture (Nginx) within one worker. Orbit further provides a finer level of isolation by isolating auxiliary tasks within one worker.

Nginx WebDAV segfault Nginx Bug #238 [5] was triggered when a custom WebDAV PUT (*i.e.*, file upload) user request did not include document body. The PUT handler assumes the request body pointer to have been allocated, and thus causes null pointer dereference. Similar to the previous Apache bug, the ported orbit version gracefully catches the failure and returns meaningful messages, while also preventing other requests in the same worker from disruption.

Redis Slowlog memory leak Although Redis uses single-threads for its request processing, its background threads can still cause issues. In case #4323 [2], a race condition happens when both slowlog and asynchronous lazy-free thread decrement a refcount, leading to neither of them freeing the object. Developer mitigated this issue by making a copy of the object. Our orbit implementation, on the other hand, transfers the object from snapshotted orbit area and designates resource management solely to the orbit's address space. Since orbit and the main process do not share the reference counter, race condition is eliminated in the first place.

5.5 Performance Overhead

We measure the end-to-end application performance impact with the orbit-based tasks. We choose application workloads that ensure the auxiliary tasks are triggered frequently.

For MySQL (t1), we run OLTP read-write test provided by the sysbench [3] benchmark tool with 16 clients. We run both Apache watchdog task (t3) and Varnish (t5) using `ab` with 1KB document length and 4 clients. Varnish web cache service uses a stock Nginx as backend. For Apache proxy balancer case (t2), we wrote a custom benchmark using `libcurl` to mix 90% non-proxy requests with 10% proxy requests with 4 clients because `ab` does not support mixed requests. Nginx

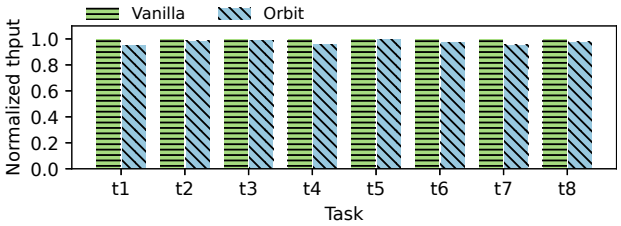

Figure 12: End-to-end application performance with the orbit-based (safe) tasks versus the original (unsafe) tasks.

Task	t1	t2	t3	t4	t5	t6	t7	t8
Calls/s	720.6	1627.2	1	1628.1	1	148.1	0.2	17.6

Table 4: Orbit call frequency in evaluated auxiliary tasks.

WebDAV (t4) benchmark is written in a similar way, with 10% WebDAV upload requests. We run both of the Redis tasks (t6, t7) with YCSB 95% read 5% write test using 32 threads, with either of the tasks enabled separately. We run LevelDB (t8) using a sequential-fill workload with LevelDB built-in benchmark tool to trigger compaction frequently.

Figure 12 shows the normalized throughput for the 8 cases. Most of the (safe) orbit tasks show comparable performance to vanilla (unsafe) tasks. The median overhead is 2.6%. The new task t3 in Apache is compared with the original Apache without our lock watchdog. It has the smallest overhead (0.3%). The largest overhead (4.83%) is the MySQL deadlock checker, which is acceptable considering the strong isolation.

Note that we intentionally choose workloads that stress test the orbit tasks. As Table 4 shows, all the tasks are frequently invoked. For example, the MySQL deadlock checker orbit is invoked 720 times per second. In practice, it may not be invoked this frequently. Thus its overhead would be much lower. Developers can also add sampling logic for orbit calls.

We also tested less intensive workloads. We reduced the write operations in MySQL (t1)'s OLTP workload, and changed the 90%/10% mix of t2 and t4 to 99%/1% mix. Task t1 and t2 only incur 2.0% and 1.2% overhead, respectively, while t4 has a negligible overhead of 0.18%.

For the MySQL deadlock detector, we implemented a fork version by creating a fork on each invocation to `check_and_resolve`. However, we did not implement IPC to pass results back to the main process, but if implemented, the fork-based performance would become even worse. In comparison, the orbit version has full functionality of pushing updates. We compare the MySQL performance under the three versions of detector: vanilla, fork-based, and orbit-based, using a user workload [1] under 8 threads. Figure 13 shows the result. The orbit version is slightly faster than the vanilla, but 9× faster than the fork-based version. For the orbit version we also compare the performance difference using the synchronous `orbit_call` versus using `orbit_call_async` under 16 threads, as shown in Figure 14.

