Operating System Support for Safe and Efficient Auxiliary Execution

Yuzhuo Jing, Peng Huang
Johns Hopkins University

OSDI ’22

Auxiliary tasks increasingly common

Deadlock detector

Fault detection

RDB checkpointing

Recovery

Autovacuum

Resource management

Auxiliary tasks are not part of core business logic but important for app reliability and performance

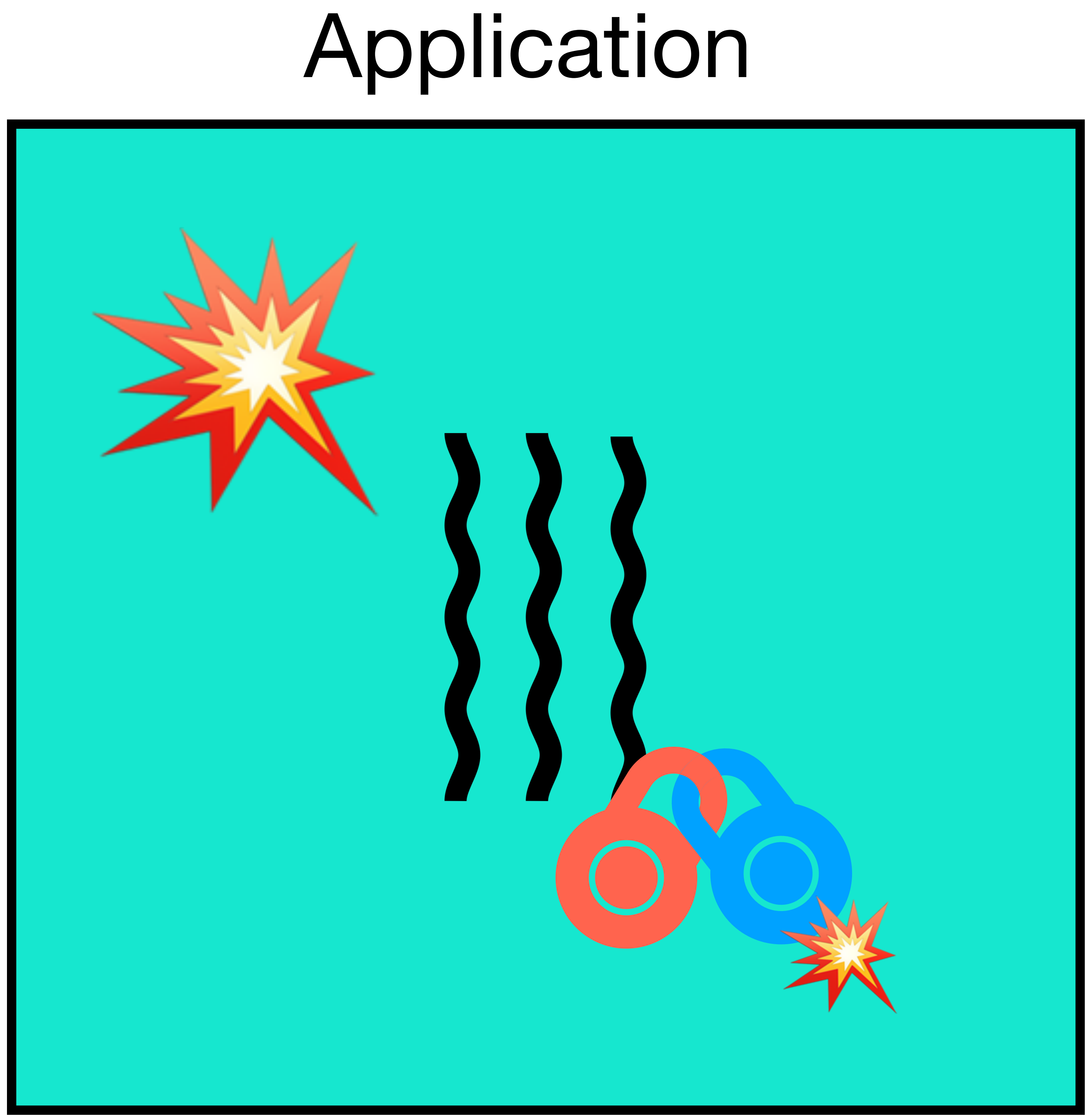
Typical characteristics of auxiliary tasks

…

1. Regularly invoked, often long-running
2. Read main program’s latest state
3. Perform inspection work
4. Take some actions
5. Optionally modify main program state

Current practice of auxiliary execution

Practice 1: running in the same address space

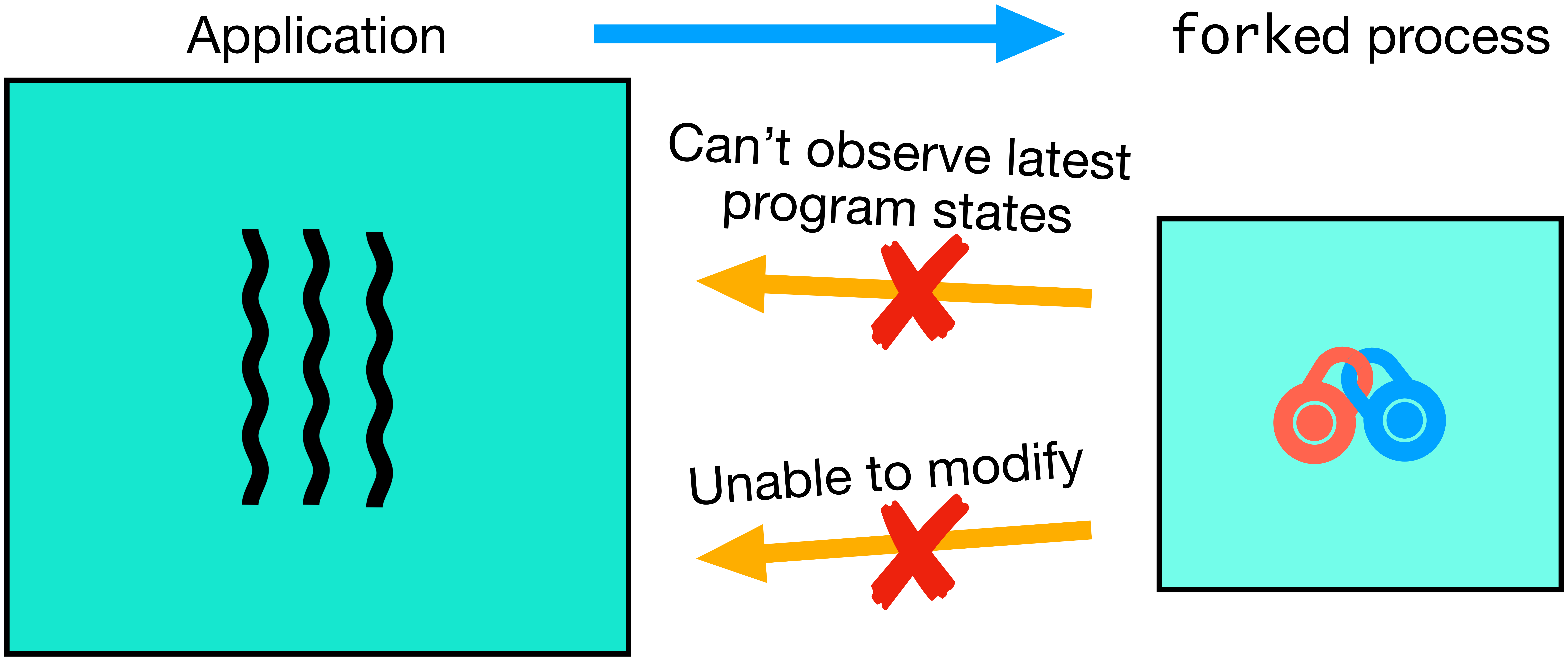


Problems:

