Pushing Performance Isolation Boundaries into Application with pBox

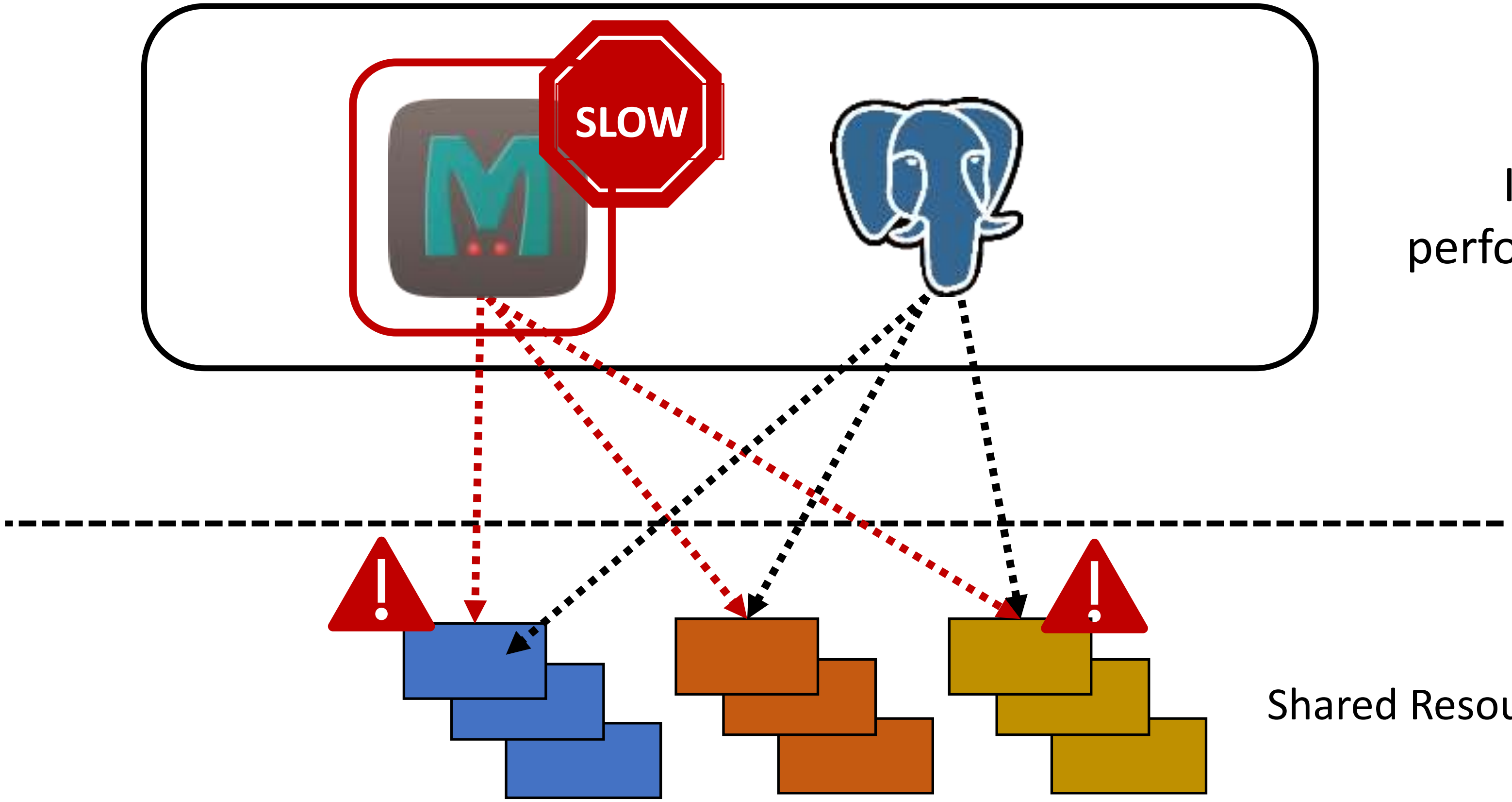
Yigong Hu, Gongqi Huang, Ryan Huang

Johns Hopkins University

University of Michigan

SOSP'23

Performance Isolation Is Critical in Application



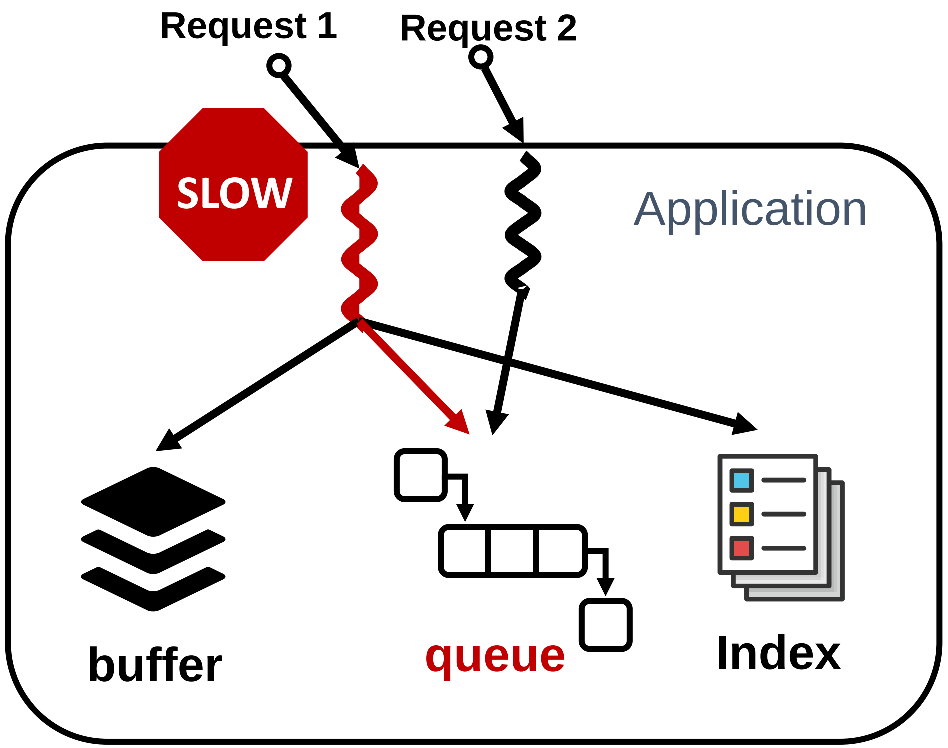
Inter-application performance interference

Shared Resource

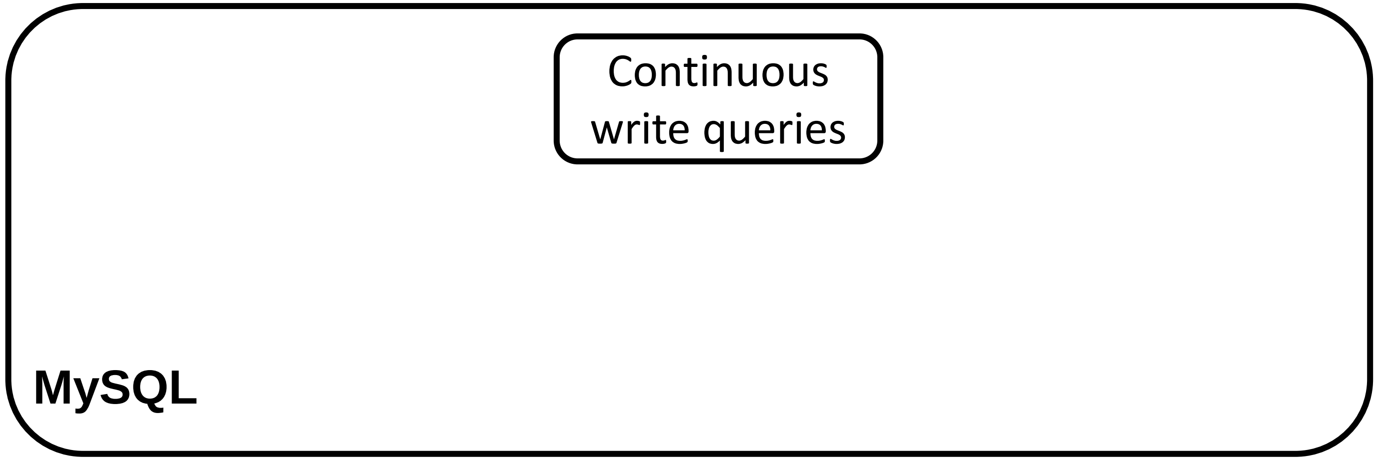
*Caladan(OSDI'20),PARTIES(ASPLOS'19)

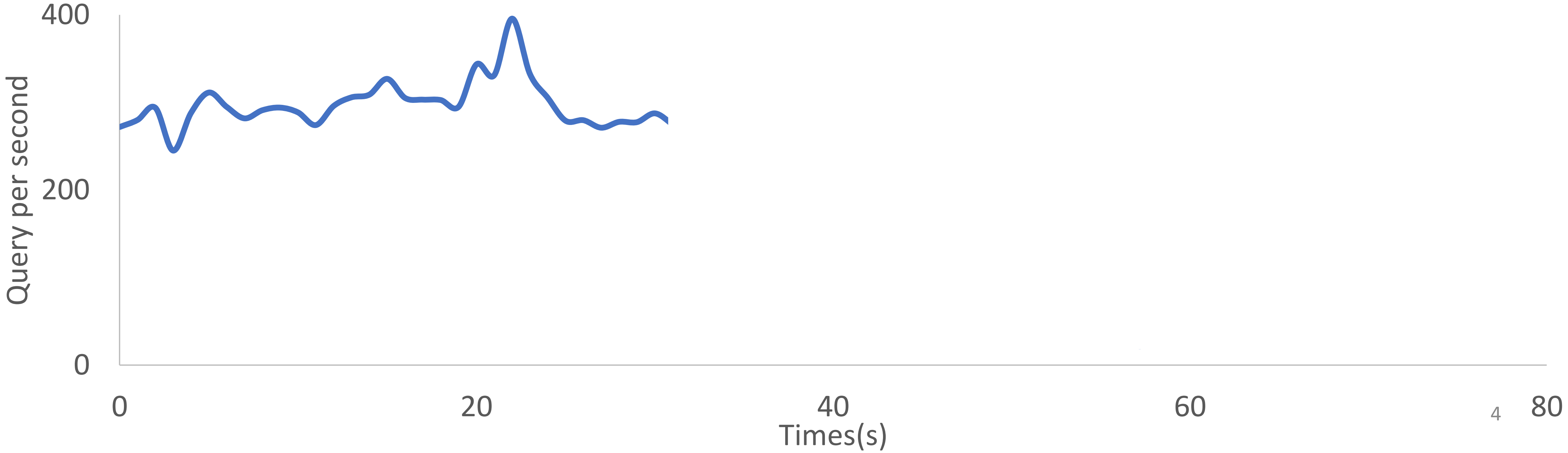
Intra-application Performance Interference

- **Performance interference can happen inside application**
  - Tasks in same application contend for shared **application *virtual resources***
  - Cause severe performance interference in production