	Throughput	**Latency**	**Orbit area**	**FPQ**	**TRX size**
No-opt.	1728.0 QPS	340.5 μs	25.7 MB	11.70	912 bytes
Delegate	3308.1 QPS	39.3 μs	1.0 MB	6.91	104 bytes
Changes	+91.4%	-88.5%	-96.1%	-40.9%	-88.6%

Table 5: Optimization effect of delegate object technique. (FPQ stands for page faults per query)

Task	t1	t2	t3	t4	t5	t6	t7	t8
Orbit area	828	20	8	4	8	268	80,644	240
Percentage	0.33	0.40	0.12	0.12	0.001	1.6	76.9	0.65

Table 6: Snapshot sizes (KB) in evaluated auxiliary tasks and their relative percentages (%) of the main program memory footprint.

5.6 Effectiveness of Optimizations

Incremental snapshotting We show the effect of incremental snapshotting by gradually allocating new objects in the orbit area and making orbit calls. We measure orbit call latencies with area sizes from 2 to 256 MB with an increment of 2 MB. Figure 15 shows the result averaged over 20 runs.

Without the optimization, the kernel wastes most cycles walking all the unchanged PTEs and thus requires longer latency. With the optimization, the new data that needs to be snapshotted in every call is a constant (2 MB). For an orbit area of 256 MB, the optimization reduces the latency by 44×.

Delegate Objects We use delegate object technique to minimize states size during snapshots, while also reduce unnecessary page faults due to main process memory writes to the other fields that orbit task does not use.

In the MySQL deadlock detector, we applied delegate object technique to transaction type `trx_t`, lock type `lock_t`, and lock information `lock_sys`. We observe that identifying such optimization opportunities is straightforward. For example, the `trx_t` is 70-field struct with only 4 fields being used in the orbit task, which is clearly an optimization target.

We run the user workload [1] with 16 clients on a 8-core vCPU QEMU VM and compare the throughput, latency, orbit area size, and average page faults per query. Table 5 shows the results. The optimization improves average throughput by 91%, and the orbit call latency to be 7.7× shorter. The total number of page faults throughout the run increases because the throughput also improves, but on average, the number of page faults each request incurs is reduced by 40.9%. In orbit calls, 96.1% of unneeded memory is saved from snapshots. In particular, the delegate object size for `trx_t` is only 11% of the original transaction structure.

5.7 Memory Footprint

Orbit provides efficient snapshotting because orbit only snapshots on necessary data for auxiliary task. We measure the average memory footprint of orbit area that was snapshotted during orbit calls. Table 6 shows the snapshot sizes along with their percentages of the main process's memory footprint. Among the ported tasks, 6 out of 8 allocate less than 1% of process data in orbit area. Redis RDB takes snapshot on its

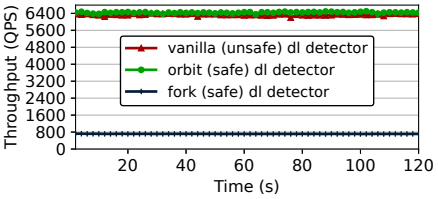

Figure 13: MySQL deadlock detector vanilla versus the orbit-based and fork-based version.

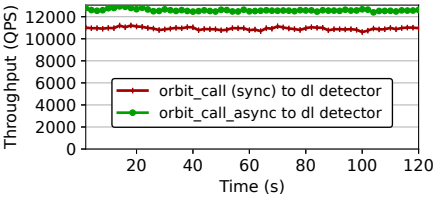

Figure 14: MySQL performance under 16 threads with sync. and async. orbit calls.

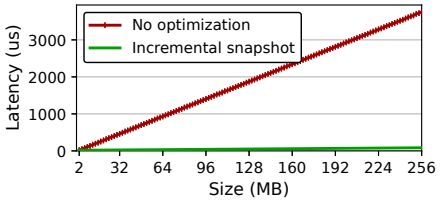

Figure 15: Orbit incremental snapshotting latency on a growing allocation size.