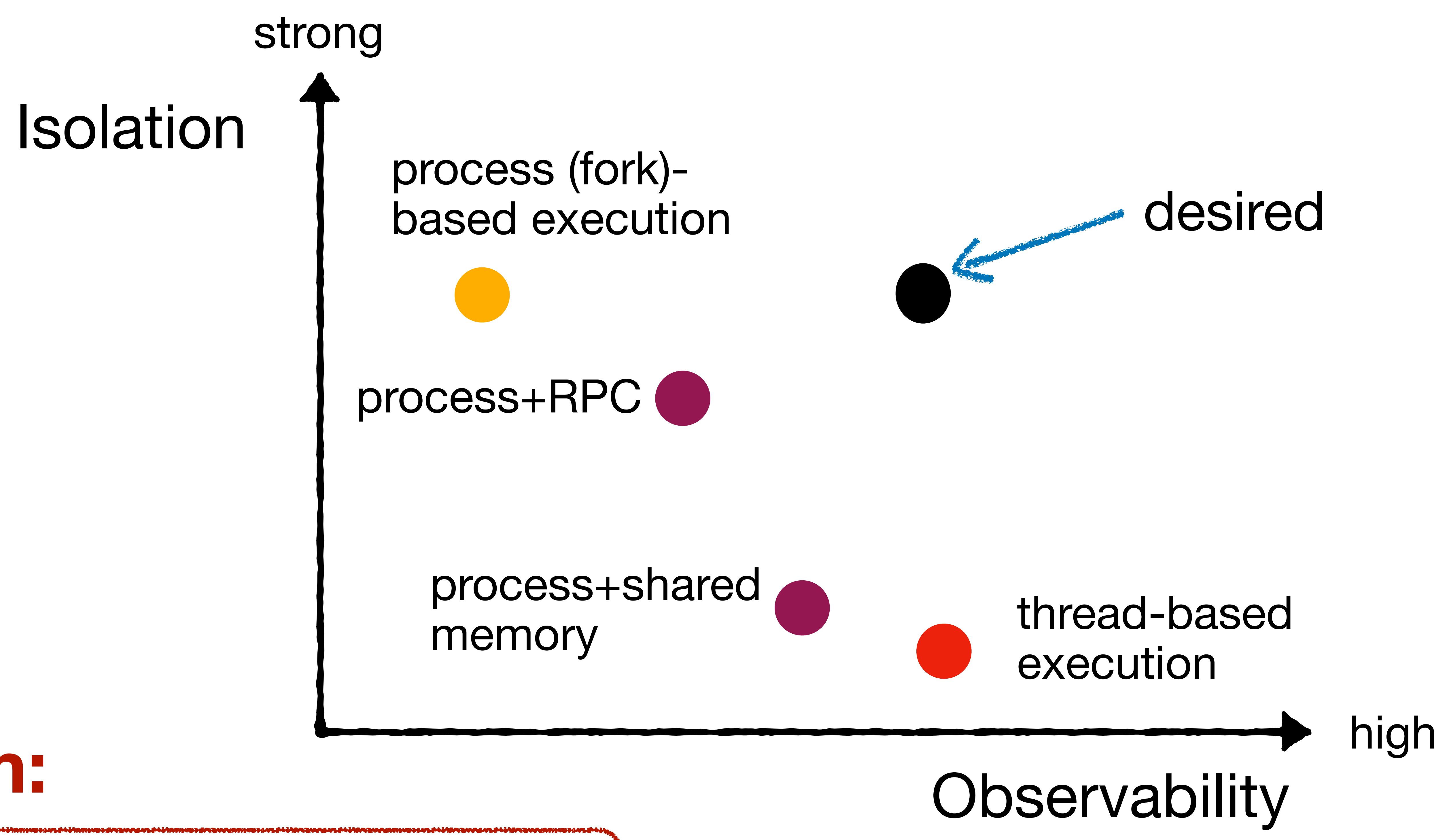
- Unsafe: a bug in auxiliary task can bring down the entire program
- A heavy task can cause severe performance interference

Current practice of auxiliary execution

Practice 2: running in another process using `fork`



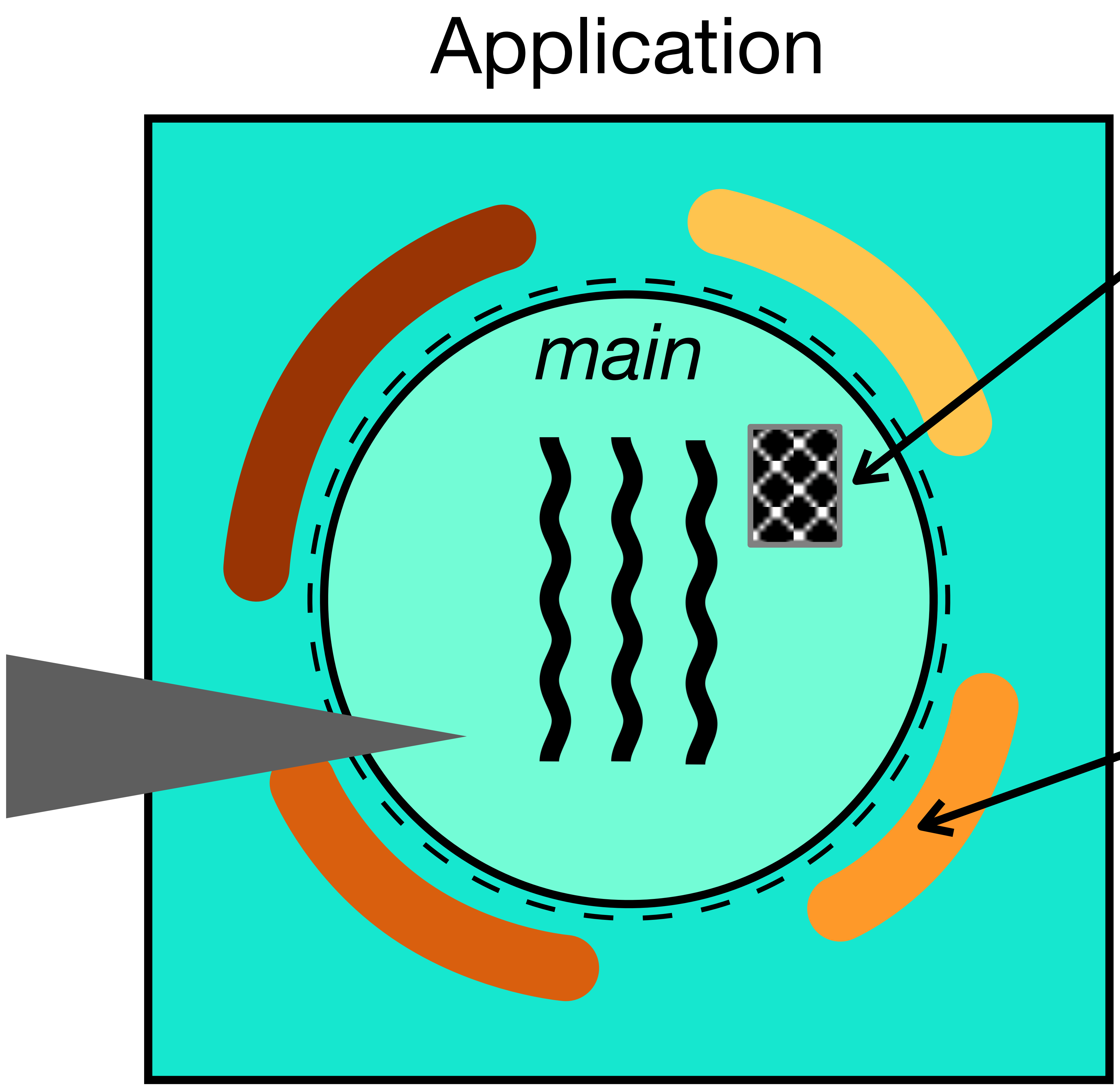
Ideal auxiliary execution



Essential problem:

Current OS abstractions force developers to choose one property over another

A missing sub-process isolation scenario



1. Extensibility

SFI (SOSP '93)

2. Secure partition

Wedge (HotOS '13), *lwC* (OSDI '16)

3. Maintenance

(most auxiliary tasks)

under-explored

Our focus!