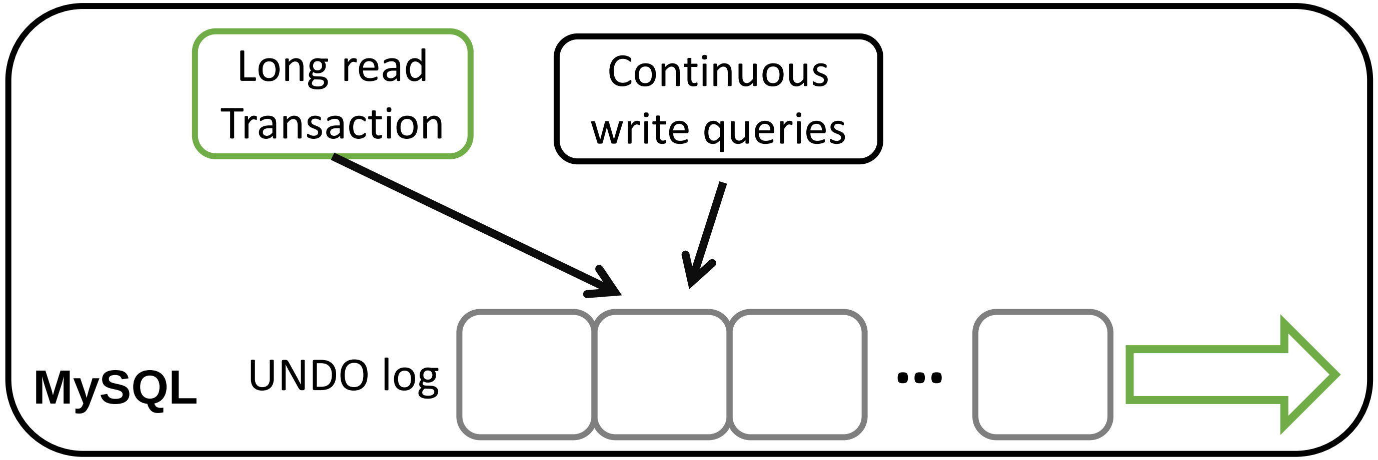


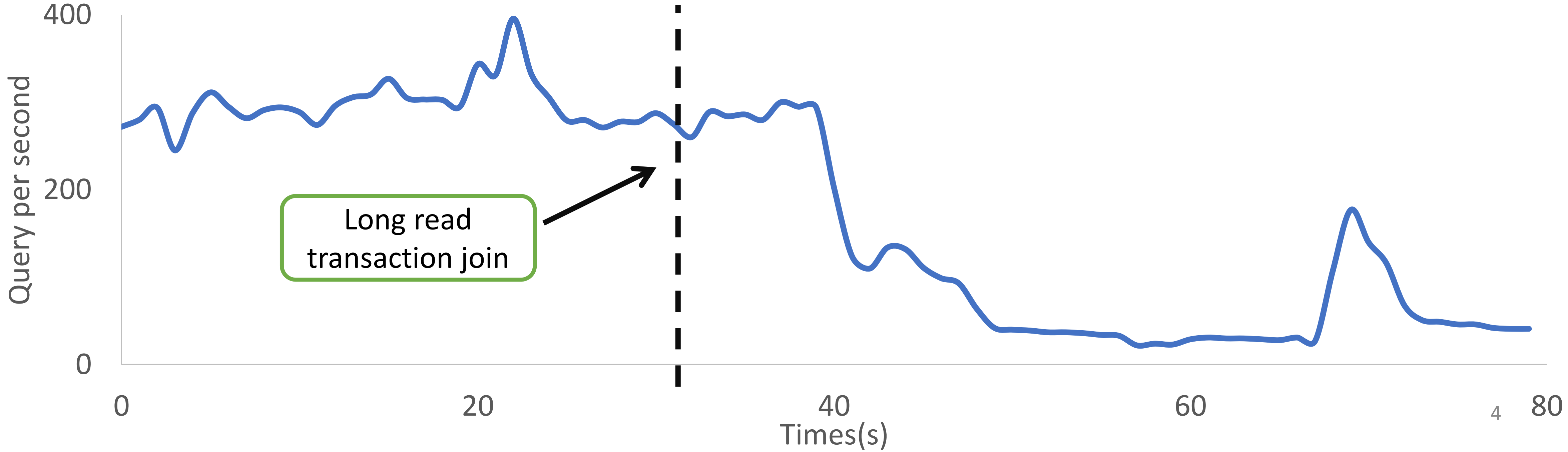
Intra-app Interference Example



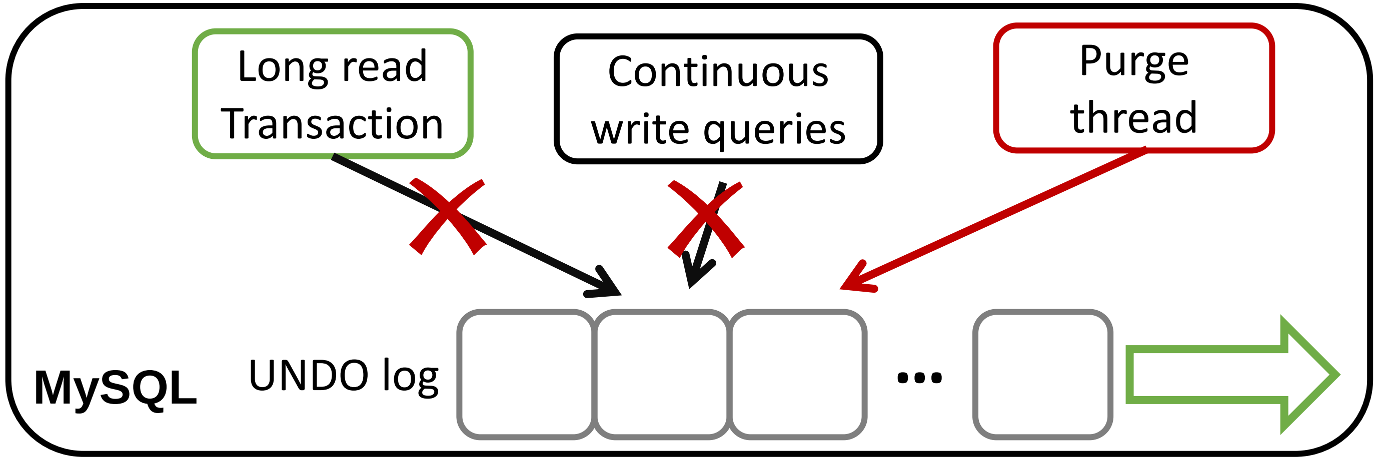


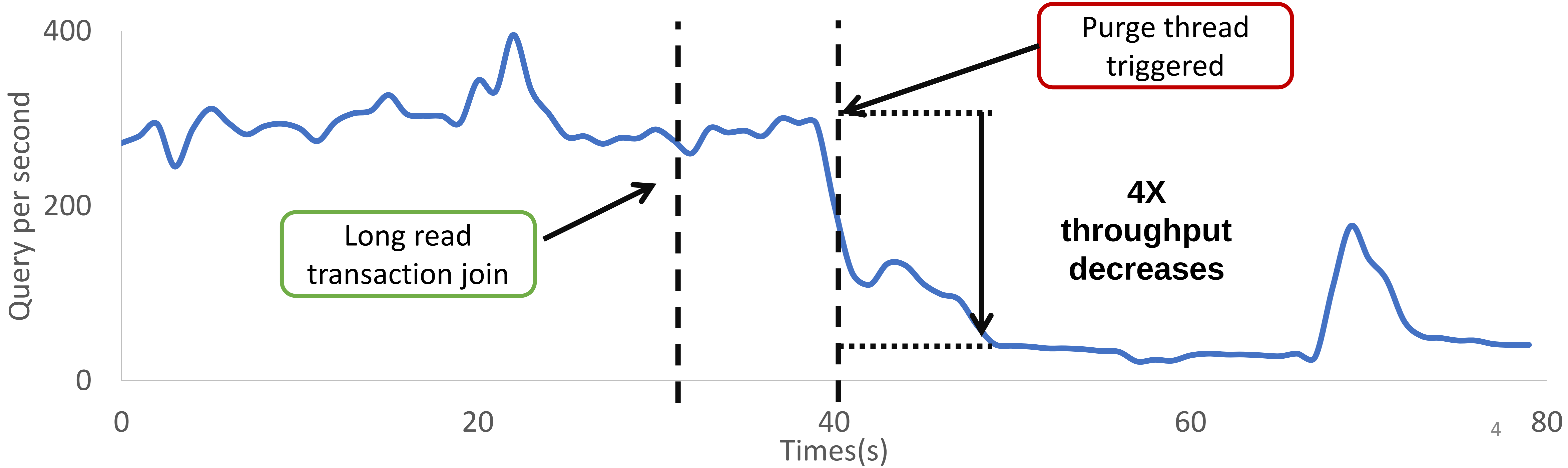
Intra-app Interference Example





Intra-app Interference Example





Intra-app Interference Is Prevalent

MySQL performance implications of InnoDB isolation modes

January 14, 2015 Peter Zaitsev

Over the past few months I've written a couple of posts about dangerous debt of InnoDB Transactional

Hi

[21 Jan 2015 14:34] MySQL Verification Team

This is a very well known bottleneck that required changes in current design. The new design has been described in several WorkLog entries for the purpose.

It will require a new UNDO log design to get fixed. Purge will need to skip the blocking transaction and delete the UNDO entries that are no longer needed. It is contained in several WL # entries, like 5742.

In progress INSERT wrecks plans on table

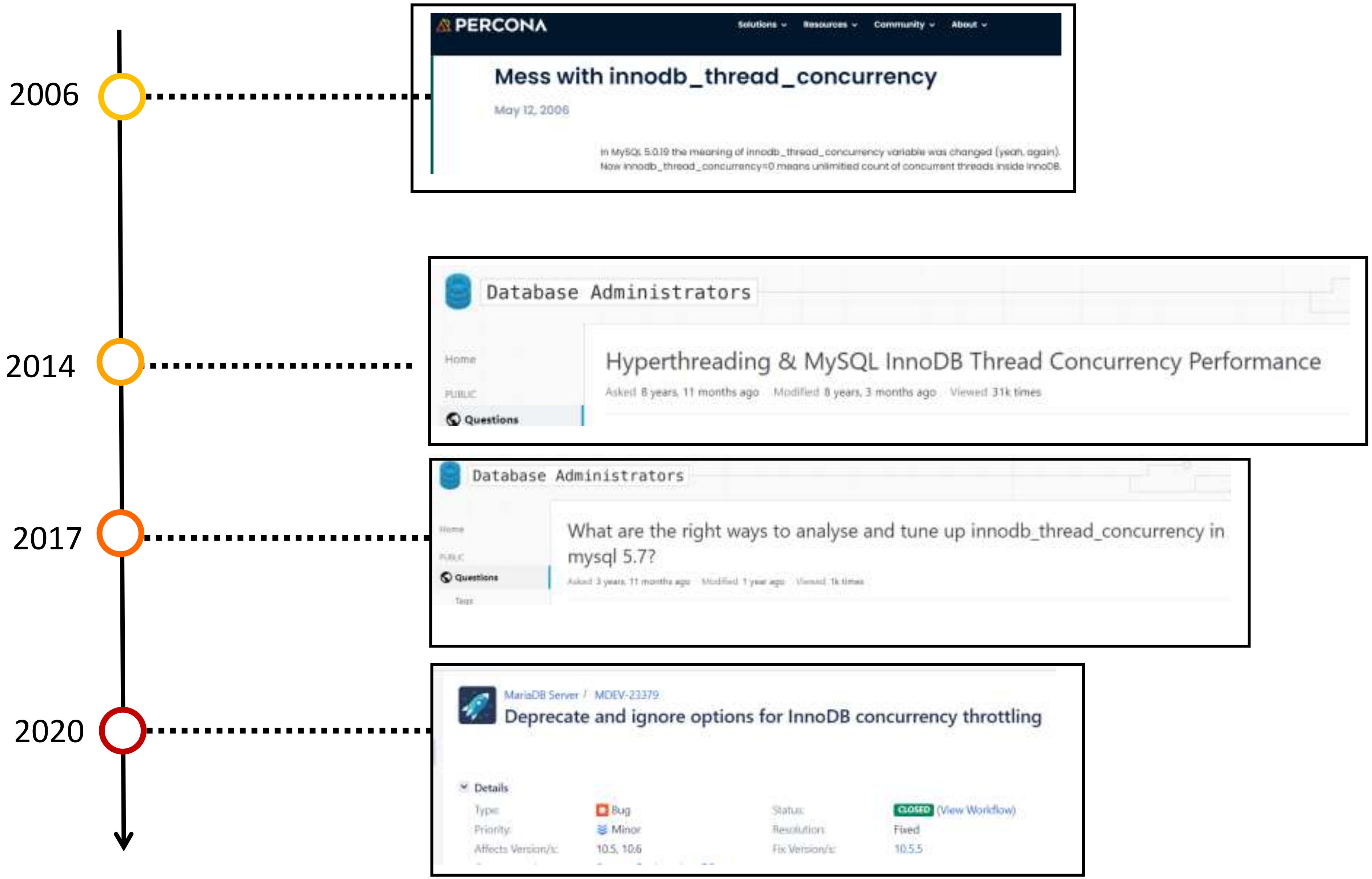
Lists: pgsql-hackerspgsql-performance

From: Mark Kirkwood <mark(dot)kirkwood(at)catalyst(dot)net(dot)nz>
To: "pgsql-performance(at)postgresql(dot)org" <pgsql-performance(at)postgresql(dot)org>
Subject: In progress INSERT wrecks plans on table
Date: 2013-04-26 02:33:31
Message-ID: 5179E
Views: Raw M
Lists: pgsql-

Recently we enco

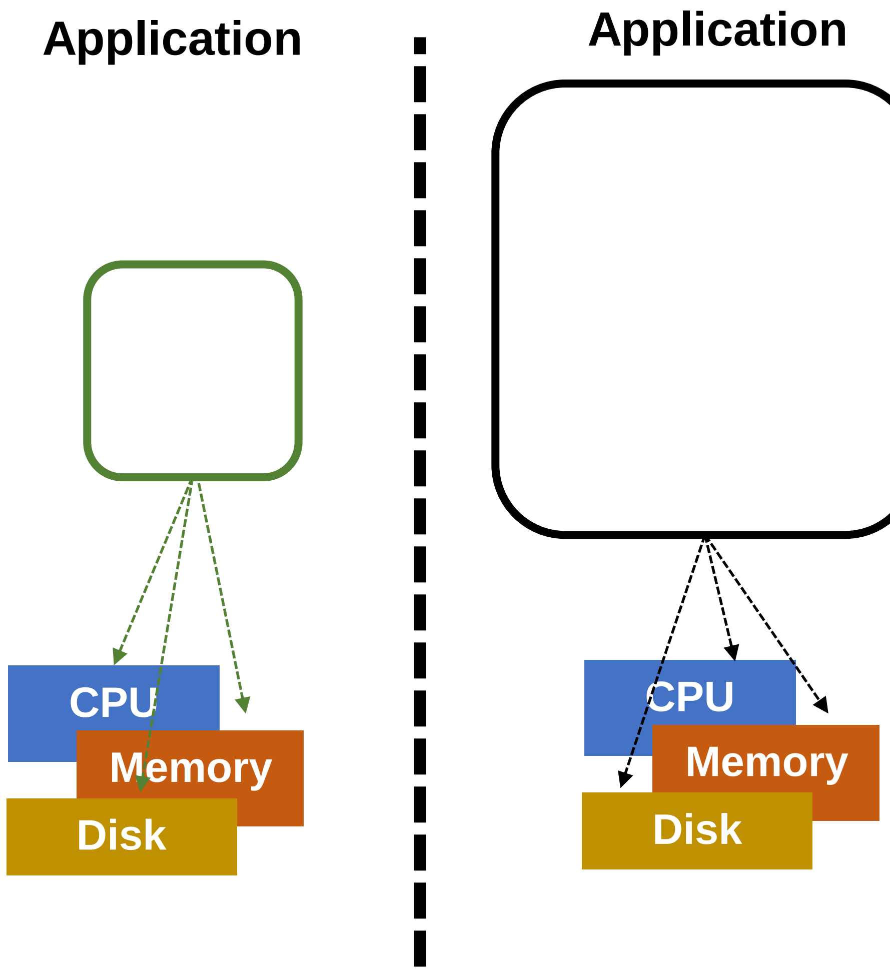
solved in various ways. I don't see much that Postgres can do because it can't know ahead of time you're about to load rows. We could imagine an optimizer that set thresholds on plans that caused the whole plan to be recalculated half way thru a run, but that would be a lot of work to design and implement and even harder to test. Having static plans at least allows us to discuss what it does after the fact with some ease.

Intra-app Interference Is Long-lived



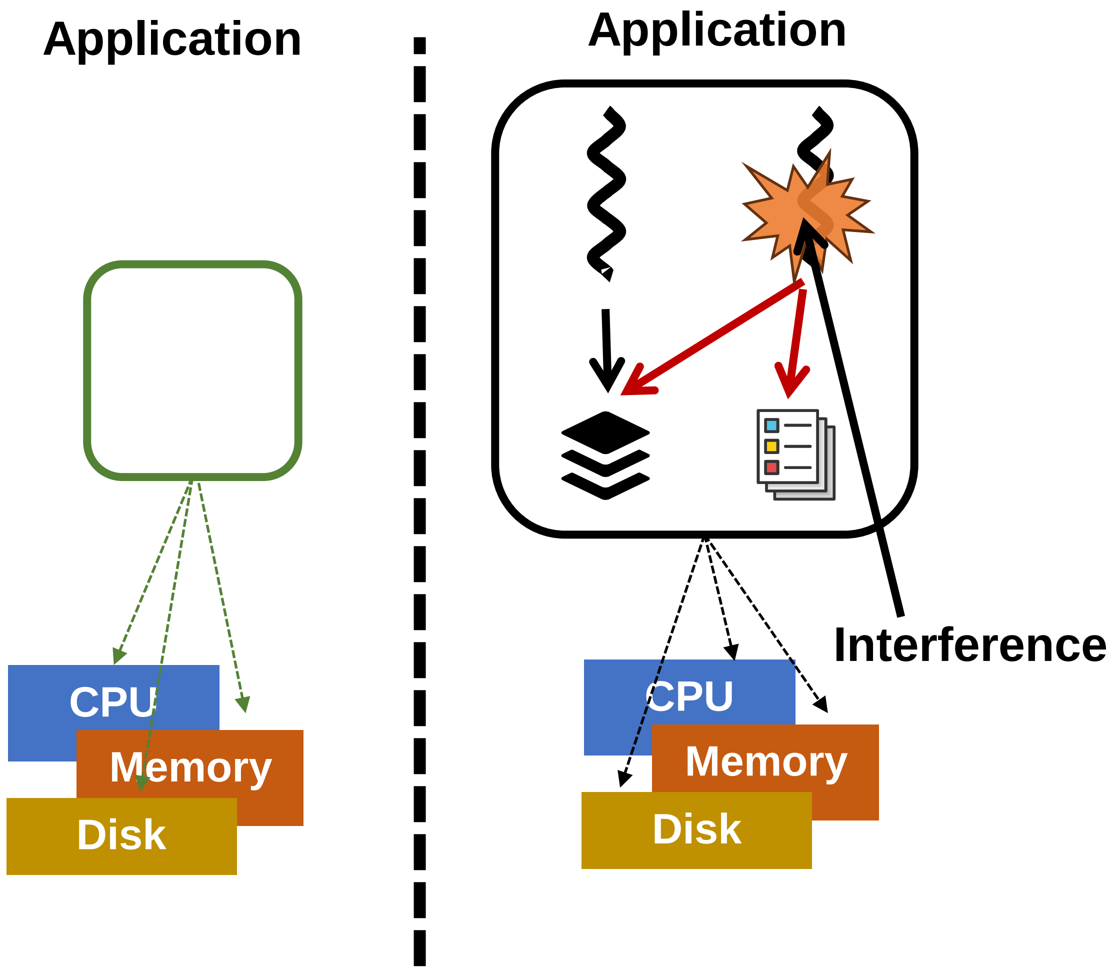
Current Practice of Performance Isolation