Task	t1	t2	t4	t5	t6	t7	Total
Manual port	7	16	7	3	11	12	56
Compiler	7	56	44	3	20	65	195
Common	5	8	5	1	9	11	39

Table 7: Allocation points in our manual port and compiler result.

key-value dictionary that dominates the memory usage, and thus require the largest portion of memory to be snapshotted.

5.8 Usage Effort

We count the lines of code changes we make to applications in porting the 7 existing auxiliary tasks. The changes include (1) replacing the allocation and free points with orbit allocations; (2) making orbit calls, pushing updates, and applying updates.

The combined changes for (1) range from 40 to 158 lines with a median of 115 lines. The Redis RDB task requires the most changes. We modified some application functions that create certain data structures to provide two versions (one for regular code paths, another for code paths to the orbit task) to avoid putting many unneeded objects in the orbit area. These modifications involved either duplicating the original function or changing its interface. The combined changes for (2) range from 45 to 272 lines with a median of 96 lines.

Our analyzer (Section 4.7) was developed after and motivated by our manual porting effort. We apply it on 6 of the evaluated tasks. The new implementation (t3) case has 0 original allocation points, thus it does not apply. The tool cannot analyze allocations in C++ STL container accurately due to its limited support for STL's complicated internal allocation implementation, thus t8 is excluded.

Table 7 shows the result of manually ported allocation points, detected points and the common ones between the two. From all 56 ported allocation points, our compiler detects 39 of them (70%). The detected points include ported, unported correct points, and false points. For the tasks that have larger number of detected but un-ported points (such as t7), we observe that most of these detected points are correct. They are missed from porting because our workload does not exercise those functionalities. There are also a few cases missing from detection because of unexpected corner cases. For example, a variable in Varnish (t5) used by the auxiliary task was directly allocated on stack instead of using allocator.

6 Discussions and Limitations

As a new abstraction support for auxiliary tasks, our current orbit design has several limitations.

State synchronization Our state synchronization mechanism works at the page granularity, which can incur unnecessary snapshot costs and page faults. Fine-grained object-level snapshotting is feasible but heavily depends on accurate static analysis and instrumentation. We plan to explore potential hybrid solutions that have the advantages of both approaches.

Observable states Our design only considers observing memory states, but not other system states such as file states. Those states would be more complicated to coordinate as they involve kernel and library buffer and position pointer. Creating file snapshots will require a different technique. The tasks we ported are relatively modular and self-contained. For example, our ported checkpointing tasks (Redis RDB, LevelDB compaction) require file operations, but they can create, write, close, and move files within the same orbit context, without the need to share file descriptors with the main program.

Code changes and compiler support We currently require developers to replace the allocation points of needed state variables. For some tasks, a relatively large number of places may need to be replaced. Our future work plans to leverage lightweight memory tracing [31] to dynamically identify the state variables and minimize the code changes.

The analysis in our compiler support for assisting developers to use orbit is basic. Although it supports field-sensitive pointer analysis, it can still miss corner-case allocation points. Developers need to manually find these points. Furthermore, our implementation of *def-use* chain analysis is not accurate enough to determine complex data flow, and thus will yield a handful of false positives. We will enhance the compiler support to enable fully automated porting for developers.

Comparison of programming difficulty Compared to programming with threads, using orbit requires the additional effort to properly change some allocation points. However, although developers do not need to change allocations when using threads, they still need clear knowledge of all the global variables that will be accessed in the thread, and ensure proper synchronizations for them. Thus, developers likely already have some knowledge about the allocation points of these variables. In addition, some of the synchronization would become unnecessary when using orbit. Therefore, the programming overall would be comparable.

Compared to the RPC model, orbit allows developers to write task functions in the same application codebase and directly refer to existing variables and functions. Unlike RPCs

that require code changes to enable object marshalling and unmarshalling, which are difficult for complex objects like transactions and locks, using orbit does not require such changes. With the mirroring orbit area, orbit calls directly access needed objects when crossing the address spaces.

Tolerance of bugs Orbit aims to protect the main program from issues in the auxiliary task execution. It tolerates common bugs such as memory errors in the auxiliary task functions, as well as bugs in the main program that pass bad (or corrupt) values to the auxiliary tasks.