Our Solution: Orbit

- An OS abstraction for auxiliary tasks
- Properties:

Strong isolation	buggy orbit task will *not* affect main program
Observability	easily observe main program states
Safe alteration	alter main program states safely
Efficiency	low overhead even under high frequency
First-class entity	schedulable like process & threads

Key Challenges

1. Isolation and observability are “contradictory”
   - Something isolated typically cannot see updated information

2. Isolation comes at a cost
   - Possible technique like shared memory is efficient *but against isolation*

Insights

1. Separate address spaces are essential but we can continuously mirror them

2. State observed in each invocation is typically only a small portion of all state

Overview of using orbit

1. Directly in the same application codebase
2. Easily refer to any existing variables and functions

Create	`orbit *orbit_create(orbit_entry entry, ...);`
Invoke	`long orbit_call(orbit *ob, ...);` `orbit_future *orbit_call_async(orbit *ob, ...);`
Alter	`long pull_orbit(orbit_future *f, ...);` `long orbit_push(orbit_update *update, ...);`

Orbit creation

```
int mysqld_main() {

}

lock_t* RecLock::lock_alloc(trx_t* trx) {
  lock_t* lock;
  lock = (lock_t*) mem_heap_alloc(heap, sizeof(*lock));
  return lock;
}

dberr_t lock_rec_lock() {
  if (status == LOCK_REC_FAIL) {
    check_and_resolve(lock, m_trx);
  }
}
```

Example: MySQL deadlock detector code

Orbit creation

```
+ struct orbit *ob;
  int mysqld_main() {
+   ob = orbit_create("dl_checker", check_and_resolve, NULL);
  }

  lock_t* RecLock::lock_alloc(trx_t* trx) {
    lock_t* lock;
    lock = (lock_t*) mem_heap_alloc(heap, sizeof(*lock));
    return lock;
  }

  dberr_t lock_rec_lock() {
    if (status == LOCK_REC_FAIL) {
      check_and_resolve(lock, m_trx);
    }
  }
```

orbit handle

Example: MySQL deadlock detector code

Orbit creation

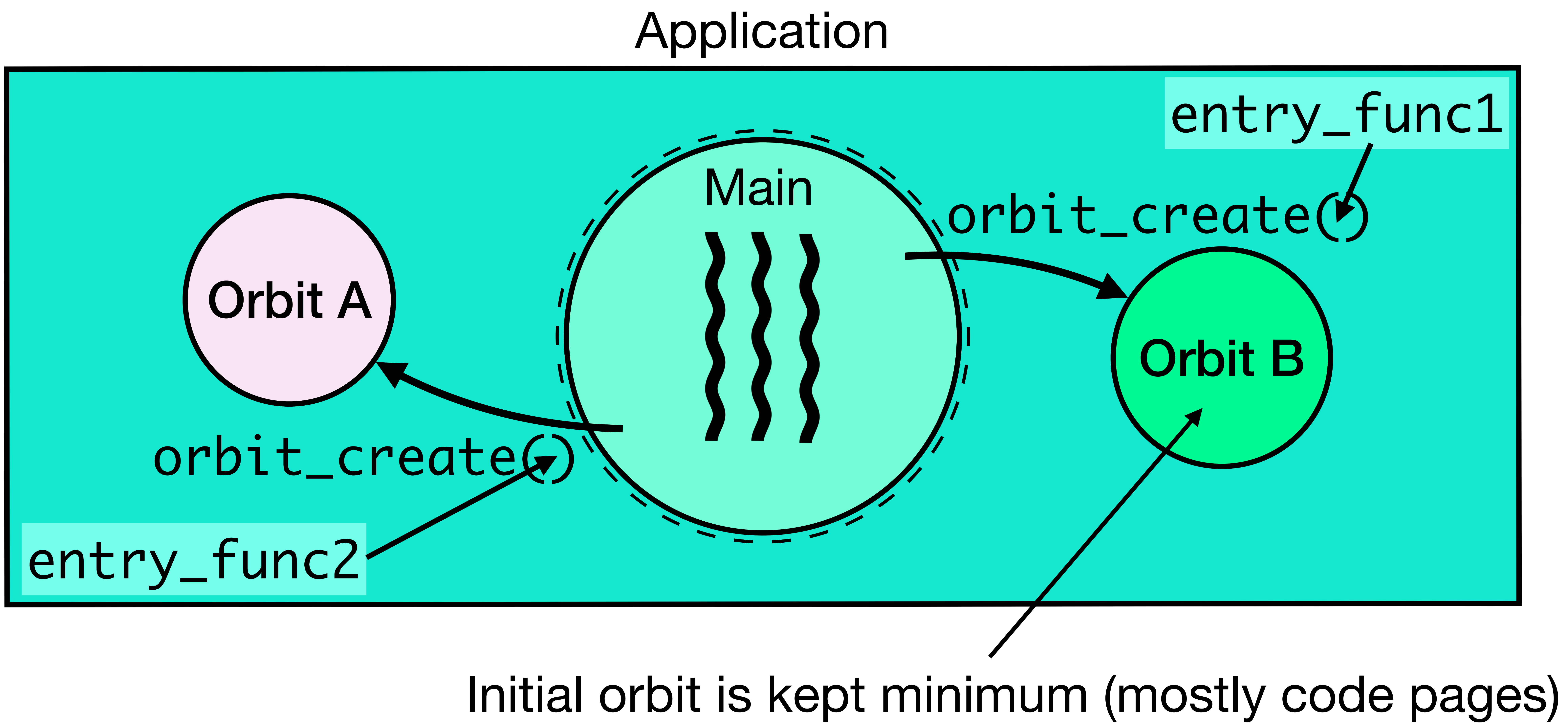
API:
```
orbit *orbit_create(const char *name, orbit_entry entry,
                    void* (*init)(void));
```

A function in app code representing the entry of an auxiliary task

Similar to `pthread_create()` but key differences:

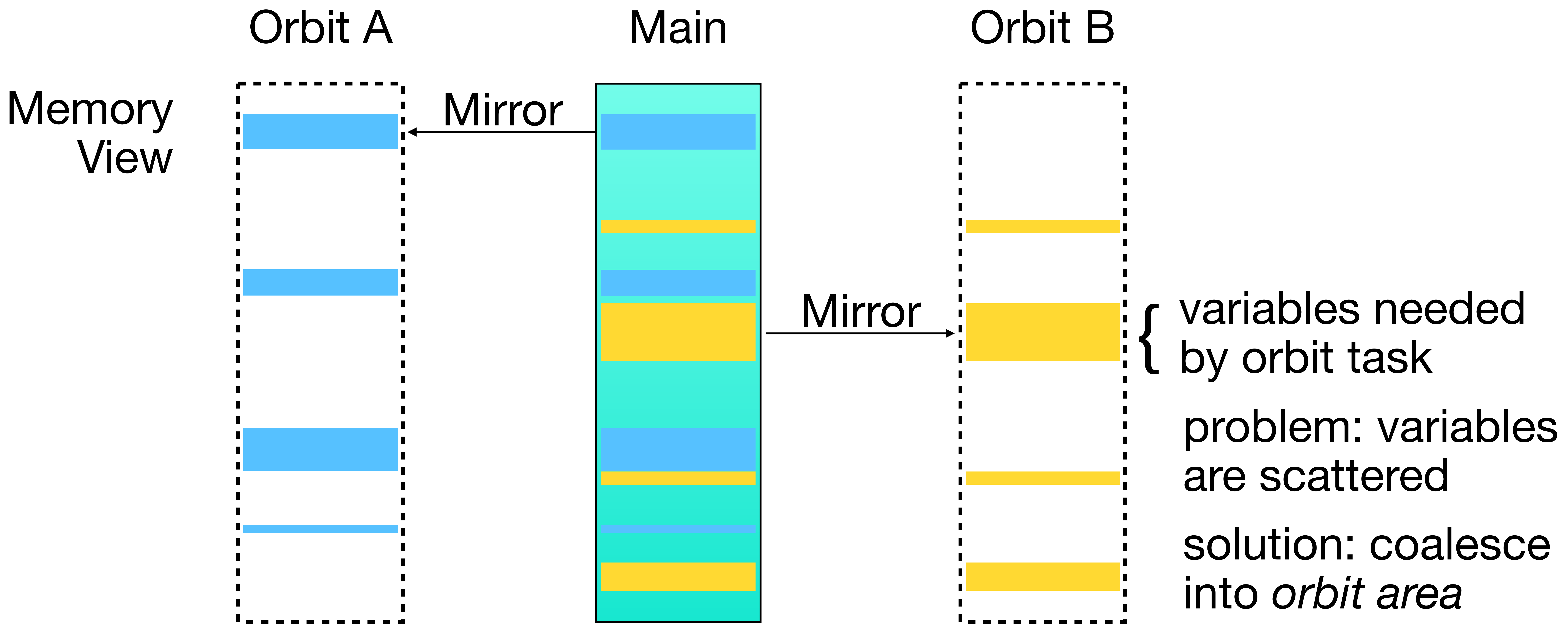
- Executes in a different address space
- Created once but not immediately executed
- Invoked multiple times later

