Scaling Guest OS Critical Sections with eCS

Sanidhya Kashyap, Changwoo Min, Taesoo Kim
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction

![Diagram showing physical and virtual CPU abstraction](image-url)
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction

- VM consolidation
  - Contention on pCPU
The physical and virtual CPU abstraction

A vCPU can be preempted without notification

- VM consolidation
  - Contention on pCPU
The physical and virtual CPU abstraction

A vCPU can be preempted without notification

Double scheduling issue
Double scheduling: Lock holder preemption (LHP)

- vCPU holding a lock is preempted
- Preemption hinders forward progress of the VM
- Can lead to application slowdown by 20 -- 130%
Efforts to mitigate preemption issues

Research efforts
- Focussed only non-blocking locks
  - Acquire iff sufficient schedule time
- Hotplug vCPUs on the fly
  - May not scale to large vCPU VMs
- VM co-scheduling
  - Does not always alleviate the issue

Current practice
- Mostly address other preemption problem
  - Blocking locks
  - Unfair non-blocking locks
- Hardware features to mitigate preemptions
Efforts to mitigate preemption issues

Research efforts
- Focussed only non-blocking locks
  - Acquire iff sufficient schedule time
- Hotplug vCPUs on the fly

Current practice
- Mostly address other preemption problem
  - Blocking locks

Prior approaches are mostly specialized
Still the double scheduling is looming!

- LHP for blocking locks
  - mutex, rwsem
- Readers preemption (RP) in read-write locks
  - A reader is preempted while holding the lock
- Interrupt context preemption (ICP)
  - Preemption of a vCPU processing an interrupt
- Blocked-waiter wakeup (BWW)
  - Waking up a blocked thread on an idle vCPU is at least 10 times costlier
Still the double scheduling is looming!

- LHP for blocking locks
  - mutex, rwsem
- Readers preemption (RP) in read-write locks

Semantic gap between virtual and physical CPU

- Blocked-waiter wakeup (BWW)
  - Waking up a blocked thread on an idle vCPU is at least 10 times costlier
Our approach to address semantic gap

**Insight:**
A vCPU may be running a critical task!

**Approach:**
Avoid preempting a vCPU with a critical task

**Design:**
Identify and mark/unmark a critical task
Identifying each critical section with eCS

- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
  - 60 LoC annotates 85K lock invocations in 13M LoC in Linux
Identifying each critical section with eCS

- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
  - 60 LoC annotates 85K lock invocations in 13M LoC in Linux
Identifying each critical section with eCS

- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
  - 60 LoC annotates 85K lock invocations in 13M LoC in Linux
Sharing the state for efficient notification

- Each vCPU shares memory with the hypervisor
- vCPU updates information for critical sections
  - Notifies critical task to the hypervisor
- Hypervisor also updates scheduler context before/after scheduling out a vCPU
  - Enables vCPU to make efficient scheduling decisions
Lightweight para-virtualized APIs to update states

<table>
<thead>
<tr>
<th></th>
<th>Hint</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM → Hypervisor</td>
<td>activate_non_preemptable_ecs(cpu)</td>
<td>deactivate_non_preemptable_ecs(cpu_id)</td>
</tr>
<tr>
<td></td>
<td>deactivate_preemptable_ecs(cpu_id)</td>
<td>activate_preemptable_ecs(cpu_id)</td>
</tr>
<tr>
<td></td>
<td>deactivate_preemptable_ecs(cpu_id)</td>
<td></td>
</tr>
<tr>
<td>Hypervisor → VM</td>
<td>is_vcpu_preempted(cpu_id)</td>
<td>is_pcpu_overloaded(cpu_id)</td>
</tr>
</tbody>
</table>

- Red: Updated by each vCPU; read by the hypervisor
- Blue: Update by the hypervisor; read by a vCPU
Hypervisor checks eCS state before scheduling out a vCPU

1. Running vCPU 1
2. vCPU 1 acquires lock
3. vCPU 1 updates eCS count
4. Hypervisor checks states before vCPU 1 preemption
5. Hypervisor lets vCPU 1 runs for extra time
6. vCPU 1 finishes and updates eCS count
7. Hypervisor penalizes vCPU 1 later
Hypervisor checks eCS state before scheduling out a vCPU