It does not prevent an auxiliary task from sending an incorrect update back to the main program and cause the main program to malfunction. But the orbit abstraction encourages modularization for auxiliary execution, *i.e.*, an orbit task performs most of its operations in a separate address space before pushing updates back. This modularization minimizes the time window for the main program to see bad values and increases the chance that the orbit task itself encounters issues (*e.g.*, dereferencing a bad pointer) before the main program does, which still achieves protection. This is also one reason we choose to provide one-way automatic state synchronization (Section 4.3) with controlled state alteration, instead of a transparent, eager bidirectional state synchronization.

Auxiliary versus main tasks Determining whether a task is auxiliary or main can be subjective. While orbit is designed for auxiliary tasks, it does not require a clear-cut distinction—developers can use it to execute some tasks that they consider as main features for achieving strong isolation. We demonstrate this usage in the evaluation with two cases (t2 and t4).

7 Related Work

There is a wealth of work on protection and fault isolation. They vary widely in their target scenarios (OS extensibility, application extensions, sensitive code, *etc.*), goals (reliability, security, *etc.*), and approaches (software, hardware, hybrid). Our work is complementary to the existing efforts and targets a different, emerging protection scenario—auxiliary tasks in modern applications. Our proposed orbit abstraction aims to provide strong isolation for auxiliary tasks, while also achieving high observability and convenient usage.

SFI [41] is a software isolation technique that restricts the memory accesses of untrusted code in an application by rewriting the application binary. XFI [16] similarly uses binary rewriting to instrument software guards to check memory accesses. Extensive work has followed up this direction, such as NaCl [46] and RLBox [28]. As Section 2.4 elaborates, the sandbox model is not well suited for auxiliary tasks.

Several sub-process OS abstractions [10, 12, 23] provide secure partitioning in applications. They generally use private memory for executing sensitive code to ensure security. Wedge [10] provides the *sthread* primitive to partition an application into compartments and a scheme to tag memory regions and define access rights for the tags. Shreds [12] provides a segment of an execution unit called *shred* and relies on the ARM memory domains hardware feature to provide a private memory pool for each shred. Lightweight context (*lwC*) [23] creates a separate address space for each *lwC* in an application and allows a process to switch to some *lwC* when executing sensitive code. These abstractions typically get executed synchronously and are not independently schedulable.

Determinator OS [9] provides a private workspace model for deterministic parallelism. It runs user code in *spaces* and relies on processes to explicitly synchronize the spaces. Orbit provides automatic, fine-grained state address space synchronization between orbit and the main program. An orbit also has richer features due to its completely different design purpose. SpaceJMP [15] allows a process to define multiple address spaces and switch between address spaces, but with a main goal of enabling applications to use more physical memory rather than fault isolation.

Memory checkpointing takes snapshots of a running program's memory for debugging, failure recovery, quick initialization, *etc.* [11, 13, 21, 45] The checkpoint techniques usually rely on the copy-on-write (COW) mechanism through fork [32, 33, 37] or `mprotect`. On-demand-fork [47] optimizes the fork performance by extending COW to page tables. Orbit synchronizes only needed objects in the orbit areas. Lightweight memory checkpointing [40] uses shadow memory to checkpoint at object granularity. While it is more fine-grained than the page-level COW, shadow memory has several disadvantages for our scenario as described in Section 4.3. Overall, we focus on designing a complete OS abstraction for the isolation of auxiliary tasks. Our work is complementary to existing solutions and can benefit from their optimizations.

Protection schemes are also extensively explored in the context of OS extensibility. To name a few, Nooks [38] provides isolation of device drivers by executing them in different protection domains and using Extension Procedure Call (XPC) for control transfer; Mondrian memory protection (MMP) [43, 44] provides fine-grained protection by using hardware extensions and permission tables.

8 Conclusion

We discuss the trend of auxiliary tasks in applications and the lack of system support for providing safe and efficient execution for these tasks. We propose a new OS abstraction `orbit` to address the gap. Orbit offers high observability and flexible control, while providing strong isolation and efficiency. We evaluate orbit on 8 auxiliary tasks from 6 large applications. The applications achieve enhanced safety with the orbit tasks, and only incur a median of 3.3% performance overhead.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd Gerd Zellweger for their valuable feedback. We thank Shreyas Aiyar for his contribution to the orbit compiler. This work was supported in part by NSF grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757.