Orbit creation



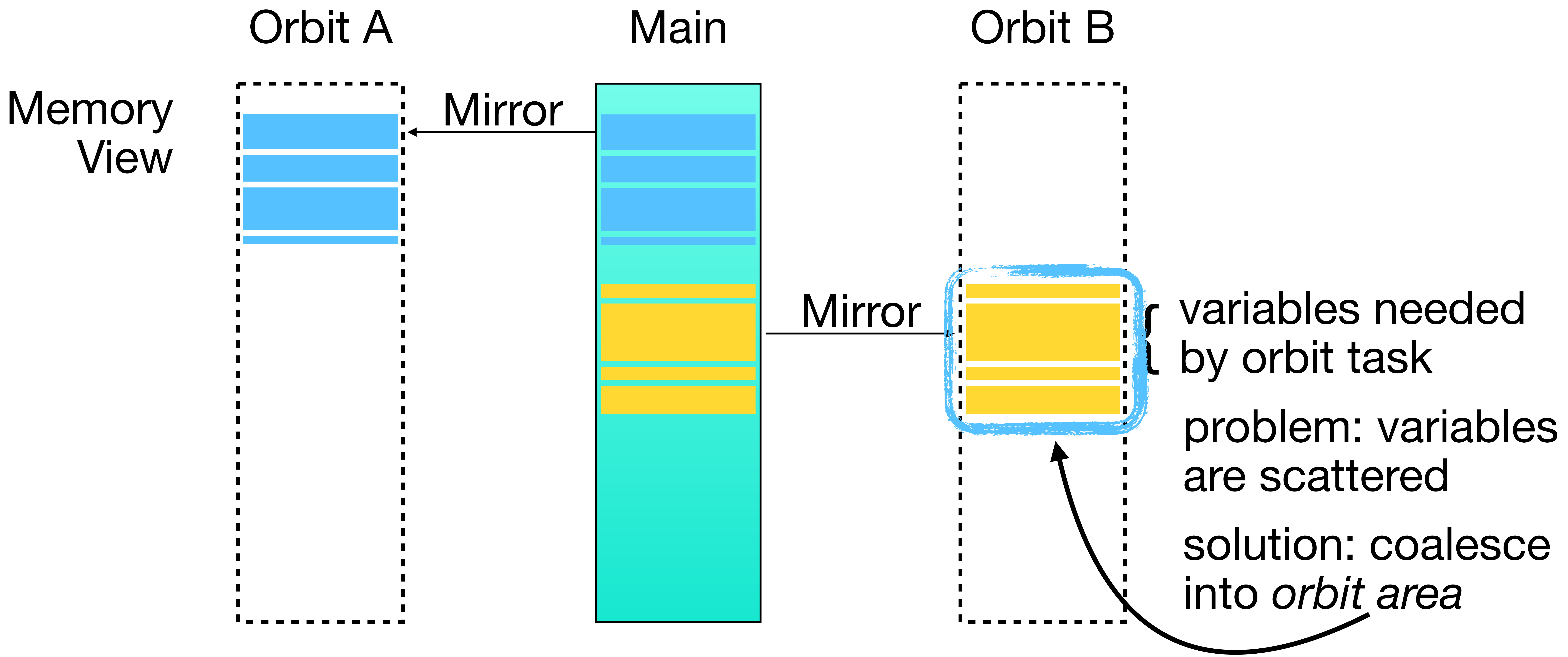
Automatic state synchronization

Orbit's memory is mirror of main program's fragments (at the same virtual address)



Automatic state synchronization

Orbit's memory is mirror of main program's fragments (at the same virtual address)



Orbit area

```
  struct orbit *ob;
+ struct orbit_area *area;
  int mysqld_main() {
    ob = orbit_create("dl_checker", check_and_resolve, NULL);
+   area = orbit_area_create(4096);
  }
  lock_t* RecLock::lock_alloc(trx_t* trx) {
    lock_t* lock;
-   lock = (lock_t*) mem_heap_alloc(heap, sizeof(*lock));
+   lock = (lock_t*) orbit_alloc(area, sizeof(*lock));
    return lock;
  }
```

Example: MySQL deadlock detector code

Compiler support

- Analyze the allocation points used by the orbit task
  - Output hints of allocation points
  - Static analysis using def-use chain

```
struct trx_t {
    int *a;
};
void modify(struct trx_t *t) {
    t->a = (int*)malloc(sizeof(int));
    *t->a = 10;
}
void check(struct trx_t *t) {
    printf("%d\n", *t->a);
}
int main() {
    struct trx_t t;
    modify(t);
    check(t);
}
```

Check the paper for details!

Orbit invocation

Make a **snapshot** of specified states **right before** the orbit call,

then execute the entry function in orbit side using snapshotted state

```
long orbit_call(orbit *ob, orbit_area** areas, ...);
```

Sync: **waits** until the entry function has executed and returned

```
orbit_future *orbit_call_async(...);
```

Async: returns after creating a snapshot with a handle **to be waited** on

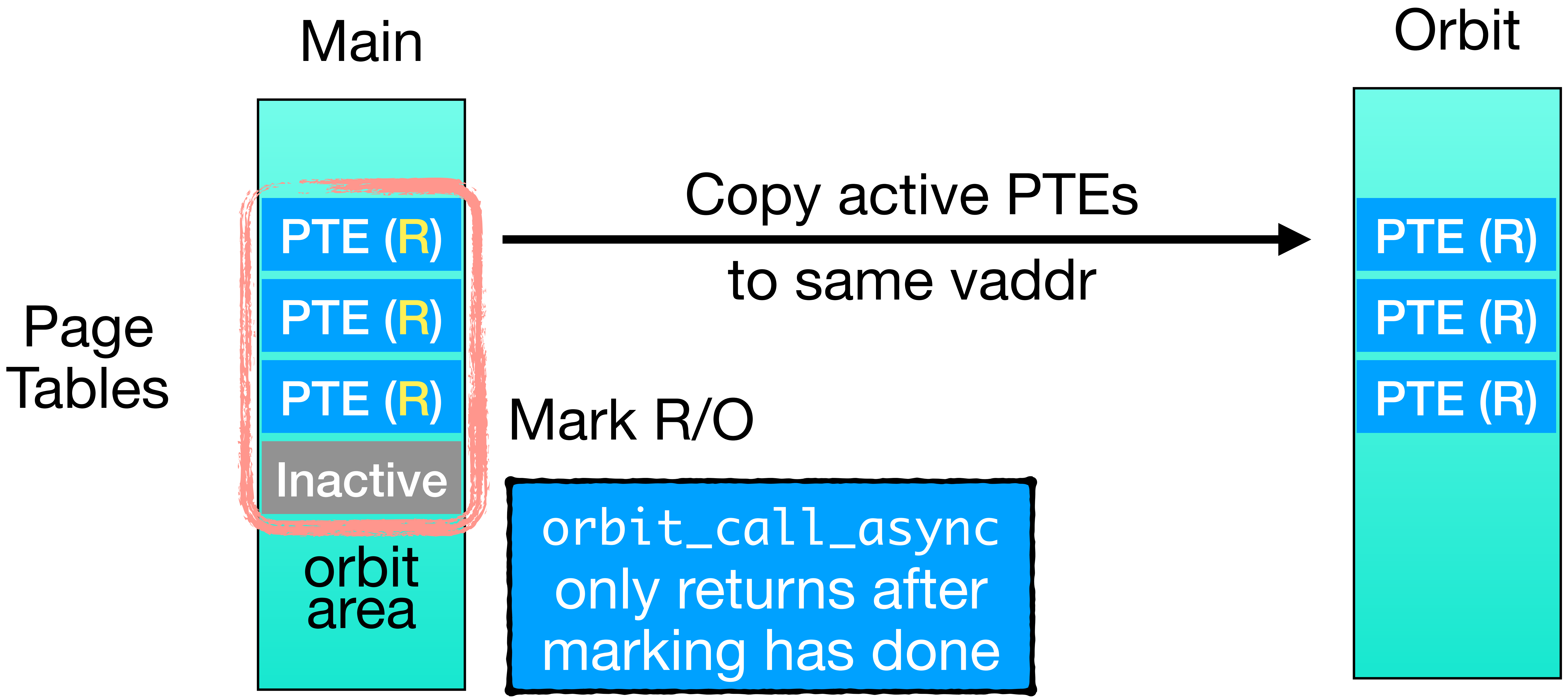
State snapshotting

- Possible approaches:
  - Data copying: slow, waste memory
  - Shadow memory: weak isolation, instrumentation, high overhead