Practice 1: performance isolation by partitioning hardware resource



Current Practice of Performance Isolation

Practice 1: performance isolation by partitioning hardware resource

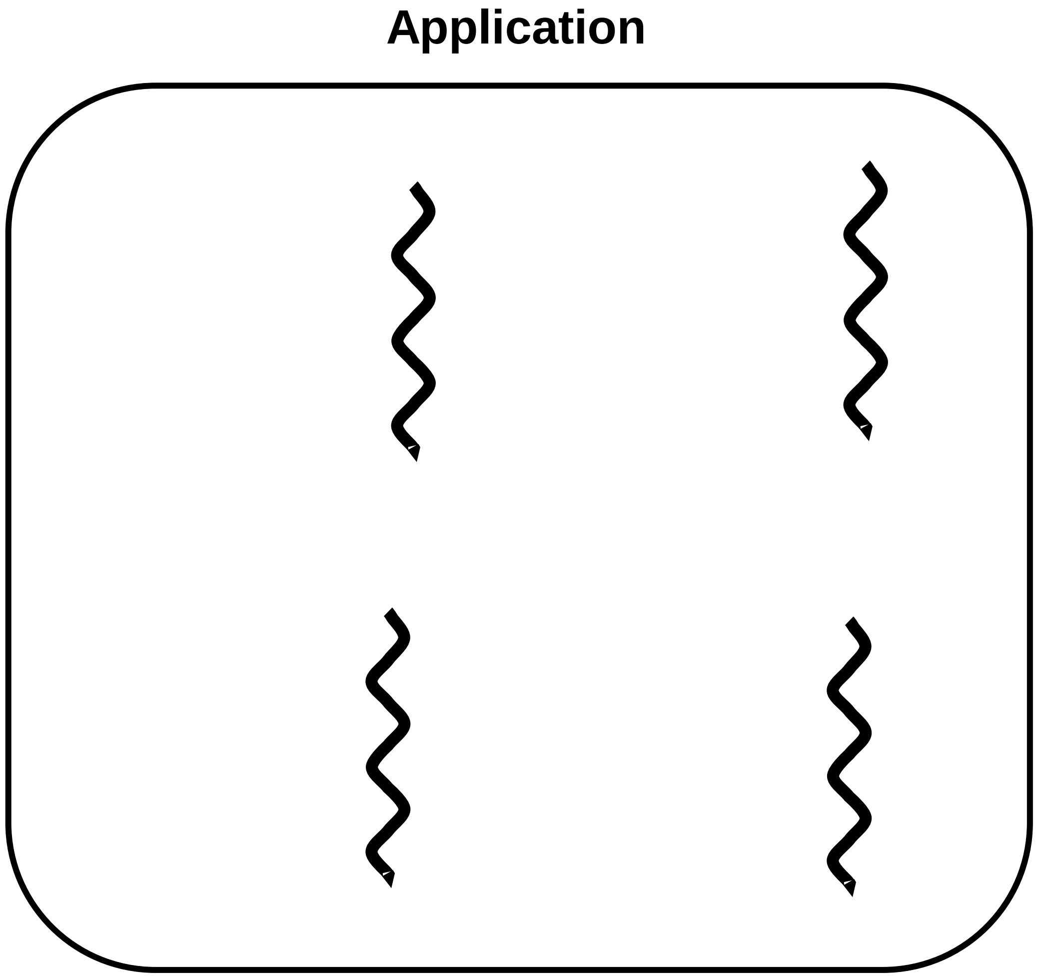


Problems:

- Application virtual resource is **invisible** to OS
- Allocating hardware resource can't directly affect application resource contention

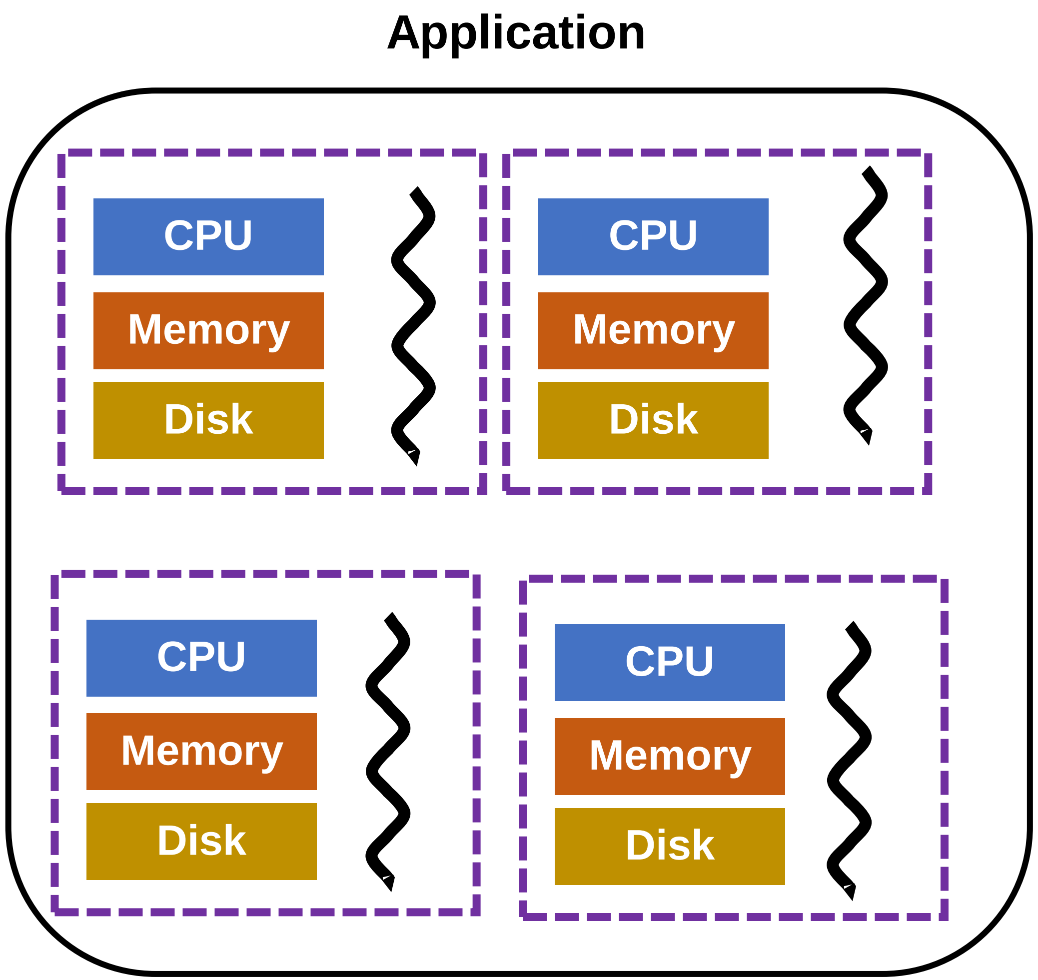
Current Practice of Performance Isolation

Practice 2: fine-grained resource quota



Current Practice of Performance Isolation

Practice 2: fine-grained resource quota



Solutions:

- Assign Fixed resource quota
- Trace application tasks' resource usage
- Deny excessive resource usage

Problems:

- Resource usage is shifting
- Hard to set quota statically

Issues of Current Practice

Difficult to enforce isolation inside applications

“Invisible to diverse application-level virtual resources

Insufficient and inflexible to enforce resource quota

Our Solution – *pBox*

Difficult to enforce isolation inside applications

→ Let developers **define** performance isolation domain in **application**

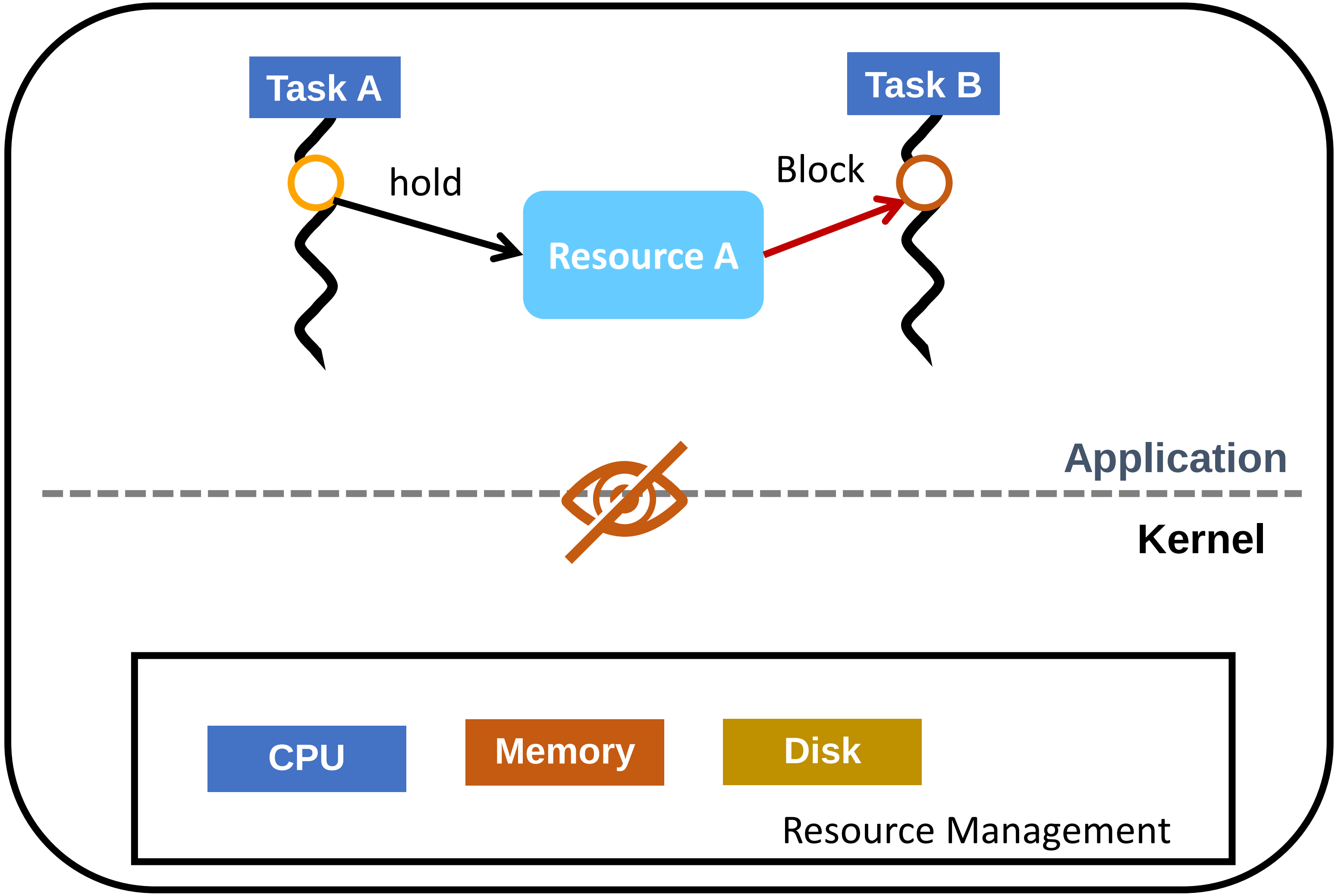
"Invisible to diverse application-level virtual resources

→ Expose set of **APIs** to easily trace application resource usage

Insufficient and inflexible to enforce resource quota

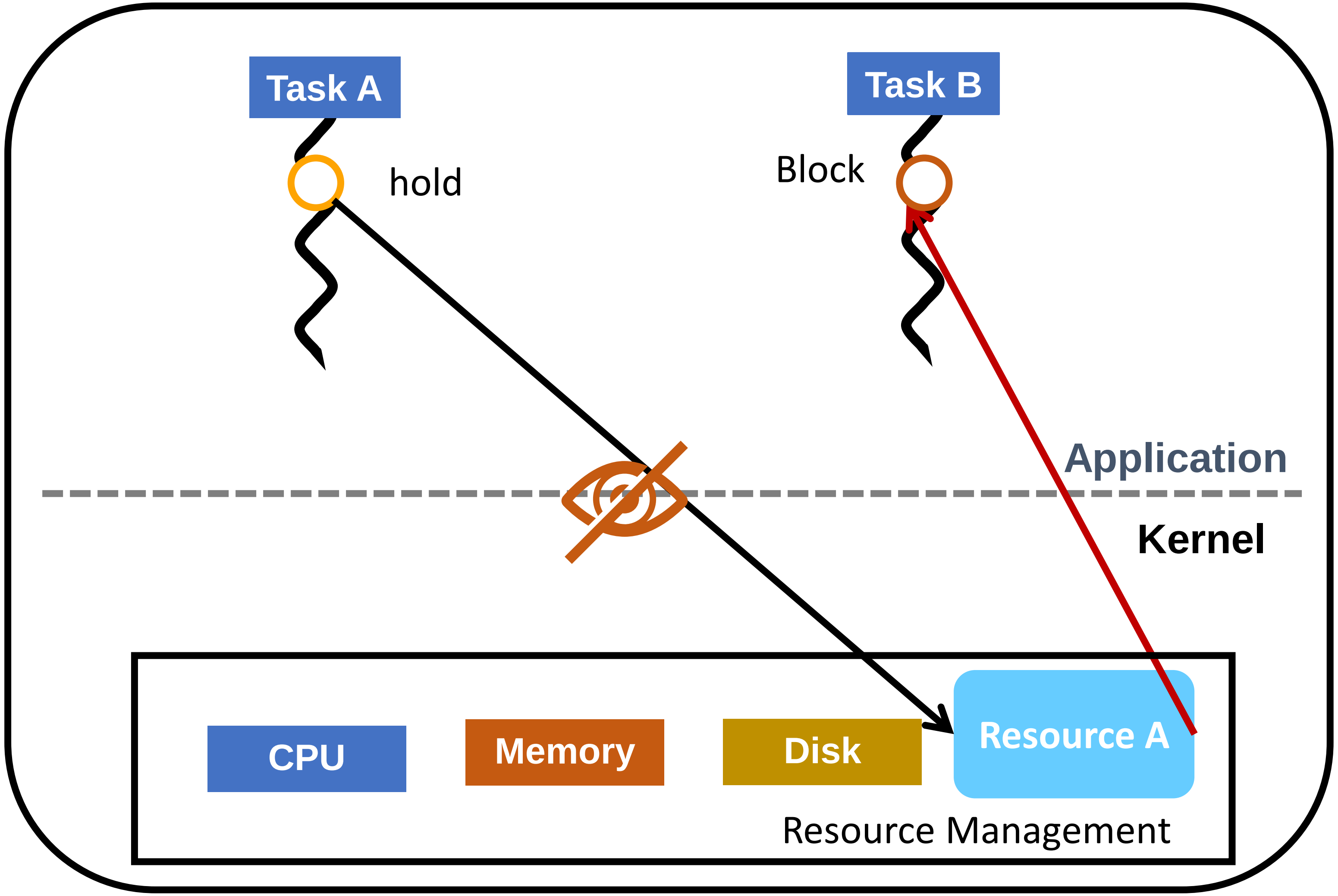
→ Design mechanism to **detect and mitigate** intra-app interference

