Pushing Performance Isolation Boundaries into Application with pBox

Yigong Hu, Gongqi Huang, Ryan Huang
Performance Isolation Is Critical in Application

Inter-application performance interference

Shared Resource

*S 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

MySQL

Continuous write queries
Intra-app Interference Example

MySQL

UNDO log

Long read Transaction

Continuous write queries

Long read transaction join

Query per second

Times (s)
Intra-app Interference Example

MySQL

Purge thread triggered

4X throughput decreases

Long read transaction join

Continuous write queries

Purge thread

Long read Transaction
Intra-app Interference Is Prevalent

MySQL performance implications of InnoDB isolation modes

January 14, 2015

[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. Oracle will need to skip the blocking transaction and delete the UNDO entries that are no longer needed. It is contained in several ML # entries, like 5742.

In progress INSERT wrecks plans on table

Lists:pgsql-hackerspgsql-performance

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

2006

Mess with innodb_thread_concurrency
May 12, 2006

In MySQL 5.0 the meaning of innodb_thread_concurrency was changed (again!).
Now innodb_thread_concurrency is the number of concurrent threads instead of threads.

2014

Database Administrators
Hyperthreading & MySQL InnoDB Thread Concurrency Performance

2017

What are the right ways to analyse and tune innodb_thread_concurrency in
mysql 5.7?

2020

Deprecate and ignore options for InnoDB concurrency throttling
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

Application
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 – \textit{pBox}

- Difficult to enforce isolation inside applications
- Let developers \textbf{define} performance isolation domain in \textit{application}
- Invisible to diverse application-level virtual resources
- Expose set of \textbf{APIs} to easily trace application resource usage
- Insufficient and inflexible to enforce resource quota
- Design mechanism to \textbf{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
Design goal: monitor application resource contention and expose to Kernel
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

<table>
<thead>
<tr>
<th></th>
<th>int create_pbox(IsolationRule rule);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>int release_pbox(IsolationRule rule);</td>
</tr>
<tr>
<td></td>
<td>int activate_pbox(int psid) ;</td>
</tr>
<tr>
<td>Activate</td>
<td>int freeze_pbox(int psid);</td>
</tr>
<tr>
<td>Trace</td>
<td>int update_pbox(size key, event_type event);</td>
</tr>
</tbody>
</table>
pBox Creation

```c
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

```c
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

```c
void add_connection(THD *thd) {
    rule = { .type = RELATIVE, .isolation_level = 50 };  // Isolation rule
    psid = create_pbox(rule);
    ...
    While (thd_connection_alive(thd)) {
        if (do_command(thd))  // Handle one request
            break;
        end_connection(thd);
    }
    close_connection(thd);  // Loop to handle all requests from one client
    release_pbox(psid);
}
```

Example: MySQL thread handler for a client connection
Define pBox Isolation Area

```c
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

```c
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

```c
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:**

```c
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
  o High overhead to inform every changes of resource

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

• Our approach:
  o A new concept, state event, to capture key application resources event
# State Event

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREPARE</td>
<td>A pBox is <strong>deferred</strong> by an application resource</td>
</tr>
<tr>
<td>ENTER</td>
<td>A pBox is <strong>no longer</strong> deferred by an application resource</td>
</tr>
<tr>
<td>HOLD</td>
<td>A pBox is <strong>holding</strong> an application resource</td>
</tr>
<tr>
<td>UNHOLD</td>
<td>A pBox <strong>releases</strong> an application resource</td>
</tr>
</tbody>
</table>

**API:**

```c
int update_pbox(size_t key, event_type event)
```
Tracing Deferring Time by State Event

Holding time

noisy pBox

Shared Resource

victim pBox

Defer time

- PREPARE
- ENTER
- HOLD
- UNHOLD
Tracing Deferring Time by State Event

Holding time

noisy pBox

Shared Resource

victim pBox

Defer time

- PREPARE
- ENTER
- HOLD
- UNHOLD
Tracing Deferring Time by State Event

Holding time

noisy pBox

Shared Resource

victim pBox

Defer time

- PREPARE
- ENTER
- HOLD
- UNHOLD
Tracing Deferring Time by State Event

Holding time

noisy pBox

Shared Resource

victim pBox

Defer time

PREPARE
ENTER
HOLD
UNHOLD
Detecting Performance Interference for Activity

- **State event only trace deferring time on resource level**
  - Resource level interference ≠ end-to-end performance interference
  - No priori knowledge of future resource usage and interference level
Activities' Interference Level

\[ \text{Interference level} = \frac{\text{Defer time}}{\text{Execution time} - \text{Defer time}} \]