- We choose to leverage copy-on-write
  - **Efficiency:** only copy PTEs + optimization techniques
  - **Consistency & concurrency:** ensured by several designs

State snapshotting

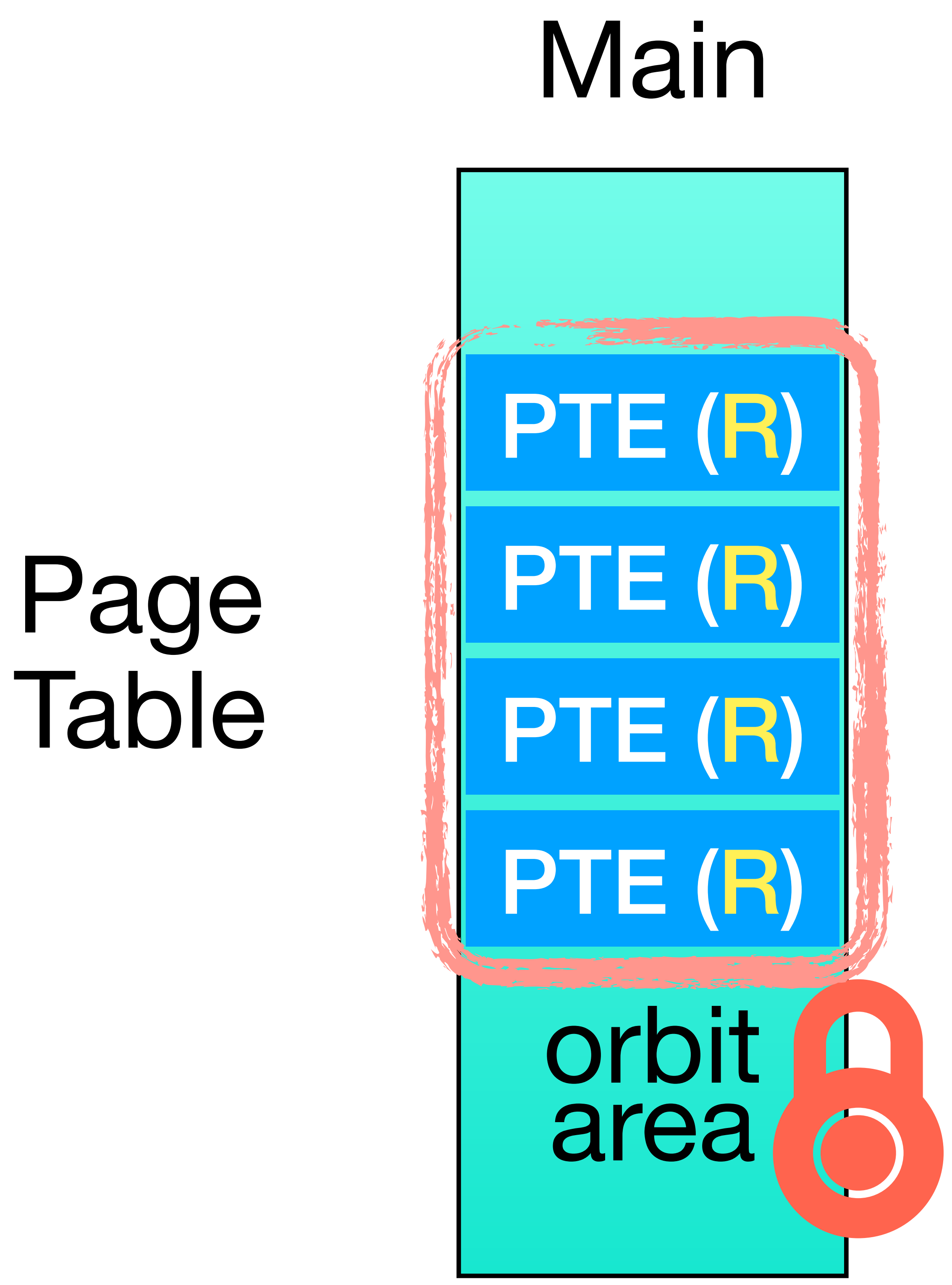
Classic COW



W: writable R: read-only

State snapshotting

Scenario 1: multi-threaded application



Possible solution: pause all threads when snapshotting

- Significant performance penalty

Observation: the original call sites are usually already synchronized

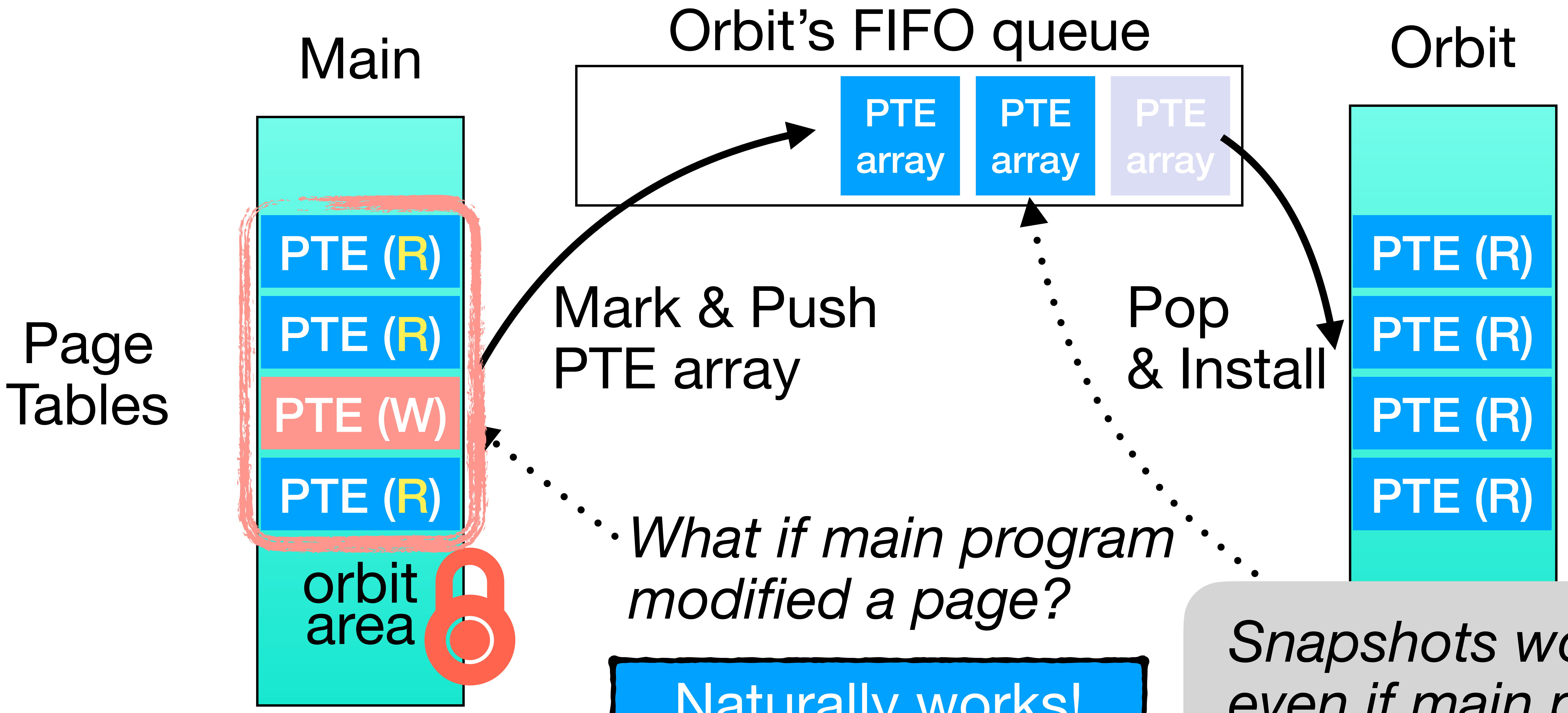
Rely on app-level synchronization

W: writable R: read-only

State snapshotting

Scenario 2: concurrent orbit calls

serialize orbit calls



Naturally works!