References

[1] InnoDB deadlock detection is CPU intensive with many locks on a single row. https://bugs.mysql.com/bug.php?id=49047.

[2] Redis 4.x lazyfree: memory leak may happen when free slowlog entry. https://github.com/redis/redis/issues/4323.

[3] Sysbench. https://github.com/akopytov/sysbench.

[4] Too many safemode monitor threads being created in the standby namenode causing it to fail with out of memory error. https://issues.apache.org/jira/browse/HDFS-5140.

[5] Segfault in DAV module during PUT processing. https://trac.nginx.org/nginx/ticket/238, 2012.

[6] Bug 59864 - segfault when using route-redirect pairs and both servers are disabled/in error mode. https://bz.apache.org/bugzilla/show_bug.cgi?id=59864, 2016.

[7] Bug 28523042 - innodb: assertion failure: lock0lock.cc:7034 in deadlockchecker::search. https://github.com/mysql/mysql-server/commit/97c49a66cea30c96ebc48129f3c4d59ac7a7c913, 2018.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, Alexandria, VA, USA, 2005.

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, Vancouver, BC, Canada, Oct. 2010.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, page 309–322, San Francisco, California, 2008.

[11] E. Bugnion, V. Chipounov, and G. Candea. Lightweight snapshots and system-level backtracking. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 23, Santa Ana Pueblo, New Mexcio, 2013.

[12] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[13] G. Cox, Z. Yan, A. Bhattacharjee, and V. Ganapathy. Secure, consistent, and high-performance memory snapshotting. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 236–247, Tempe, AZ, USA, 2018.

[14] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, FAST '21, pages 33–49. USENIX Association, Feb. 2021.

[15] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan. SpaceJMP: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 353–368, Atlanta, Georgia, USA, 2016.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 75–88, Seattle, Washington, 2006.

[17] P. G. D. Group. PostgreSQL's routine vacuuming. https://www.postgresql.org/docs/current/routine-vacuuming.html, 2022.

[18] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[19] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, 2017.

[20] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu. PerfGuard: Binary-centric application performance monitoring in production environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 595–606, Seattle, WA, USA, 2016.

[21] P. F. Klemperer, H. Y. Jeon, B. D. Payne, and J. C. Hoe. High-performance memory snapshotting for real-time, consistent, hypervisor-based monitors. *IEEE Transactions on Dependable and Secure Computing*, 17(3):518–535, 2020.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Palo Alto, California, 2004.

[23] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 49–64, Savannah, GA, USA, 2016.

[24] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu. iReplayer: In-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 344–358, Philadelphia, PA, USA, 2018.

[25] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS '19, Bertinoro, Italy, May 2019.

[26] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems*

Design and Implementation, NSDI '20. USENIX, February 2020.

[27] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, 2015.

[28] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium*, USENIX Security '20, pages 699–716, Aug. 2020.

[29] Oracle. MySQL's deadlock detection. https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html, 2022.

[30] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 13–26, Nuremberg, Germany, 2009.

[31] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC '13, pages 115–126, San Jose, CA, USA, June 2013.

[32] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the 1995 USENIX Technical Conference*, USENIX ATC '95, New Orleans, LA, USA, Jan. 1995.

[33] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 235–248, Brighton, United Kingdom, Oct. 2005.

[34] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, page 107–120, Hollywood, CA, USA, 2012.

[35] G. E. Reeves. What really happened on Mars? http://hdl.handle.net/2014/19020, February 1998.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[37] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, USENIX ATC '04, Boston, MA, USA, June 2004.

[38] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 207–222, Bolton Landing, NY, USA, Oct. 2003.

[39] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security '19, pages 1221–1238, Santa Clara, CA, USA, Aug. 2019.

[40] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum. Lightweight memory checkpointing. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 474–484, Rio de Janeiro, Brazil, June 2015.

[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 203–216, Asheville, North Carolina, USA, Dec. 1993.

[42] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 281–294, San Diego, California, 2008.

[43] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304–316, San Jose, California, 2002.

[44] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, Brighton, United Kingdom, 2005.

[45] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328, Dallas, Texas, USA, 2017.

[46] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 IEEE Symposium on Security and Privacy (SP)*, 2009.

[47] K. Zhao, S. Gong, and P. Fonseca. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555, Online Event, United Kingdom, 2021.