Key Behavior of Intra-app Interference



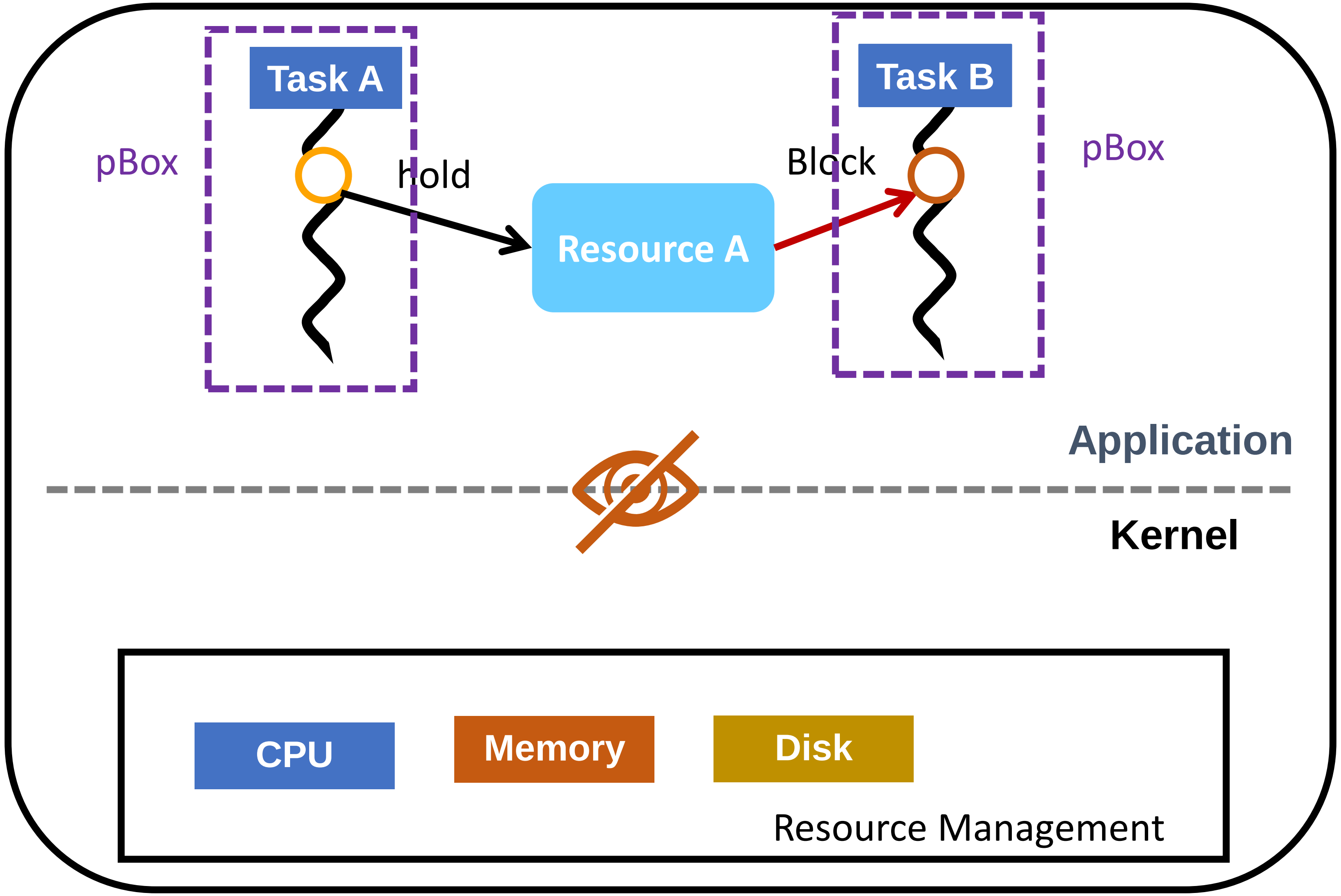
Key Behavior of Intra-app Interference

Insight: make the OS aware of virtual resource contention



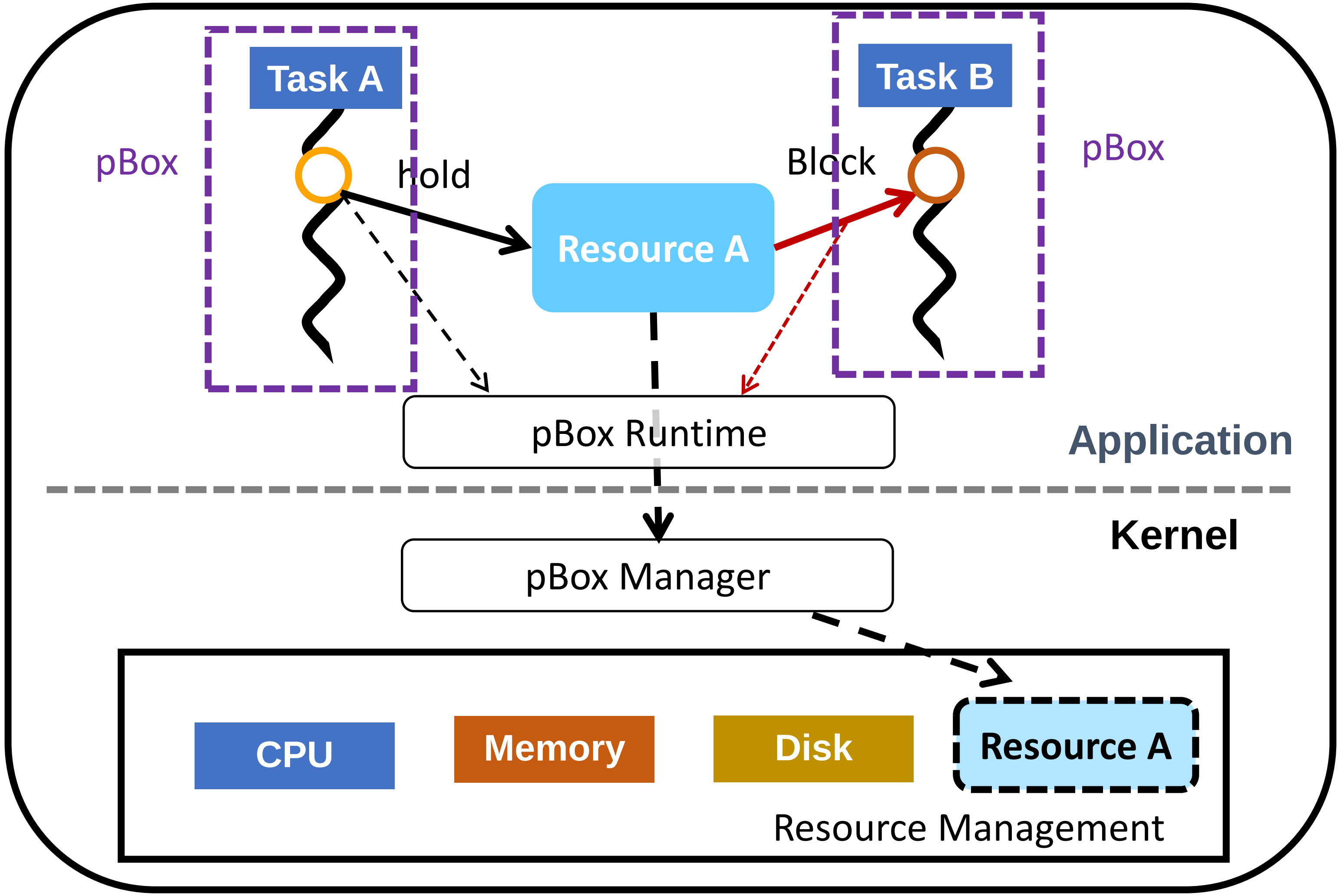
Performance Box

Design goal: monitor application resource contention and expose to Kernel



Performance Box

Design goal: monitor application resource contention and expose to Kernel



pBox APIs

1. Easily define the isolation boundary by developers
2. Automatically enforce performance isolation among pBox

Create	`int create_pbox(IsolationRule rule);` `int release_pbox(IsolationRule rule);`
Activate	`int activate_pbox(int psid) ;` `int freeze_pbox(int psid);`
Trace	`int update_pbox(size key, event_type event);`

pBox Creation

```
void add_connection(THD *thd) {
  …
  While (thd_connection_alive(thd)) {
    if (do_command(thd))
      break;
    end_connection(thd);
  }
  close_connection(thd);
}
```

Loop to handle all requests from one client

Example: MySQL thread handler for a client connection

pBox Creation

```
void add_connection(THD *thd) {
  rule = { .type = RELATIVE, .isolation_level = 50 };
  psid = create_pbox(rule);
  …
  While (thd_connection_alive(thd)) {
    if (do_command(thd))
      break;
    end_connection(thd);
  }
  close_connection(thd);
  release_pbox(psid);
}
```

Loop to handle all requests from one client

pBox Creation

```
void add_connection(THD *thd) {
  rule = { .type = RELATIVE, .isolation_level = 50 };
  psid = create_pbox(rule);
  …
  While (thd_connection_alive(thd)) {
    if (do_command(thd))
      break;
    end_connection(thd);
  }
  close_connection(thd);
  release_pbox(psid);
}
```

Isolation rule

Loop to handle all requests from one client

Handle one request

Example: MySQL thread handler for a client connection

Define pBox Isolation Area

```
bool do_command(THD *thd) {
  …
  command = thd->net.read_pos[0];
  ret = dispatch_command(command, thd, ...);
  …
}
```

Example: MySQL thread handler for a client connection

Define pBox Isolation Area

```
bool do_command(THD *thd) {
  …
  command = thd->net.read_pos[0];
  psid = get_current_pbox();
  …
  activate_pbox(psid);
  ret = dispatch_command(command, thd, ...);
  freeze_pbox(psid);
  …
}
```

Define pBox Isolation Area

```
bool do_command(THD *thd) {
  …
  command = thd->net.read_pos[0];
  psid = get_current_pbox();
  …
  activate_pbox(psid);
  ret = dispatch_command(command, thd, ...);
  freeze_pbox(psid);
  …
}
```

Additional code to filter out requests from pBox's isolation area

Example: MySQL thread handler for a client connection

pBox APIs for Thread Pool Model

API:
```
int bind_pbox(size_t key, unbind_flags flags)
int unbind_pbox(size_t key, unbind_flags flags)
```

Functions to transfer the ownership of pBox:

- `bind_pbox` finds the pBox from the key and binds it with the current thread
- `unbind_pbox` detaches the pBox from current thread
- Detach/attach to thread when the task is detached/attached from thread pool

How to Trace Application Resource

- **Require developer to expose resource usage**
  - High overhead to inform every changes of resource

- **Observation:**
  - two key questions for performance interference: which activity is causing **delay** and which one is **deferred**
  - But we also need to adapt different resource implementation, variable types and resource use pattern

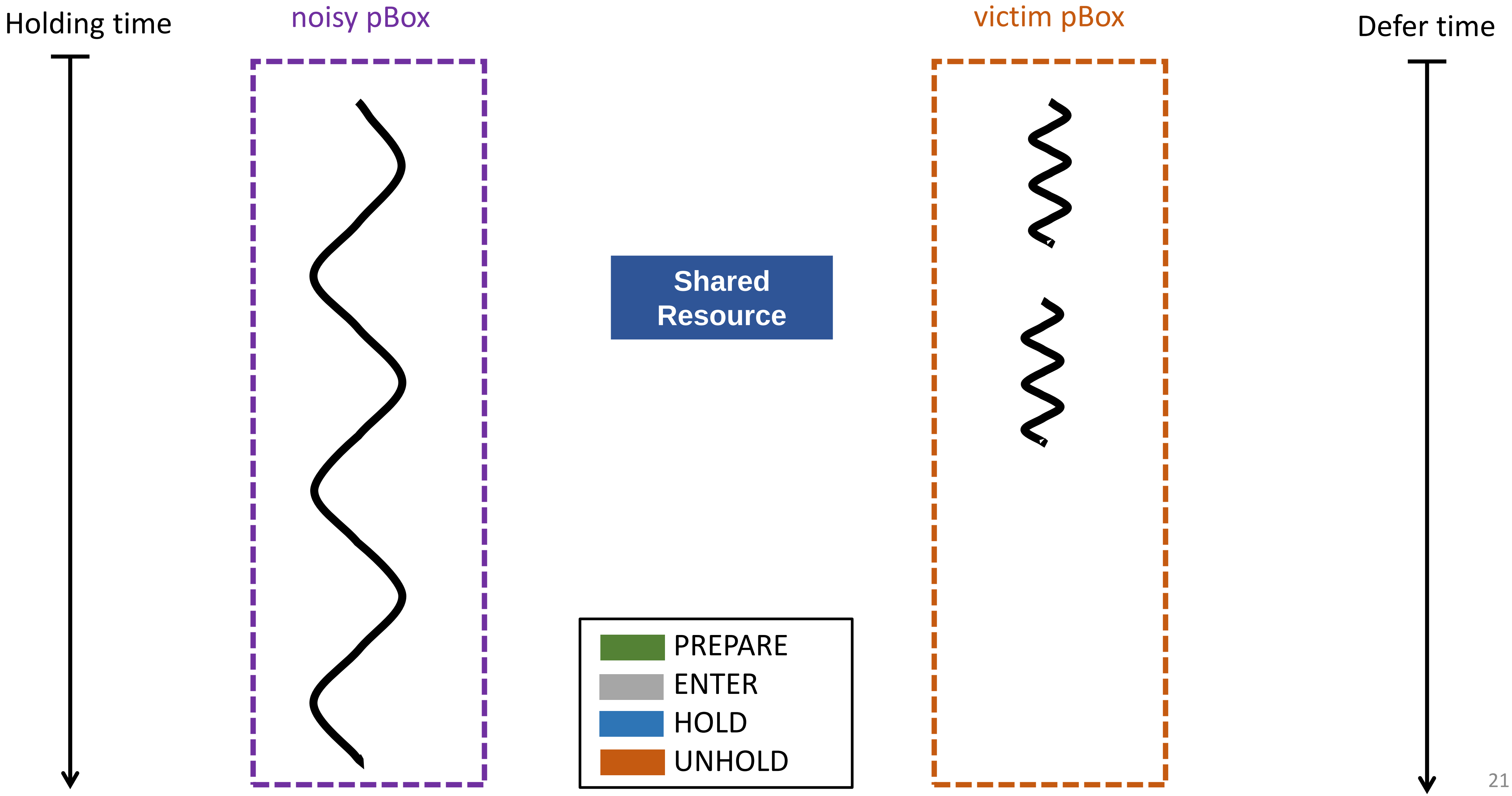
- **Our approach:**
  - A new concept, ***state event***, to capture key application resources event

State Event

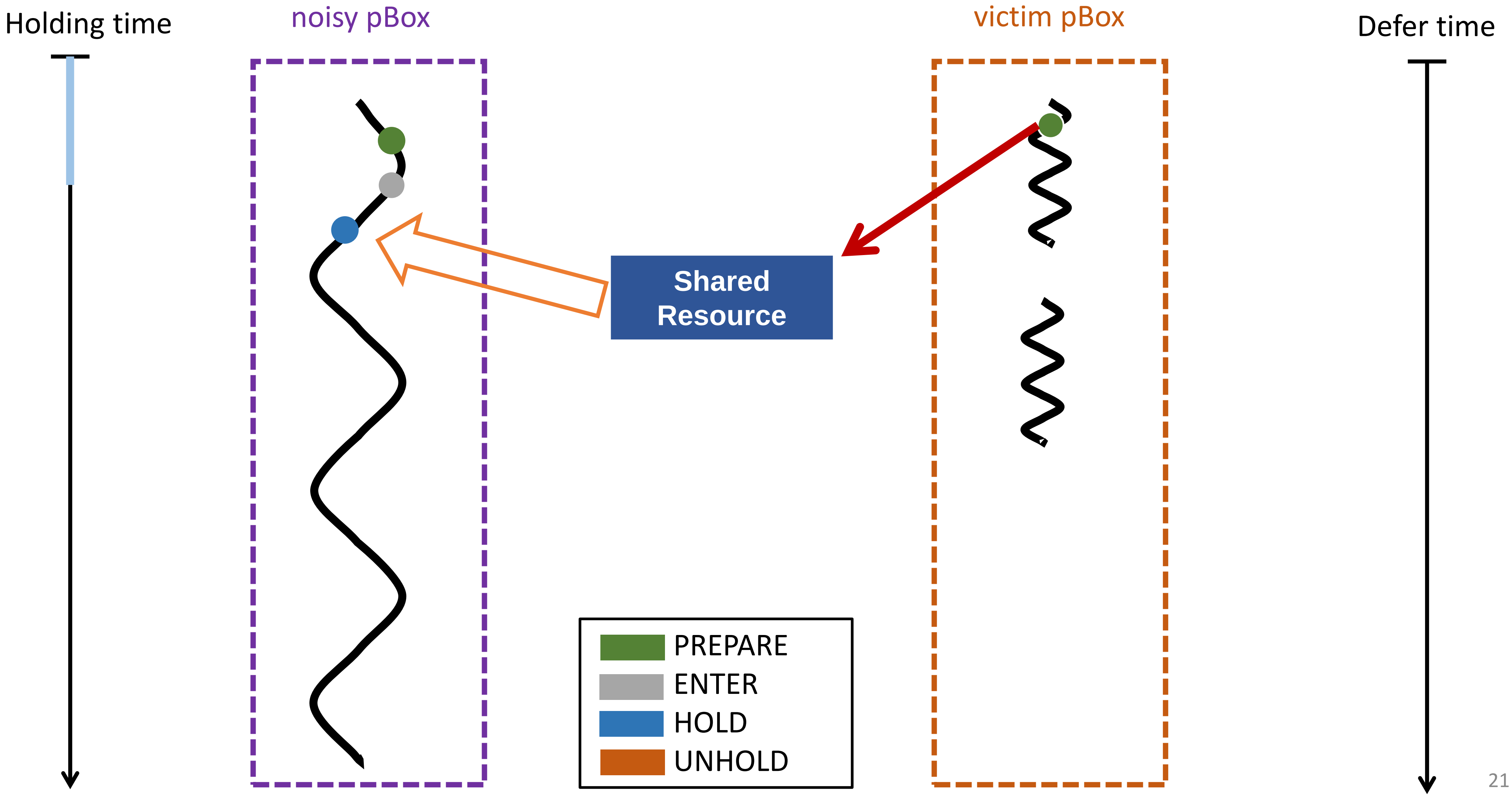
PREPARE	A pBox is derferred by an application resource
ENTER	A pBox is no longer derferred by an application resource
HOLD	A pBox is holding an application resource
UNHOLD	A pBox releases an application resource

API: `int update_pbox(size_t key, event_type event)`

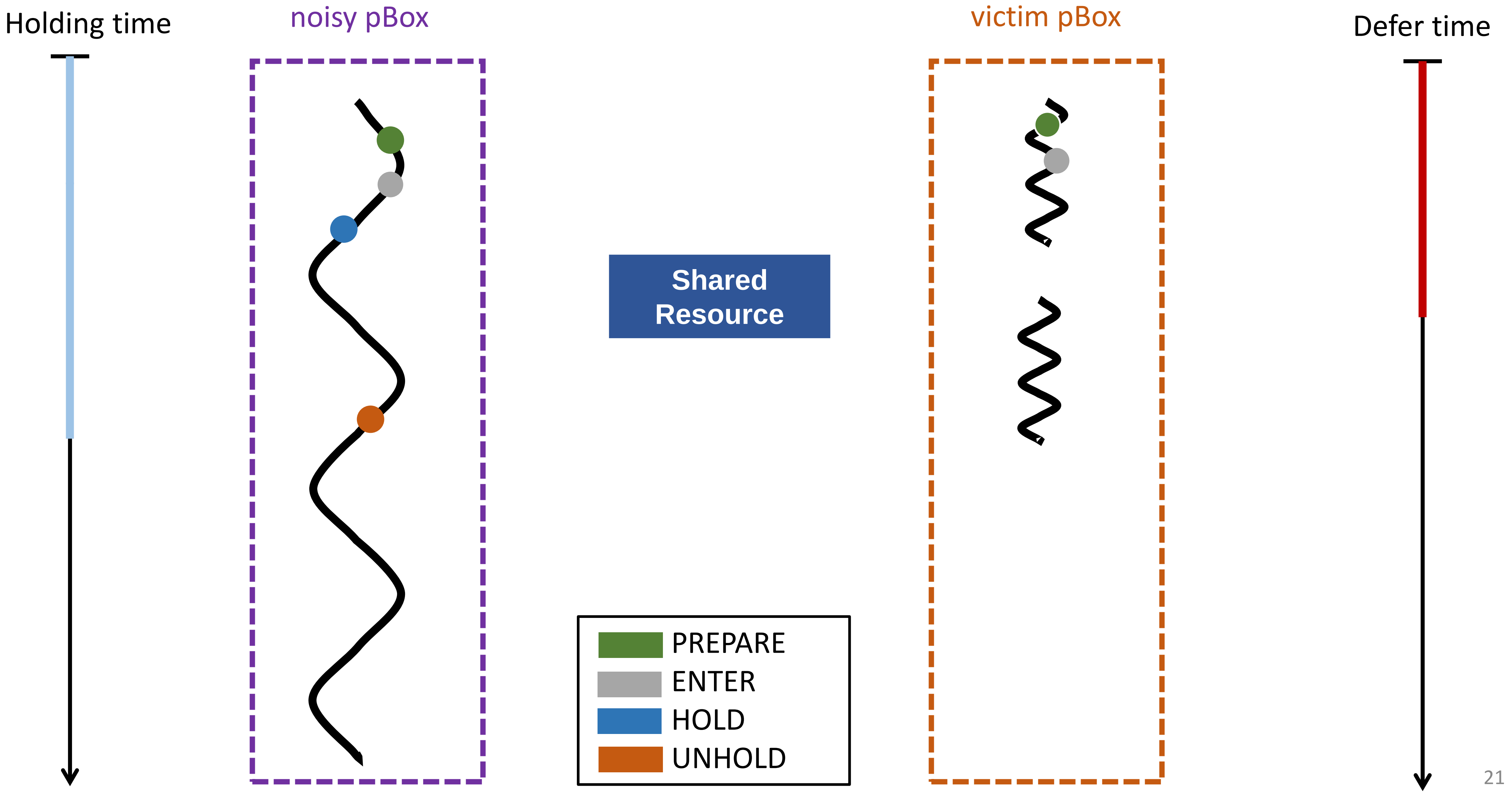
Tracing Deferring Time by State Event



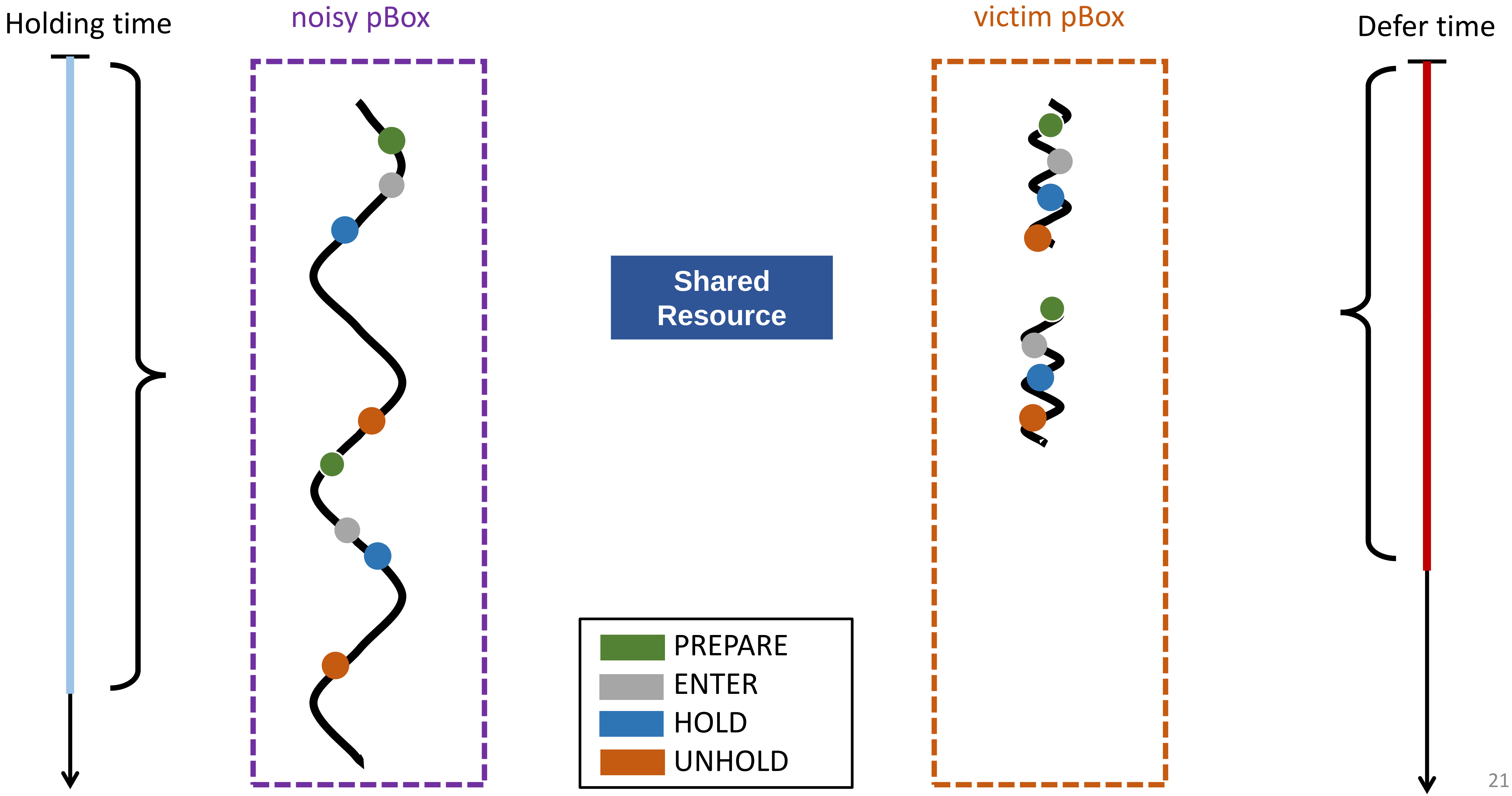
Tracing Deferring Time by State Event



Tracing Deferring Time by State Event



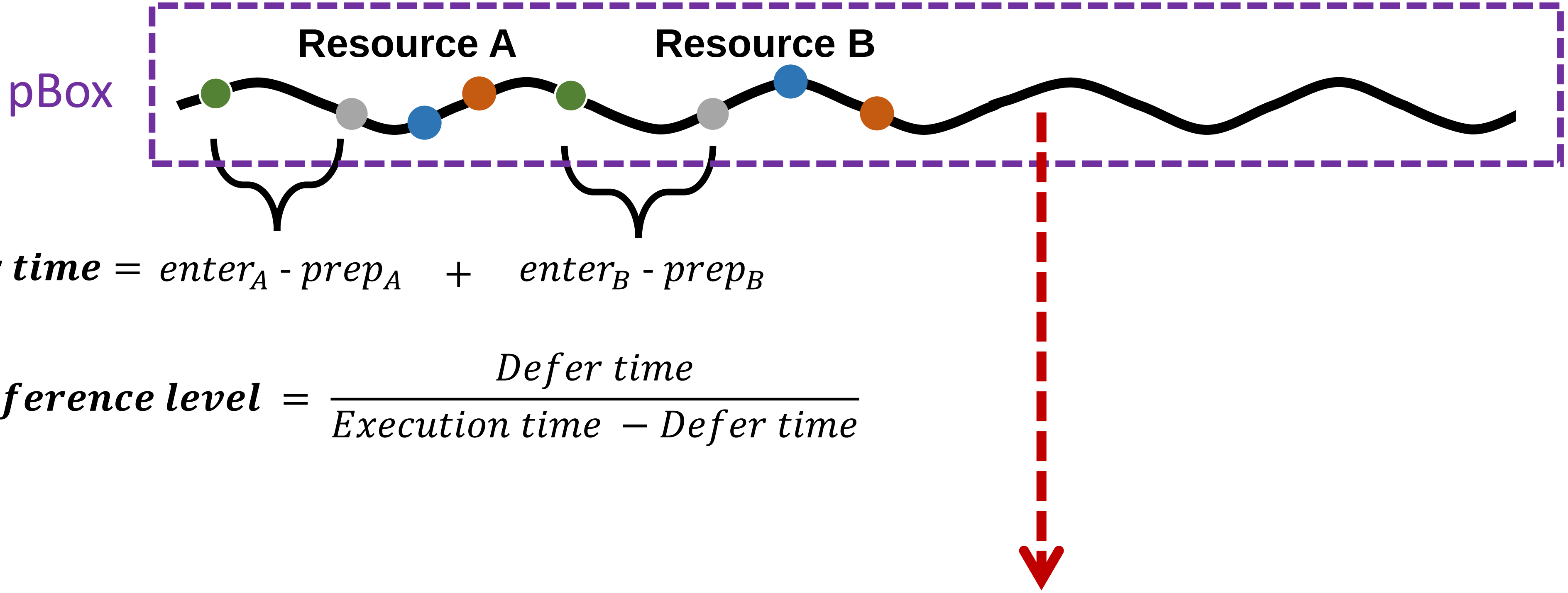
Tracing Deferring Time by State Event



Detecting Performance Interference for Activity

- **State event only trace deferring time on resource level**
  - Resource level interference $\neq$ end-to-end performance interference
  - No priori knowledge of future resource usage and interference level

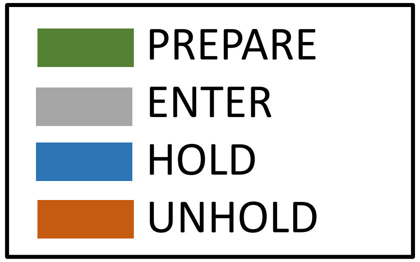
Activities' Interference Level



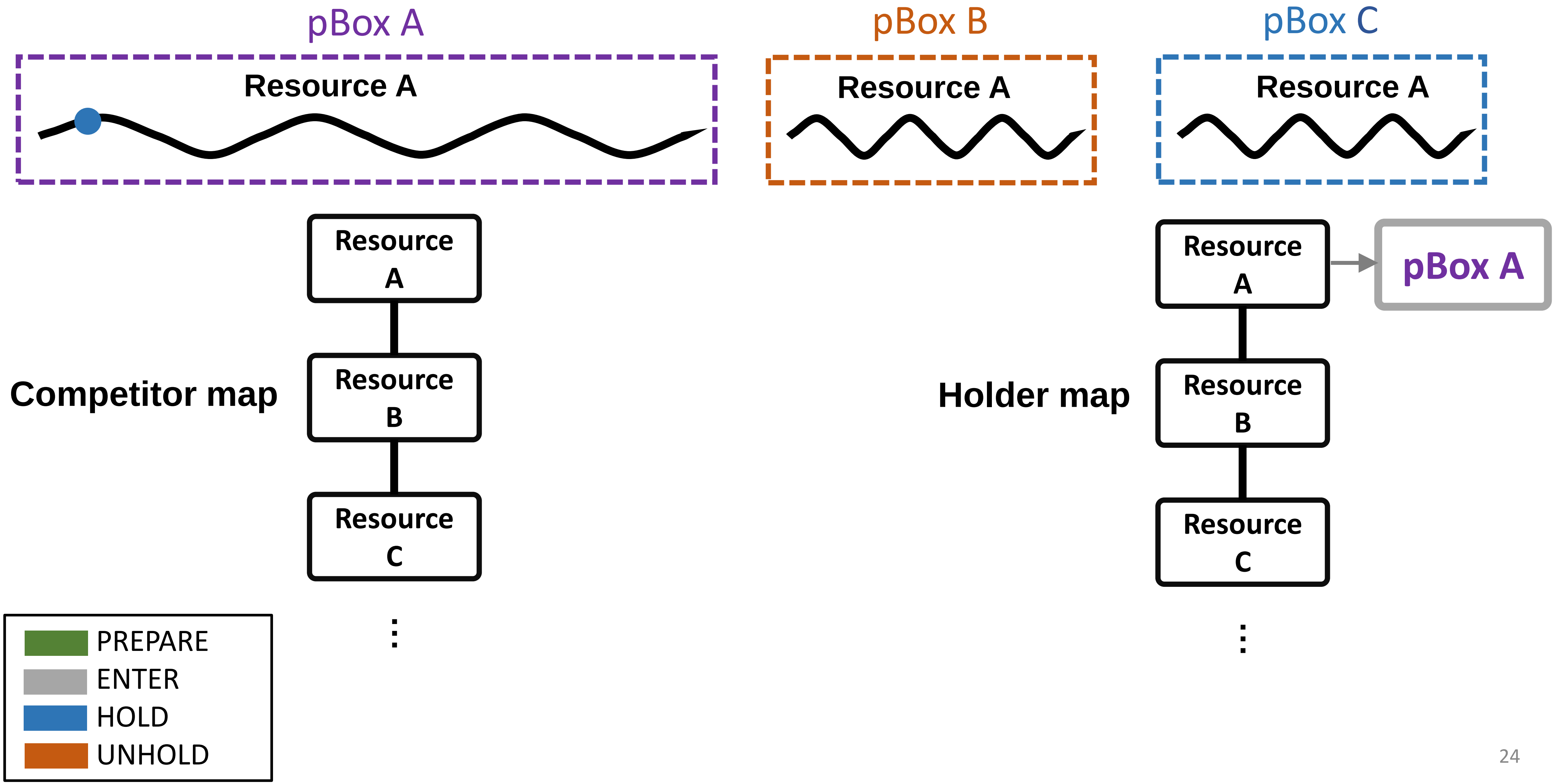
$$\boldsymbol{Defer\ time} = enter_A - prep_A \quad + \quad enter_B - prep_B$$

$$\boldsymbol{Interference\ level} = \frac{Defer\ time}{Execution\ time - Defer\ time}$$

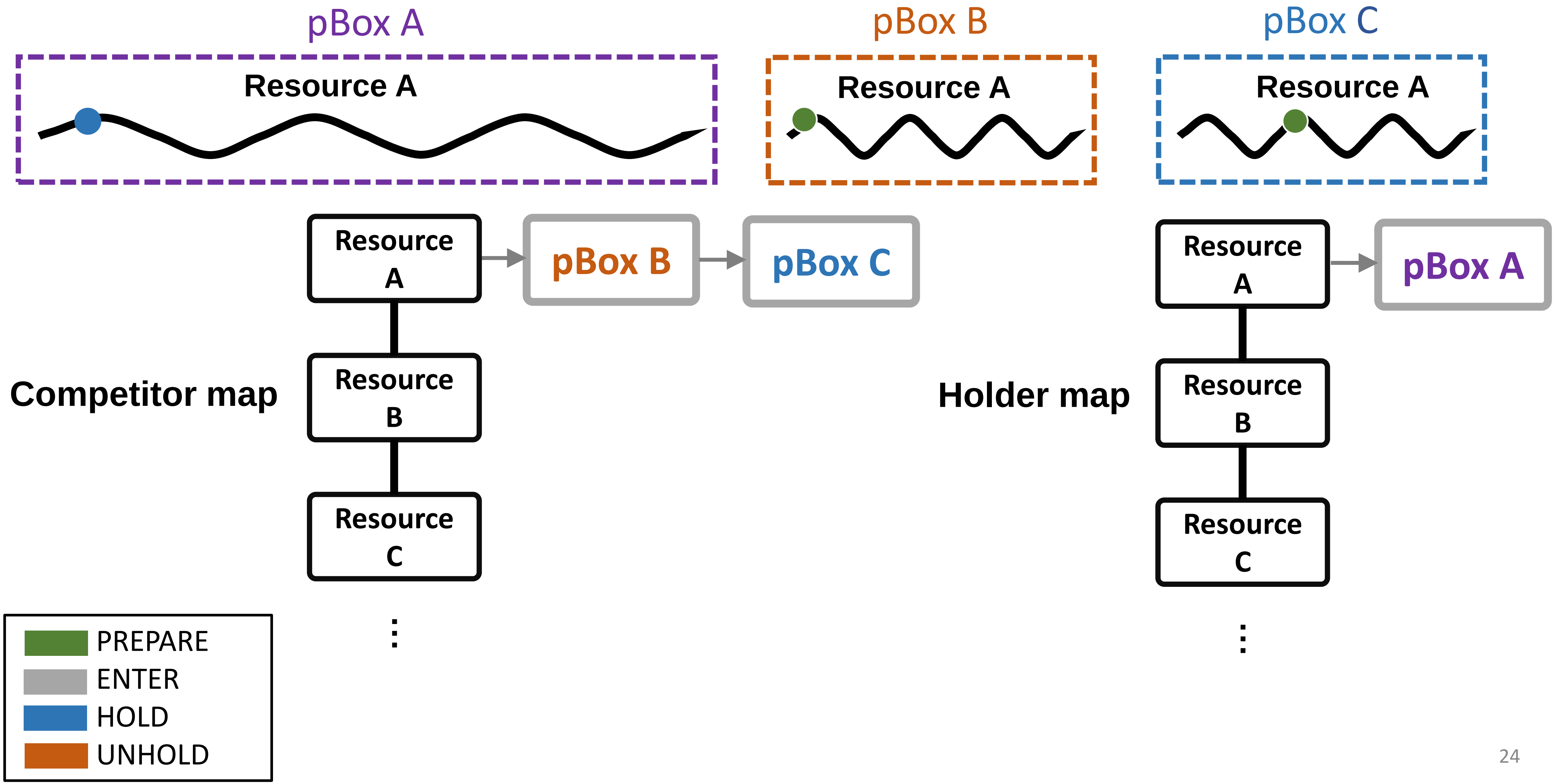
Worse-case analysis: if current $\boldsymbol{Interference\ level}$ is larger than isolation goal, it’s time to take action



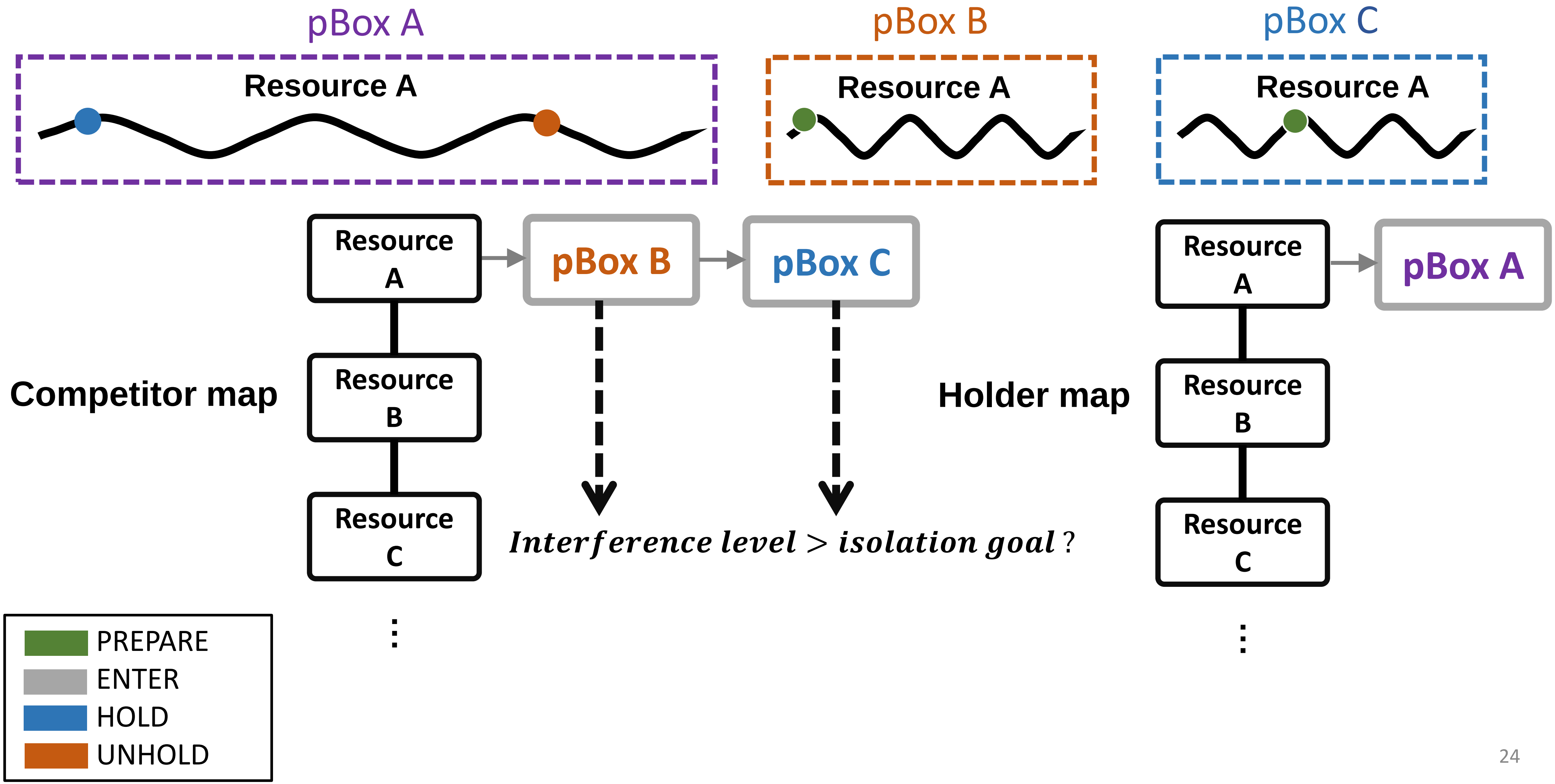
Competitor Map + Holder Map



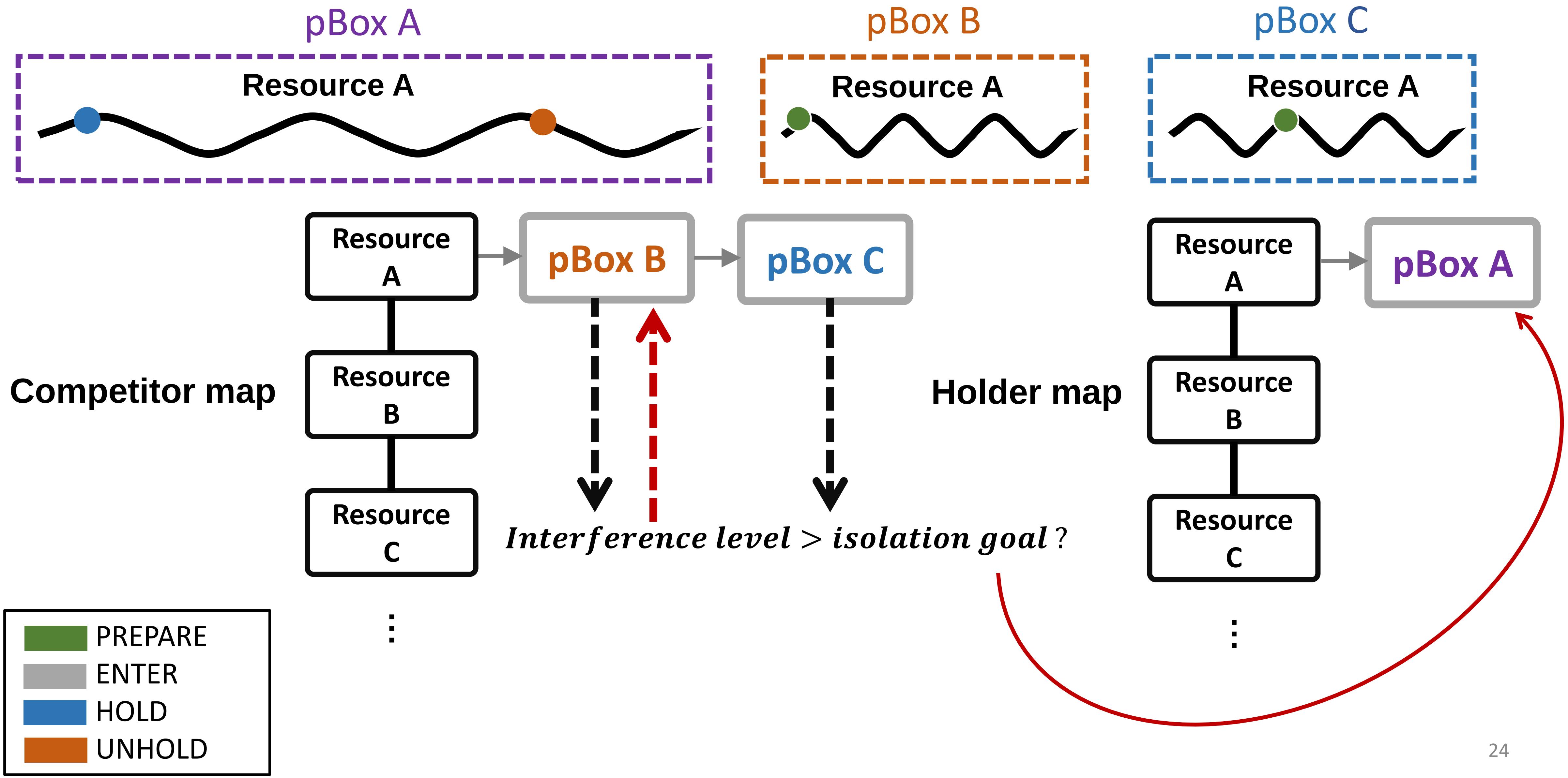
Competitor Map + Holder Map



Competitor Map + Holder Map



Competitor Map + Holder Map



Interference Mitigation

- **Reallocating application resource can introduce dangerous side effects to application**

- **Mitigating interference without breaking application logic**
  - Adding a delay to noisy activity
  - Only penalize the noisy pBox when a UNHOLD event is received

- **Handle nest state events**
  - A noisy pBox can hold multiple resources
  - Only penalize when pBox no longer holds any application resource

Score-based Penalty Length Adjustment

- **For *i* rounds of penalty, we calculate victim pbox**
  - $S_i = \frac{average\ deferring\ time}{average\ exeution\ time}$
  - $\boldsymbol{Penalty\ effectiveness\ Score} = \begin{cases} score + 1;\ if\ s(i+1) > s(i) \\ score\ - 1;\ if\ s(i+1) < s(i)\ and\ score > 1 \\ 1\ ; if\ s(i+1) < s(i)\ and\ score = 1 \end{cases}$
  - $\boldsymbol{Next\ penalty\ length} = \ current\ length\ \times (1 + score)$

Check paper for details

Other Optimization

- **Lightweight Tracing**
  - Pre-allocation for frequently used data struct to reduce the need for additional memory calls
  - Reduce the number of syscalls like *update_pbox*
  - Optimizing the datastruct

- **Lazy unbind**
  - unbind_pbox only marks a pbox as detached from thread
  - only detach when bind_pbox bind the pbox to a different thread

Evaluation

- **Can pBox reduce intra-application interference?**
- **How does pBox compare to state-of-art solutions?**
- **What is the overhead?**

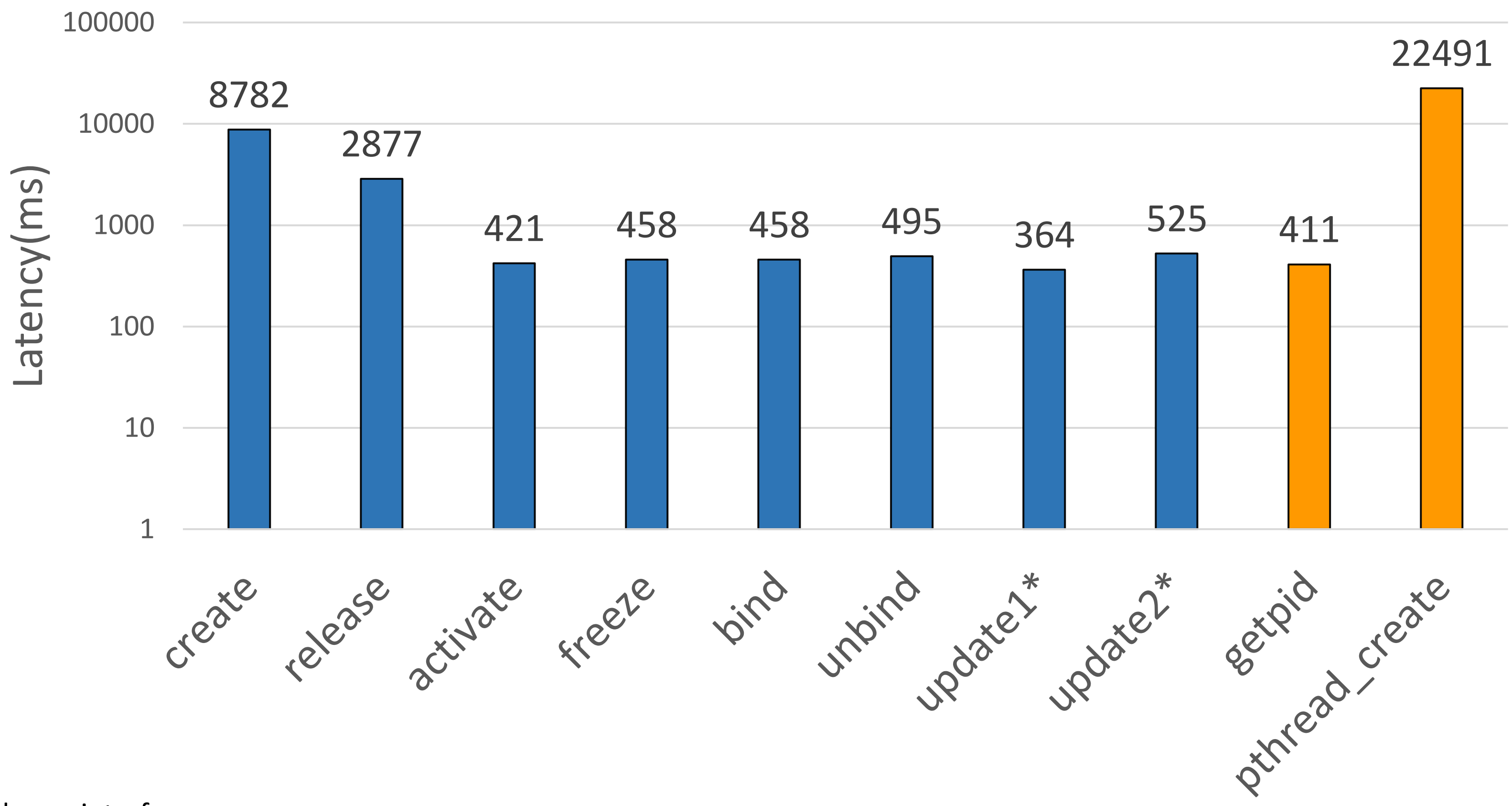
Experiment Setup

- **Implemented in Linux kernel 5.4.1 with a user-level library**
- **A Static analyzer to find the state event**
- **Ported to five systems**
  - MySQL, PostgreSQL, Apache, Varnish, Memcached

Software	SLOC	SLOC Added	Inspected Functions
MySQL	1.74M	192	83
PostgreSQL	629K	127	71
Apache	198K	71	43
Varnish	59K	77	53
Memcached	19K	70	22

Microbenchmark

- Test latency of pbox APIs compared with get_pid



update1* update_pbox under no interference
update2* update_pbox under interference

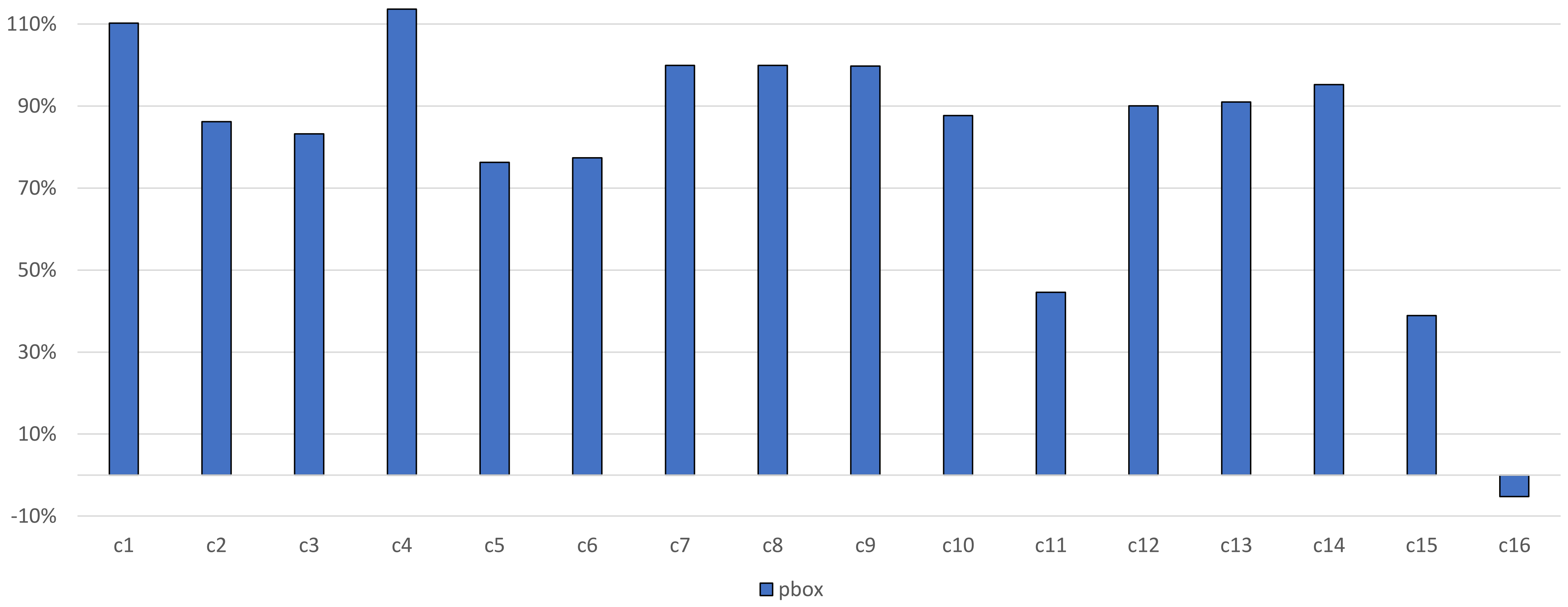
Real-world intra-app interference cases

ID	Application	Contending Resource	Description
C1	MySQL	table	Write query blocked by long update query
C2		global mutex	Inserting query on table without primary key has contention
C3		tickets	Query blocked on innodb thread concurrency
C4		transaction history length	SERIALIZABLE isolation causes overhead to read query
C5		UNDO log	Background purge task blocks client request
C6	PostgreSQL	index search tree	In-progress INSERT delayes other queries
C7		database table	Long update query blocks other requests
C8		database table	buffer content lock contention on SHARED lock
C9		dead table rows	Vacuum full process blocks other requests
C10		write-ahead log	A large WAL blocks requests
C11	Apache	fcgid request queue	slow request in mod_fcgid blocks other fast connections
C12		apache thread pools	Apache locks server if reaching maxclient
C13		php thread pool	Apache server slows due to contention on php connection
C14	Varnish	varnish thread pool	Slow request on visiting big objects block other request
C15		system lock	lock contention with high number of thread pools
C16	Memcached	system lock	lock contention in the cache replacement

Performance Interference Reduction

Reduction Ratio = $\frac{T_{Interference} - T_{Solution}}{T_{Interference} - T_{Normal}}$

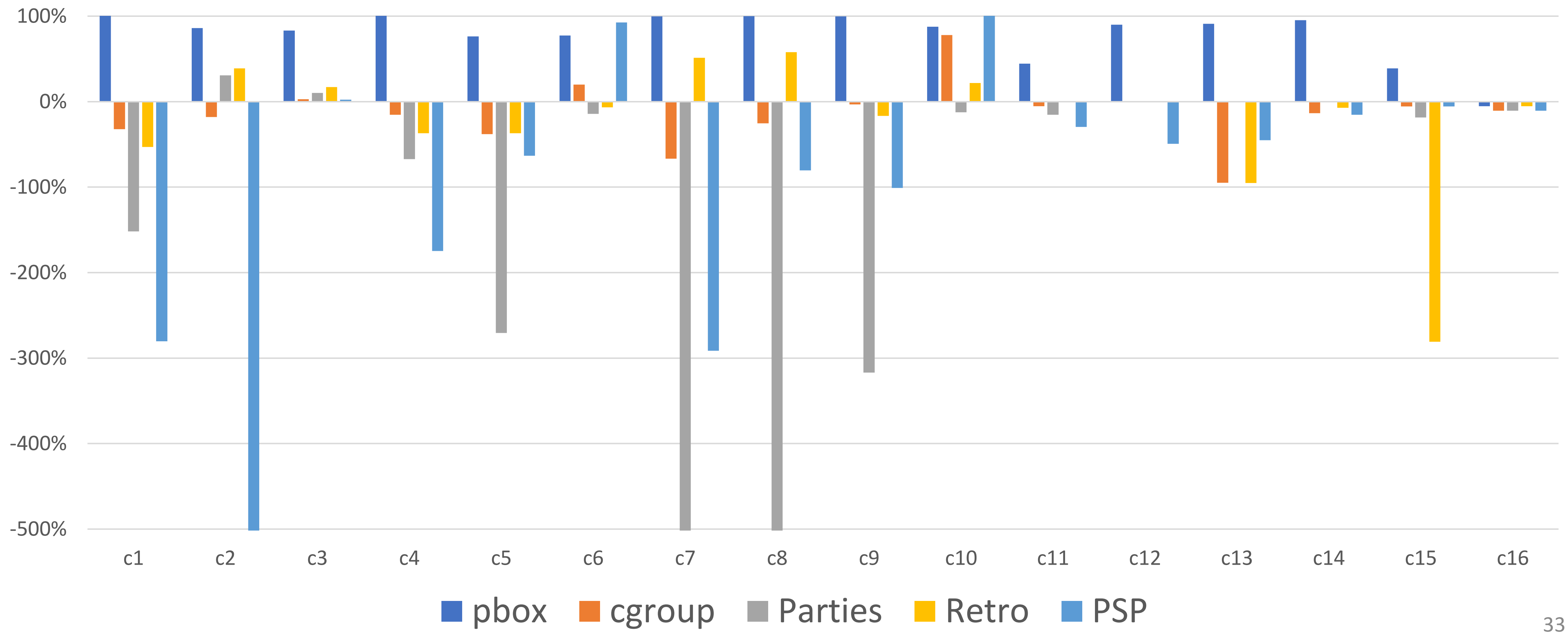
pBox: 86%



Performance Interference Reduction

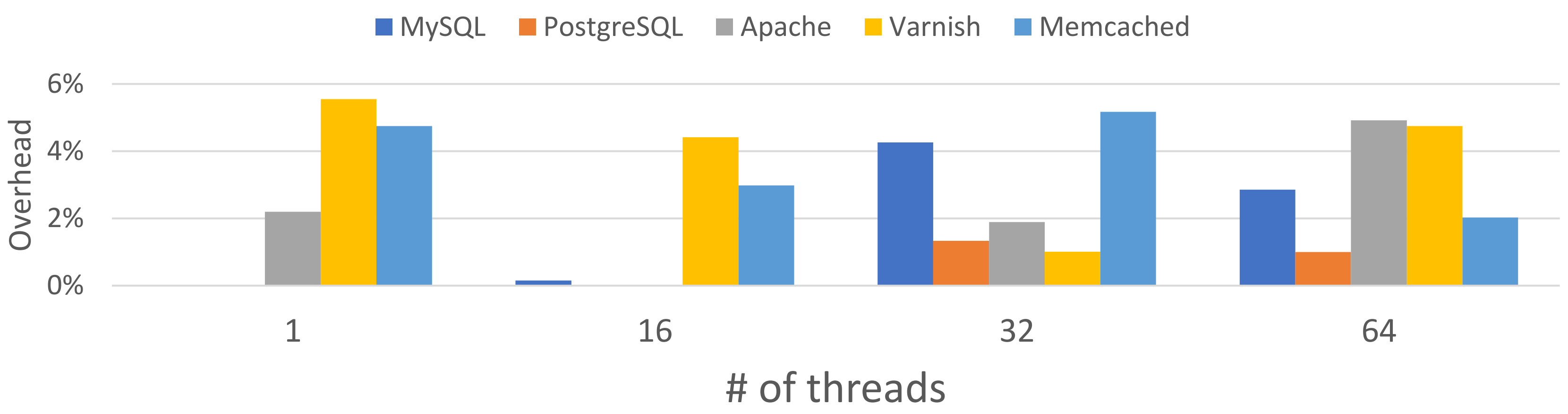
Number of case improved:

pBox (15/16) ; cgroup(3/16); PARTIES(3/16); Retro(5/16); DARC(3/16)

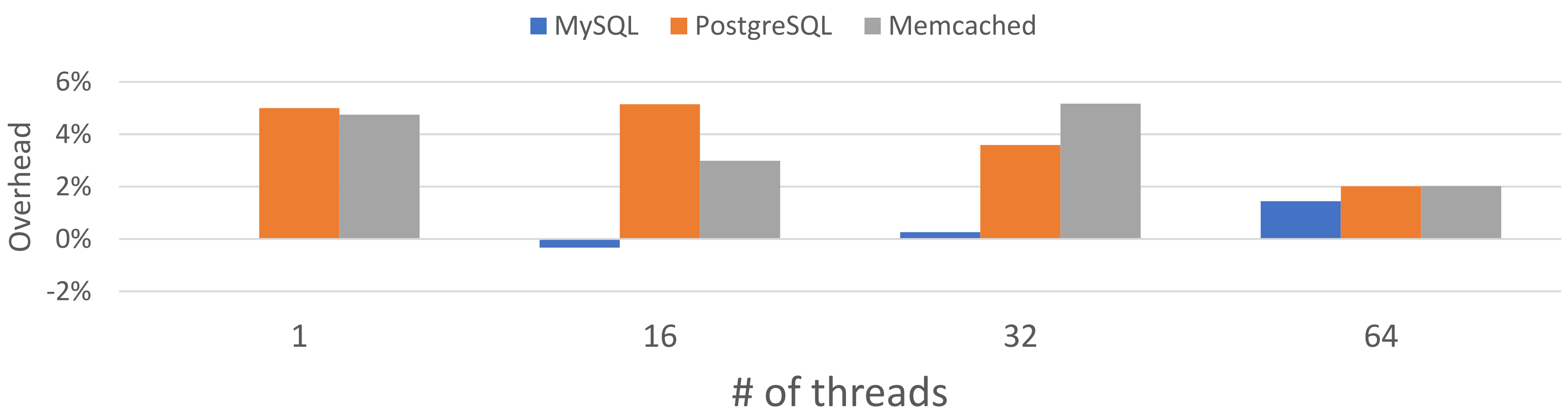


pBox Overhead

Read-intensive workload:



Write-intensive workload:



Conclusion

1. Intra-application performance interference is difficult to mitigate

2. Performance isolation needs to be enforced inside application

3. pBox, an abstraction to push performance isolation into application
    1. Make OS aware of application resource usage

4. pBox mitigate 15/16 interference cases with a 86% reduction ratio

Thank you!

Question List: Motivation

1. Why design pBox in kernel? Or Why pBox should be a OS abstraction

2. How does pBox deal with micro-second level activity

Question List: Design

1. How many manually effort to insert state event
2. How does developer know the correctness of instrumentation
3. Does pBox need a dedicate core? If so, what is the overhead
4. Would pBox make wrong penalty decision
5. Would pBox take penalty too late
6. How does pBox support event driven

Question List: Evaluation

1. It is unclear how pBox behaves for intra-application interference

2. What will happen if pBox policy make wrong decision

3. Is there any parameter in pBox

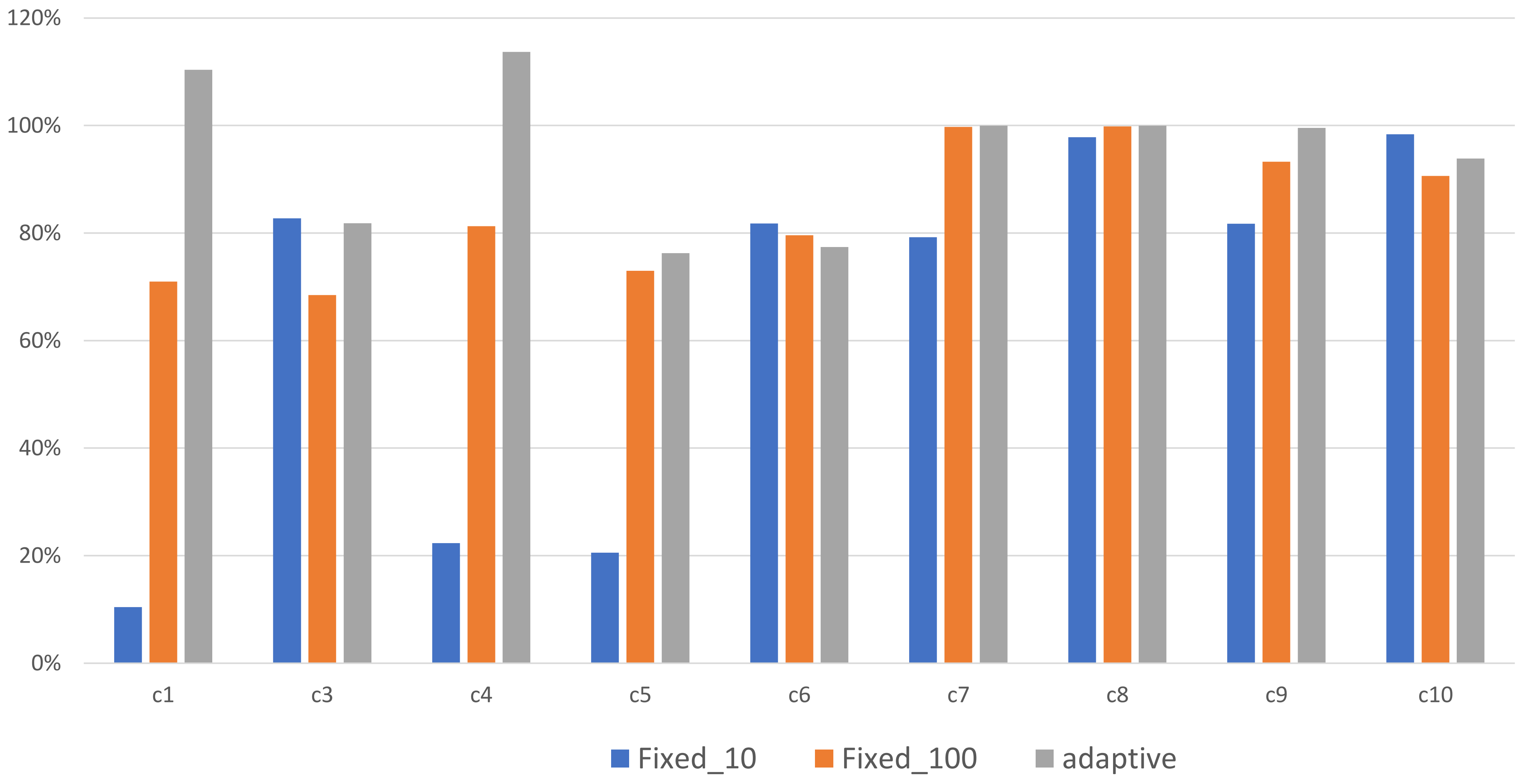
4. Why you only test on 15 cases

Static analyzer

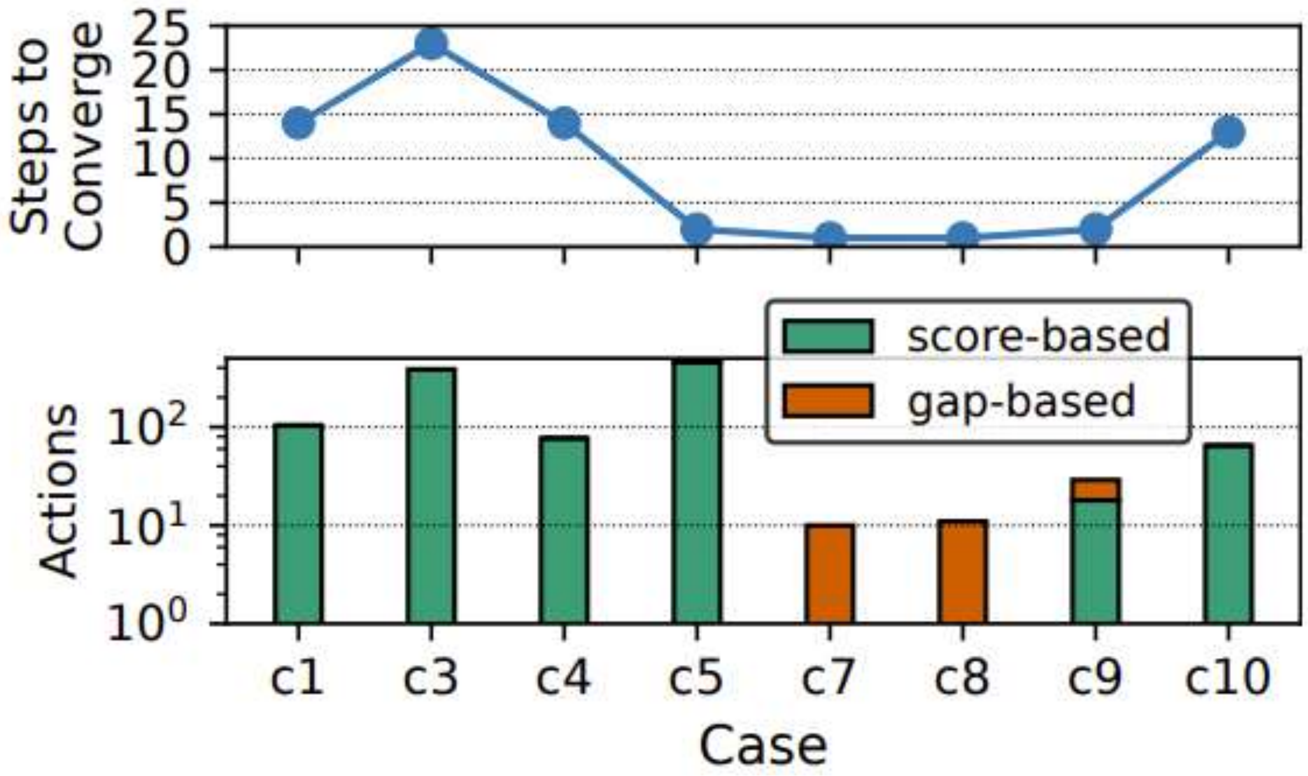
Software	Inspected Functions	State Events		SLOC Added
		Manual	Detected	
MySQL	83	57	40 (70%)	192
PostgreSQL	71	40	44 (110%)	127
Apache	43	12	8 (66%)	71
Varnish	53	16	12 (75%)	77
Memcached	22	14	12 (85%)	70

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

Effectiveness of Different Penalty Length



Penalty



Steps to Converge
25
20
15
10
5
0
score-based
gap-based
Actions
10^2
10^1
10^0
c1
c3
c4
c5
c7
c8
c9
c10
Case

Rule

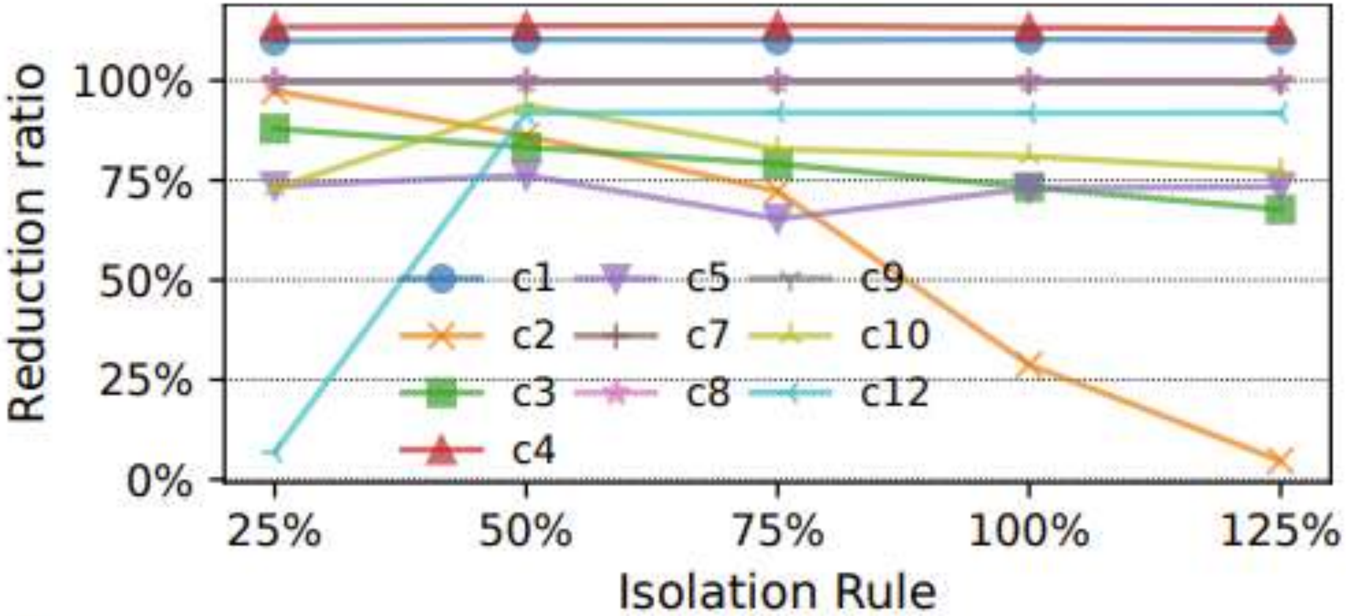


Figure 15. Interference reduction ratios for ten cases under different isolation rules from 25% to 125%. The default is 50%.

How to Find State Event

- **Manually instrumenting all state events in code base**
  - To much domain knowledge
  - A lot of resource -> waste time
  - Instrumentation can be error-prone

Notify State Event to pBox Manager

```
void srv_enter_innodb() {
  …
  for(;;) {
    if(srv_conc.n_active < thread_concurrency) {
      n_active = os_atomic_inc(&srv_conc.n_active);
      if(n_active <= thread_concurrency) {
        srv_enter_innodb_with_tickets(trx);
        return;
      }
    }
    os_thread_sleep(sleep_in_us);
  }
  …
}

void srv_exit_innodb() {
  …
  os_atomic_dec(&srv_conc.n_active, 1);
  …
}
```

get the thread tickets, enter innodb

no thread tickets, block itself

Notify State Event to pBox Manager

```
void srv_enter_innodb() {
  update_pbox(&srv_conc.n_active, PREPARE);
  for(;;) {
    if(srv_conc.n_active < thread_concurrency) {
      n_active = os_atomic_inc(&srv_conc.n_active);
      update_pbox(&srv_conc.n_active, HOLD);
      if(n_active <= thread_concurrency) {
        update_pbox(&srv_conc.n_active, ENTER);
        srv_enter_innodb_with_tickets(trx);
        return;
      }
    }
    os_thread_sleep(sleep_in_us);
  }
}

void srv_exit_innodb() {
  os_atomic_dec(&srv_conc.n_active, 1);
  update_pbox(&srv_conc.n_active, UNHOLD);
}
```

get the thread tickets, enter innodb

no thread tickets, block itself

Compiler Support

- Locate the resource variable
  - Find block function
  - Find the loop that uses block function
  - Check the conditional variable

```
void srv_enter_innodb() {
  …
  for(;;) {
    if(srv_conc.n_active < thread_concurrency) {
      n_active = os_atomic_inc(&srv_conc.n_active);
      if(n_active <= thread_concurrency) {
        srv_enter_innodb_with_tickets(trx);
        return;
      }
    }
    os_thread_sleep(sleep_in_us);
  }
  …
}

void srv_exit_innodb() {
  …
  os_atomic_dec(&srv_conc.n_active, 1);
  …
}
```

Compiler Support

- Locate the resource variable
  - Find block function
  - Find the loop that uses block function
  - Check the conditional variable
- Locate the inserting point
  - Find the hold operation for conditional variable
  - Find the unhold operation for conditional variable

```
void srv_enter_innodb() {
  …
  for(;;) {
    if(srv_conc.n_active < thread_concurrency) {
      n_active = os_atomic_inc(&srv_conc.n_active);
      if(n_active <= thread_concurrency) {
        srv_enter_innodb_with_tickets(trx);
        return;
      }
    }
    os_thread_sleep(sleep_in_us);
  }
  …
}

void srv_exit_innodb() {
  …
  os_atomic_dec(&srv_conc.n_active, 1);
  …
}
```