1. Running vCPU 1
2. vCPU 1 acquires lock
3. vCPU 1 updates eCS count
4. Hypervisor checks states before vCPU 1 preemption
5. Hypervisor lets vCPU 1 runs for extra time
6. vCPU 1 finishes and updates eCS count
7. Hypervisor penalizes vCPU 1 later
The case for system eventual fairness

- Hypervisor accounts extra time and later penalizes the enlightened VM
  - Penalize the schedule of an enlightened VM
  - Extend the schedule of the very next VM
- Hypervisor *optimistically* extends time for an enlightened CS
  - Decision made just before scheduling out a vCPU
  - Extra time (schedule) to avoid preemption: 1 ms
Even vCPU can make efficient scheduling decisions

- Share the hypervisor context with each VM
  - Lock waiters can avoid bWW problem
- Virtualized scheduling-aware spinning
  - Lock waiter keeps spinning until the lock is not acquired if the pCPU is not overloaded
Implementation

- Rely on paravirtualized VM
- Extended scheduler’s preempt_notifier API to check eCS states
  - Rely on scheduler_tick() to avoid vCPU preemption
- Overall implementation is 1000 LoC
  - 60 LoC for annotating almost every lock-based critical section
Evaluation

- Does eCS improve VM's performance?
- Does hypervisor maintain system eventual fairness?

- Setup: 8-socket, 80-core NUMA machine
Impact of eCS in over-committed scenario

- Experiment: run two VMs running same application
- eCS improves application throughput by 1.2 -- 2.3X
- eCS avoids preemptions by 85.8--100% → an extra schedule tick is sufficient

Apache web server

```
<table>
<thead>
<tr>
<th>#vCPUs</th>
<th>VM</th>
<th>eCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>50k</td>
<td>80k</td>
</tr>
<tr>
<td>20</td>
<td>100k</td>
<td>160k</td>
</tr>
<tr>
<td>30</td>
<td>150k</td>
<td>240k</td>
</tr>
<tr>
<td>40</td>
<td>200k</td>
<td>320k</td>
</tr>
<tr>
<td>50</td>
<td>250k</td>
<td>400k</td>
</tr>
<tr>
<td>60</td>
<td>300k</td>
<td>500k</td>
</tr>
<tr>
<td>70</td>
<td>350k</td>
<td>600k</td>
</tr>
<tr>
<td>80</td>
<td>400k</td>
<td>700k</td>
</tr>
</tbody>
</table>
```

Psearchy

```
<table>
<thead>
<tr>
<th>#vCPUs</th>
<th>Jobs/hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td>150</td>
</tr>
<tr>
<td>40</td>
<td>200</td>
</tr>
<tr>
<td>50</td>
<td>250</td>
</tr>
<tr>
<td>60</td>
<td>300</td>
</tr>
<tr>
<td>70</td>
<td>350</td>
</tr>
<tr>
<td>80</td>
<td>400</td>
</tr>
</tbody>
</table>
```

Preemptions avoided

```
<table>
<thead>
<tr>
<th>#vCPUs</th>
<th>% no preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>75</td>
</tr>
<tr>
<td>20</td>
<td>70</td>
</tr>
<tr>
<td>30</td>
<td>65</td>
</tr>
<tr>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>70</td>
<td>45</td>
</tr>
<tr>
<td>80</td>
<td>40</td>
</tr>
</tbody>
</table>
```

Apache, Psearchy
Impact of eCS in under-committed scenario

- Experiment: Run only one VM with an application
- eCS improves application performance by 1.2 -- 1.9X
- Virtualized scheduling-aware spinning addresses BWW for blocking locks
System eventual fairness

- Experiment: an application reading a file
- Hypervisor’s scheduler (CFS) maintains eventual fairness
- Both VMs get equal time even though VM2 (eCS) is granted extra schedules
- CFS maintains eventual fairness by penalizing VM2
  - Each run for equal time (4.95 seconds out of 10 seconds)
Discussion

● Right approach for Linux adoption
  ○ Leverage steal_time_struct that exposes preempted method

● Annotation
  ○ Use VM → Hypervisor API to mark functions

● Extending the concept to the userspace
  ○ Require composable scheduling abstraction to support user space
Conclusion

- Double scheduling leads to several preemption problems
- Six lightweight paravirtualized methods to annotate critical sections
- Leverage hypervisor’s scheduler to mitigate vCPU preemptions
- Allow vCPU to make efficient scheduling decision
- A generic approach to mitigate all preemption problems!

Thank you!