Snapshots won't change even if main program page has changed

W: writable R: read-only

Optimization

Techniques:

- Incremental snapshotting
- Delegate objects
- Dynamic page mode selection

Optimization: delegate object

Problem: large struct with only few fields accessed wastes orbit area memory

Solution: separate allocation of large struct and used fields

```
// allocate 912 bytes with malloc
struct trx_t {
  struct trx_lock_t {
    ...
-   lock_t* wait_lock;
+   lock_t*& wait_lock;
    ...
  } lock;
};
```

```
// allocate 104 bytes with orbit_alloc
struct trx_t_delegate {
  struct {
    lock_t* wait_lock;
  } lock;
};
```

Define a delegate struct that only keeps the fields needed

Optimization: delegate object

Problem: large struct with only few fields accessed wastes orbit area memory

Solution: separate allocation of large struct and used fields

```
// allocate 912 bytes with malloc
struct trx_t {
  struct trx_lock_t {
    ...
-   lock_t* wait_lock;
+   lock_t*& wait_lock;
    ...
  } lock;
};
```

C++ reference binding

no code changes needed at usage point

```
// allocate 104 bytes with orbit_alloc
struct trx_t_delegate {
  struct {
   lock_t* wait_lock;
  } lock;
};
```

Define a delegate struct that only keeps the fields needed

Altering main program states

- Transparently replace modified pages?
  - **Problem:** state merge conflicts

- Controlled alteration with `orbit_update`
  - **Precise modification:** byte-wise field copying
  - **Avoid partial updates:** batched updates

Controlled state alteration

Packing and logging modifications

```
// within orbit task
void trx_rollback(trx_t *victim) {

  orbit_update *scratch =
        orbit_update_create();









}
```

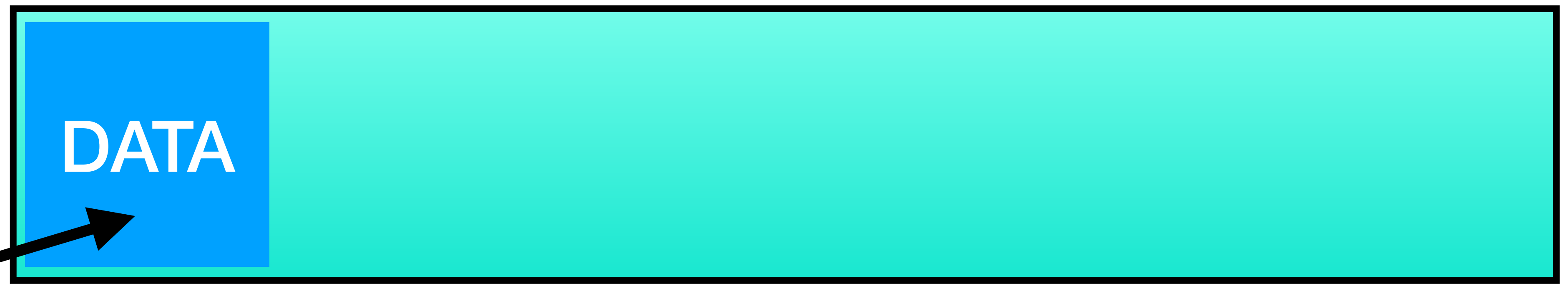
Create an empty update as a *scratch*

Controlled state alteration

Packing and logging modifications

```
// within orbit task
void trx_rollback(trx_t *victim) {
  orbit_update *scratch =
        orbit_update_create();
  orbit_update_add_data(scratch,
        &victim->version);
}
```

Scratch



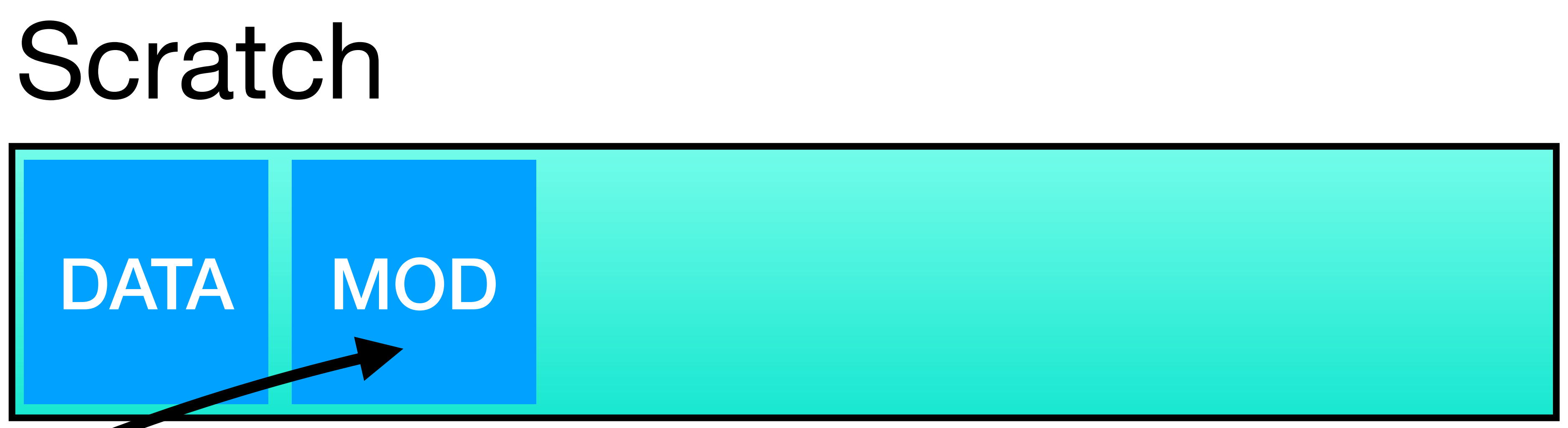
Flexibility: allow adding **arbitrary data**

- Can be made for any use
- Later in this example, `version` is used for stale check

Controlled state alteration

Packing and logging modifications

```
// within orbit task
void trx_rollback(trx_t *victim) {

  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);

}
```



Precise field update

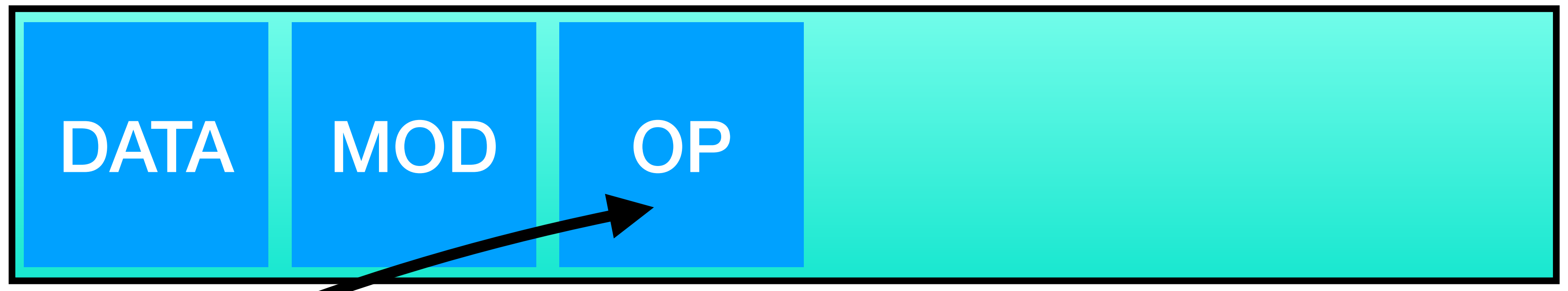
- Record memory address, value

Controlled state alteration

Packing and logging modifications

```
// within orbit task
void trx_rollback(trx_t *victim) {

  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);


}
```