\( \text{Defer time} = \text{enter}_A - \text{prep}_A + \text{enter}_B - \text{prep}_B \)

**Worse-case analysis**: if current *Interference level* is larger than isolation goal, it’s time to take action
Competitor Map + Holder Map

Competitor map

Resource A

Resource B

Resource C

pBox A

Resource A

Resource B

Resource C

pBox B

Resource A

Resource B

Resource C

pBox C

PREPARE
ENTER
HOLD
UNHOLD
Competitor Map + Holder Map

**Competitor map**

```
Resource A
Resource B
Resource C
```

**Holder map**

```
Resource A
Resource B
Resource C
```

**pBox A**

```
Resource A
```

**pBox B**

```
Resource A
```

**pBox C**

```
Resource A
```

**Interference level > isolation goal?**
Competitor Map + Holder Map

**Competitor map**

- Resource A
- Resource B
- Resource C

**Holder map**

- Resource A
- Resource B
- Resource C

Interference level > isolation goal?

Legend:
- PREPARE
- ENTER
- HOLD
- UNHOLD
Interference Mitigation

• Reallocating application resource can introduce dangerous side effects to application

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

• **Handle nest state events**
  o A noisy pBox can hold multiple resources
  o Only penalize when pBox no longer holds any application resource
Score-based Penalty Length Adjustment

- For $i$ rounds of penalty, we calculate victim $pbox$

  \[ S_i = \frac{\text{average deferring time}}{\text{average execution time}} \]

  - Penalty effectiveness Score $= \begin{cases} 
  \text{score} + 1; & \text{if } s(i + 1) > s(i) \\
  \text{score} - 1; & \text{if } s(i + 1) < s(i) \text{ and } \text{score} > 1 \\
  1; & \text{if } s(i + 1) < s(i) \text{ and } \text{score} = 1 
\end{cases}$

- Next penalty length $= \text{current length} \times (1 + \text{score})$

Check paper for details
Other Optimization

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

• **Lazy unbind**
  o *unbind_pbox* only marks a pbox as detached from thread
  o 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

<table>
<thead>
<tr>
<th>Software</th>
<th>SLOC</th>
<th>SLOC Added</th>
<th>Inspected Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>1.74M</td>
<td>192</td>
<td>83</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>629K</td>
<td>127</td>
<td>71</td>
</tr>
<tr>
<td>Apache</td>
<td>198K</td>
<td>71</td>
<td>43</td>
</tr>
<tr>
<td>Varnish</td>
<td>59K</td>
<td>77</td>
<td>53</td>
</tr>
<tr>
<td>Memcached</td>
<td>19K</td>
<td>70</td>
<td>22</td>
</tr>
</tbody>
</table>
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

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

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

pBox: 86%
Number of case improved:
- pBox (15/16)
- cgroup (3/16)
- PARTIES (3/16)
- Retro (5/16)
- DARC (3/16)
pBox Overhead

Read-intensive workload:

![Read-intensive workload diagram]

Write-intensive workload:

![Write-intensive workload diagram]
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

<table>
<thead>
<tr>
<th>Software</th>
<th>Inspected Functions</th>
<th>State Events</th>
<th>SLOC Added</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Manual</td>
<td>Detected</td>
</tr>
<tr>
<td>MySQL</td>
<td>83</td>
<td>57</td>
<td>40 (70%)</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>71</td>
<td>40</td>
<td>44 (110%)</td>
</tr>
<tr>
<td>Apache</td>
<td>43</td>
<td>12</td>
<td>8 (66%)</td>
</tr>
<tr>
<td>Varnish</td>
<td>53</td>
<td>16</td>
<td>12 (75%)</td>
</tr>
<tr>
<td>Memcached</td>
<td>22</td>
<td>14</td>
<td>12 (85%)</td>
</tr>
</tbody>
</table>

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

- **c1**
- **c3**
- **c4**
- **c5**
- **c6**
- **c7**
- **c8**
- **c9**
- **c10**

Categories:
- Fixed_10
- Fixed_100
- adaptive
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
  o Too much domain knowledge
  o A lot of resource -> waste time
  o Instrumentation can be error-prone
void srv_enter_innodb() {
    ...
    for(;;) {
        if(srv_conc.n_active < thread_concurrency)
            { get the thread tickets, enter innodb
                n_active = os_atomic_inc(&srv_conc.n_active);
                if(n_active <= thread_concurrency)
                    { no thread tickets, block itself
                        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);
    ...
}
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);
}
Compiler Support

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

```c
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);
    ...
}
```
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;
            }
        }
    }
}

void srv_exit_innodb() {
    ...
}