Scratch



Flexibility: **run operation**

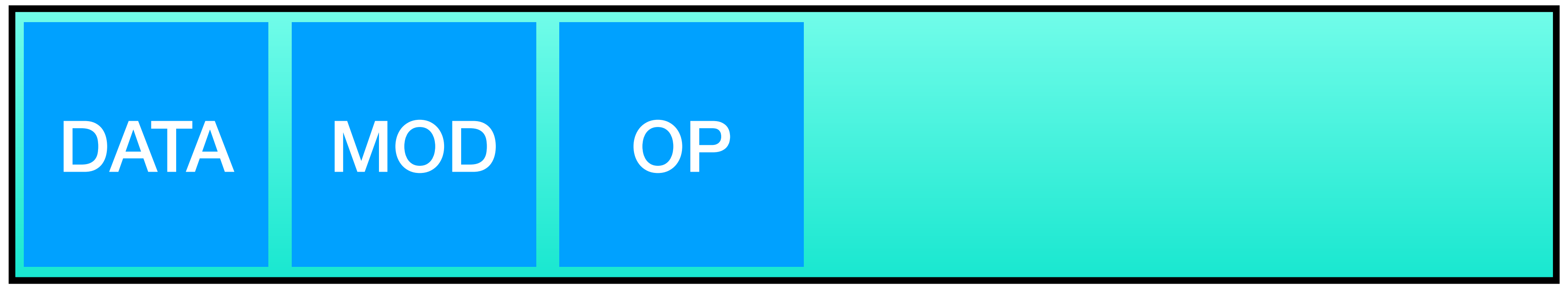
- Modification such as signaling condvar cannot be done in orbit
- Record *function* and *argument*, run in main program

Controlled state alteration

Packing and logging modifications

```
// within orbit task
void trx_rollback(trx_t *victim) {
  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);
}
```

Scratch



`orbit_push`: push back updates in a **batch**

- Prevents partial state alteration: crashed orbit will not push partial updates

Pushing back updates

Applying updates

```
// in main program
void handle_rollback(orbit_future *future) {

  orbit_update update;
  long ret = pull_orbit(future, &update);

  TrxVersion *version = orbit_update_first(update)->data;
  if (trx_is_alive(version))
    orbit_apply(update);

}
```

Main program can choose whether
to apply or to discard the updates

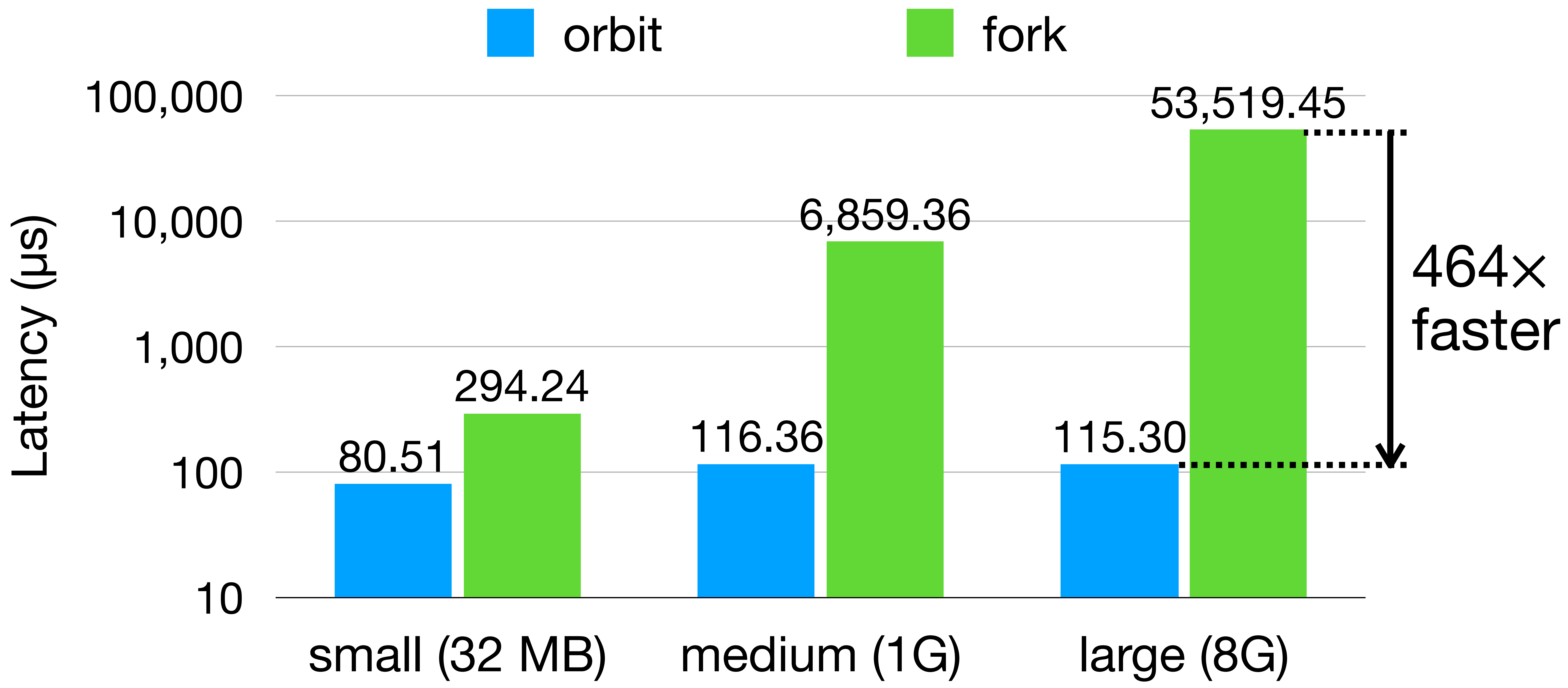
Evaluation

Setup

- Implemented orbit in Linux kernel 5.4.91
- Ported 7 tasks from 6 systems
- Implemented 1 new task
- Environment:
  - KVM-enabled QEMU VM w/ 4vCPU & 10GB memory
  - Debian 10 with custom kernel

Microbenchmark: creation

Test latency of `orbit_create` compared with `fork`



Real-world applications

App	Task	Source	Category
MySQL	#1: deadlock detector	ported	Error detector
Apache	#2: lock watchdog	new	
	#3: proxy balancer	ported	Resource manager
Varnish	#4: pool herder	ported	
Nginx	#5: WebDAV PUT handler	ported	Functionality
Redis	#6: Slow log	ported	Debugging
	#7: RDB persistence	ported	Checkpointing
LevelDB	#8: background compaction	ported	

Isolation

Bug cases

- 8 null pointer dereference injections in all tasks
- 4 real-world bugs reproduced
- 2 resource abuse bug injections: OOM bug + CPU hogging bug
- 1 long lock wait injection in new task (Apache lock watchdog)

All impacts are isolated to the orbit task, and main program not affected (example next page).

Example: Apache proxy balancer seg fault

Bug #59864: Stack overflow due to mutual fallback configuration

- Segfault makes all clients in same worker drop connection

```
proxy_worker *find_route_worker(
    const char *route) {
  ...
  rworker = find_route_worker(
    worker->s->redirect);
  ...
}
```

Infinite recursion

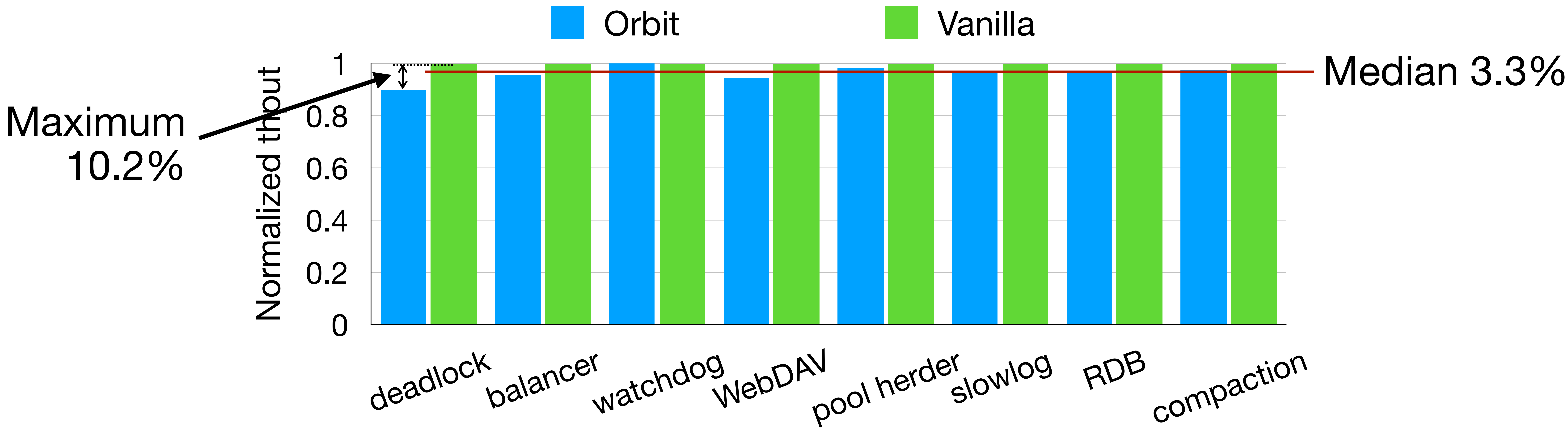
Orbit version

- All clients protected
- Graceful restart by checking `orbit_call` value
- Meaningful error message

```
int proxy_balancer_pre_request(…) {
    update = orbit_call(ob, …);
    if (is_error(update)) {
        ob = orbit_create(…);
        return HTTP_SERVICE_UNAVAILABLE;
    }
}
```

Throughput overhead

Test with YCSB, sysbench, ApacheBenchmark, YCSB, hand-written, etc.



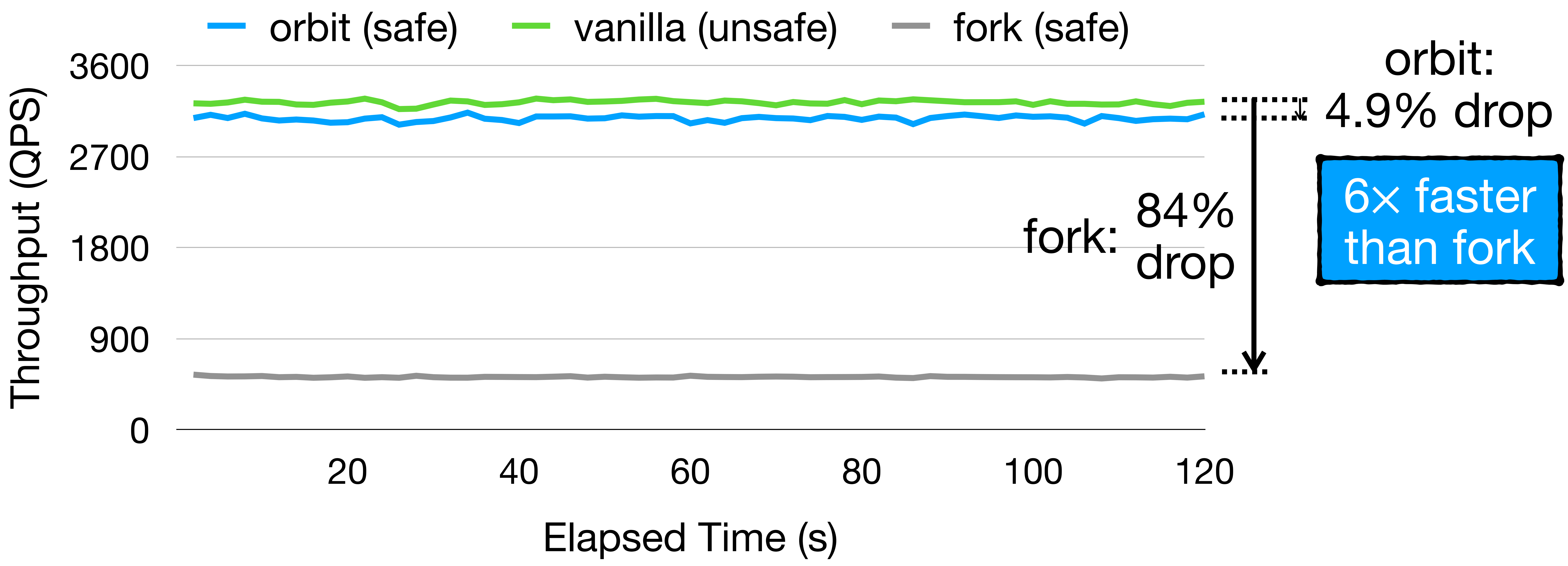
Calls/s:	510.1	1128	1	1142	1	80.7	0.2	9.9

Ensure high invocation frequency

Comparison with fork

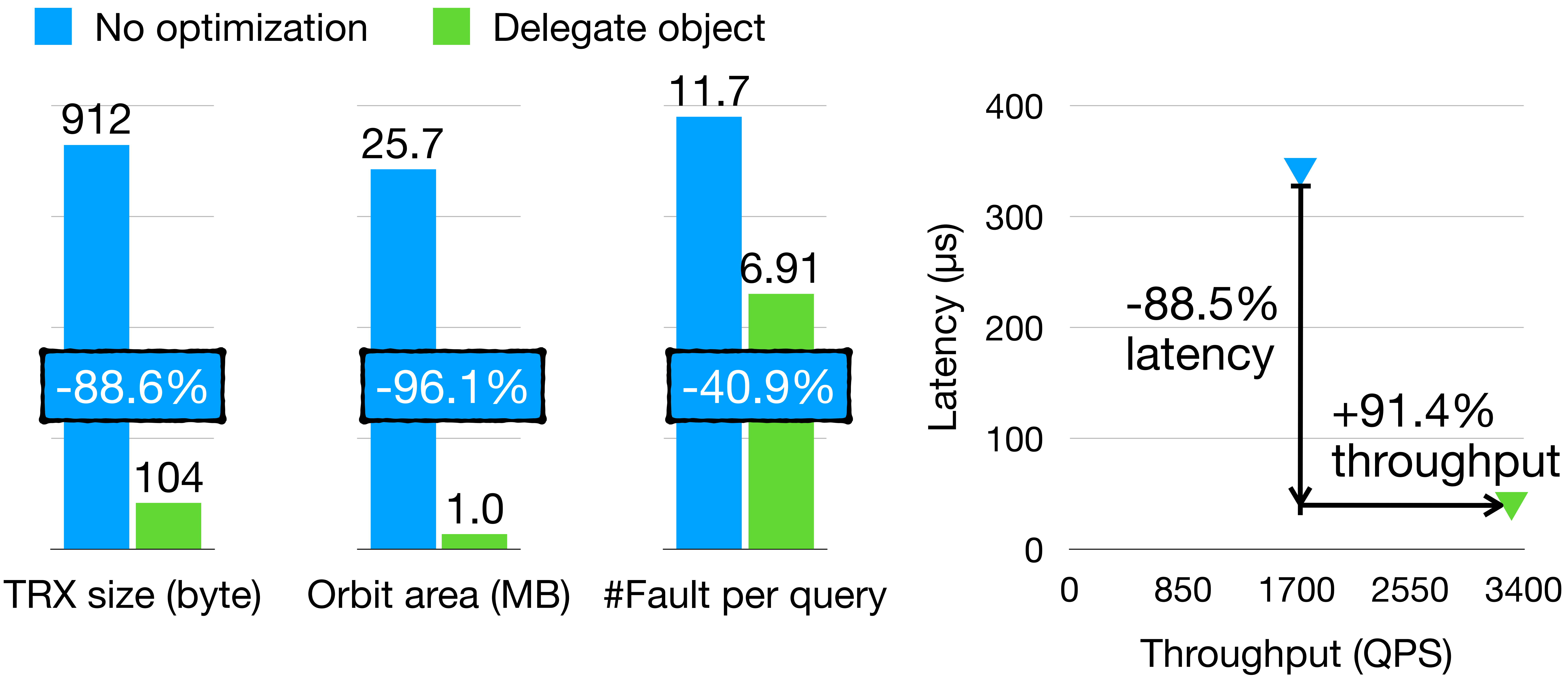
MySQL deadlock detector

Tested with a user workload in a performance bug case #49047 with 8 clients



Optimization: delegate object

MySQL deadlock detector



Conclusion

- Auxiliary tasks increasingly common
  - can cause safety and performance issues
- Current OS abstractions are not well-suited for aux tasks
- New OS abstraction *Orbit*
  - Strong isolation, high observability, efficiency
  - Evaluated on real apps & tasks

https://github.com/OrderLab/orbit

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED