Scaling Guest OS Critical Sections with *e*CS

Sanidhya Kashyap, Changwoo Min, Taesoo Kim

Georgia Tech

VIRGINIA TECH

The physical and virtual CPU abstraction

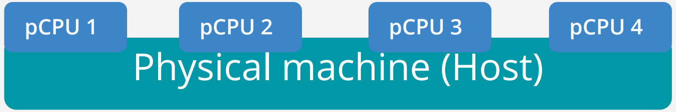
- Mismatch between

  CPU abstraction

The physical and virtual CPU abstraction

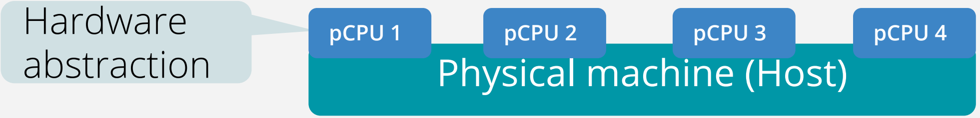
- Mismatch between

  CPU abstraction



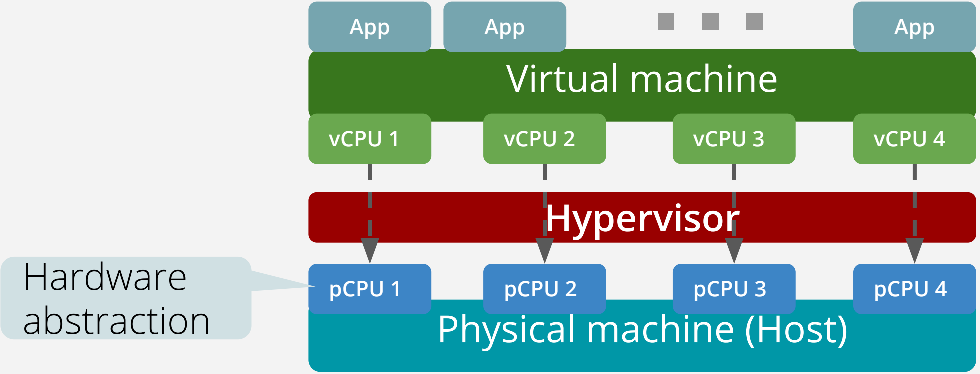
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction



The physical and virtual CPU abstraction

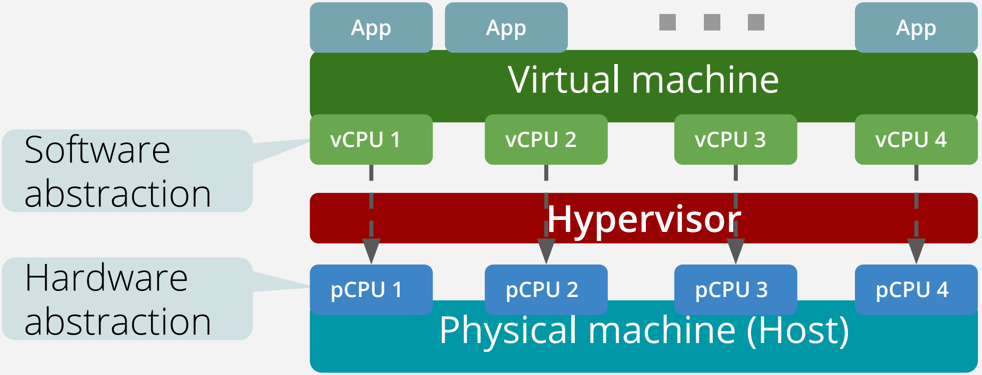
- Mismatch between CPU abstraction



Hardware abstraction

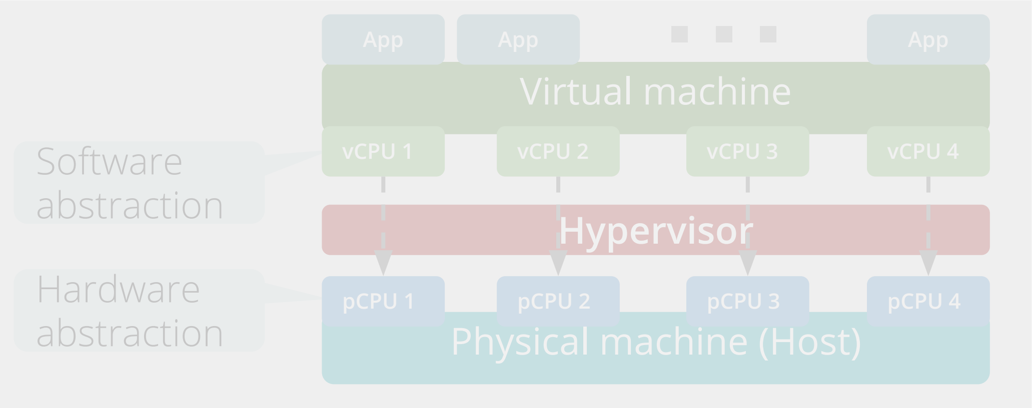
The physical and virtual CPU abstraction

- Mismatch between CPU abstraction

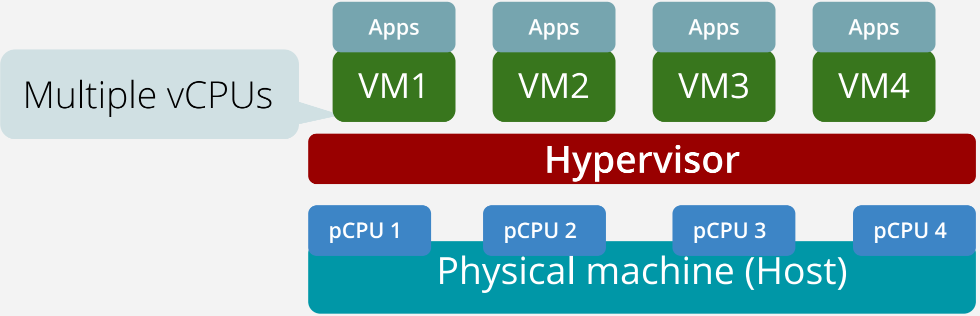


The physical and virtual CPU abstraction

- Mismatch between CPU abstraction



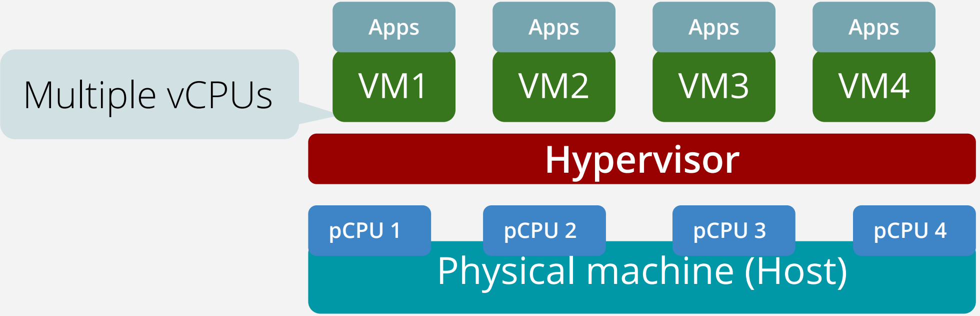
- VM consolidation
  - Contention on pCPU



The physical and virtual CPU abstraction

A vCPU can be preempted without notification

- VM consolidation
  - Contention on pCPU



Multiple vCPUs

Apps Apps Apps Apps

VM1 VM2 VM3 VM4

Hypervisor

pCPU 1 pCPU 2 pCPU 3 pCPU 4

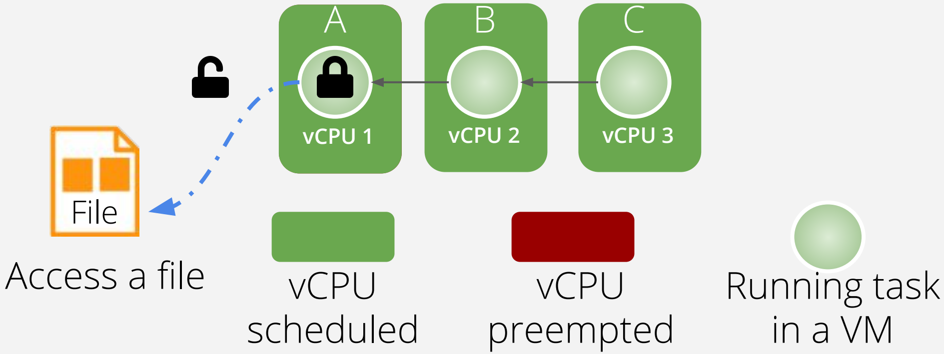
Physical machine (Host)

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!

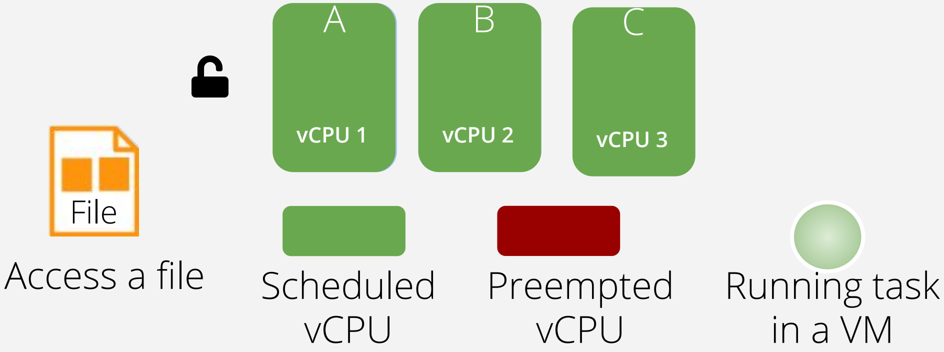
Approach:

Avoid preempting a vCPU with a critical task

Design:

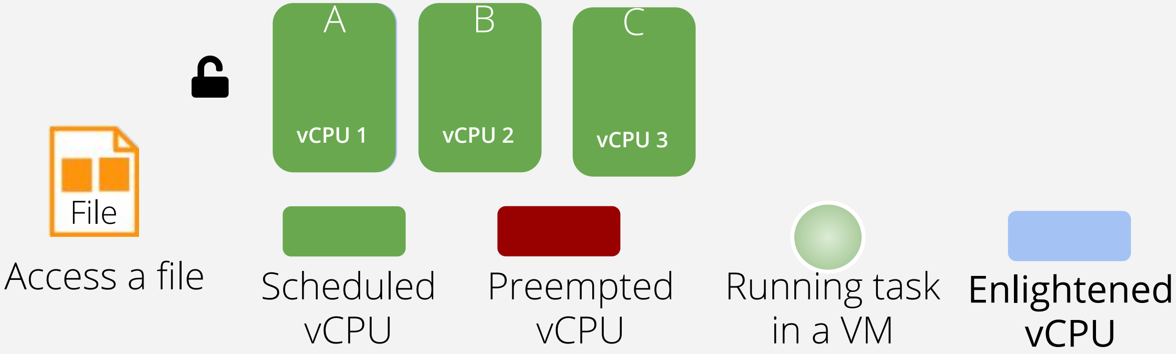
Identify and mark/unmark a critical task

Identifying each critical section with *e*CS



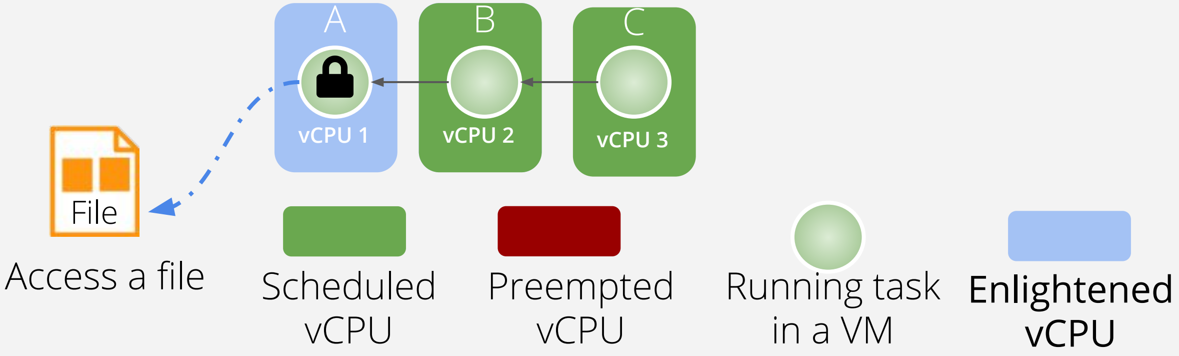
- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
    - 60 LoC annotates 85K lock invocations in 13M LoC in Linux

Identifying each critical section with *e*CS



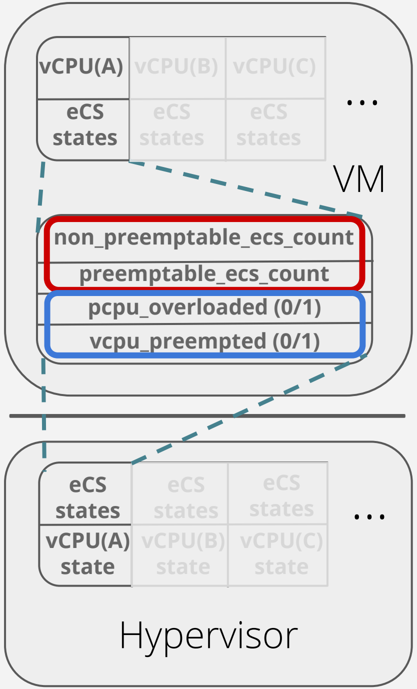
- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
    - 60 LoC annotates 85K lock invocations in 13M LoC in Linux

Identifying each critical section with *e*CS



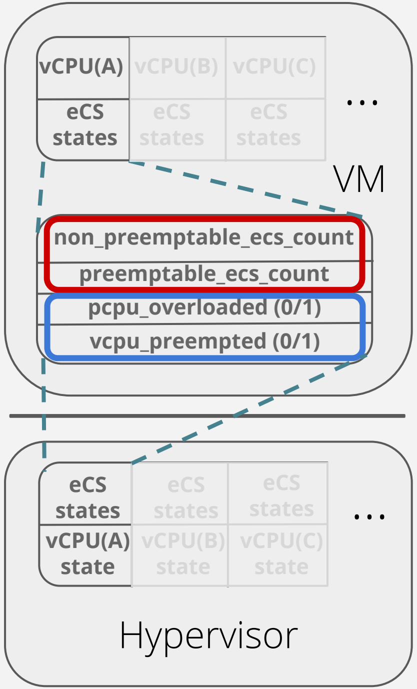
- Synchronization primitives protect critical sections → ensure OS progress
- Mark and unmark critical sections before and after the critical section
- Conservative, but effective approach to address each preemption problem
  - 60 LoC annotates 85K lock invocations in 13M LoC in Linux

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

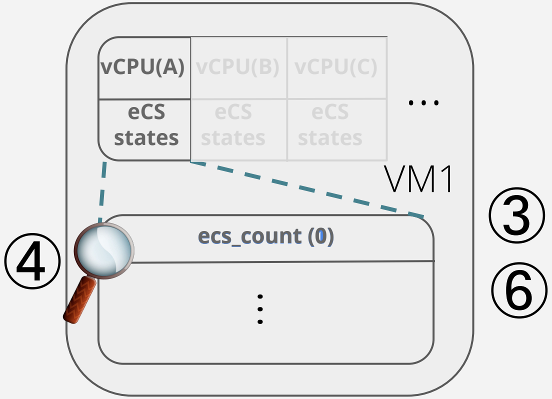


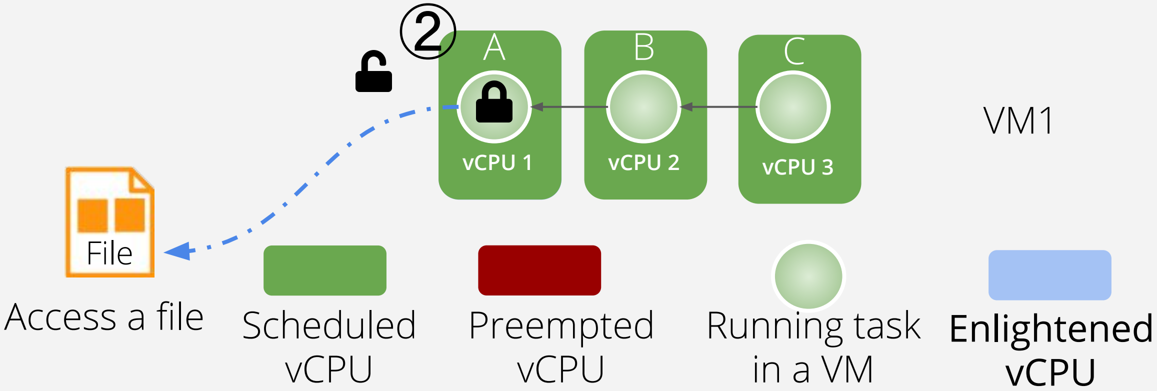
Hint	API
VM → Hypervisor	activate_non_preemptable_ecs(cpu)
	deactivate_non_preemptable_ecs(cpu_id)
	activate_preemptable_ecs(cpu_id))
	deactivate_preemptable_ecs(cpu_id)
Hypervisor → VM	is_vcpu_preempted(cpu_id)
	is_pcpu_overloaded(cpu_id)

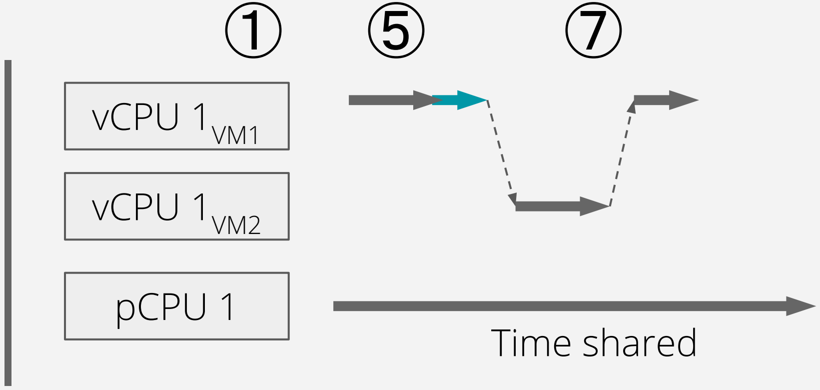
🟥 ⇨ Updated by each vCPU; read by the hypervisor

🟦 ⇨ Update by the hypervisor; read by a vCPU

Hypervisor checks eCS state before scheduling out a vCPU

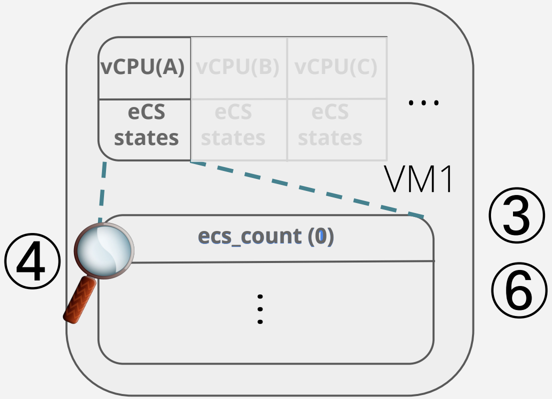


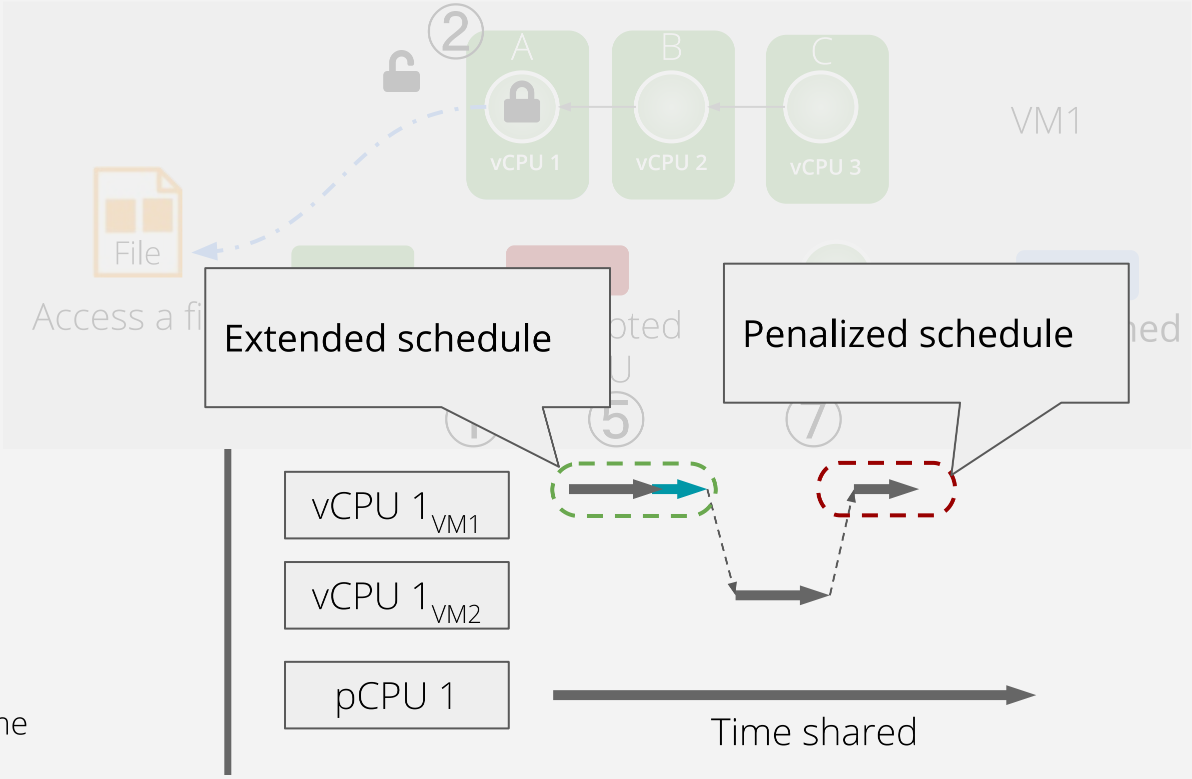




① Running vCPU 1
② vCPU 1 acquires lock
③ vCPU 1 updates eCS count
④ Hypervisor checks states before vCPU 1 preemption
⑤ Hypervisor lets vCPU 1 runs for extra time
⑥ vCPU 1 finishes and updates eCS count
⑦ Hypervisor penalizes vCPU 1 later

Hypervisor checks eCS state before scheduling out a vCPU



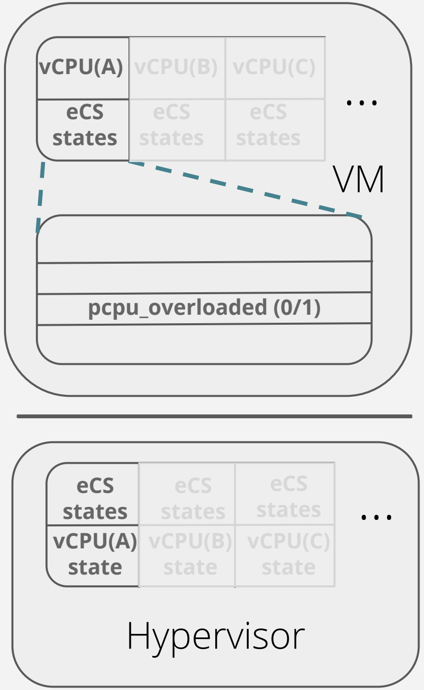


① Running vCPU 1
② vCPU 1 acquires lock
③ vCPU 1 updates eCS count
④ Hypervisor checks states before vCPU 1 preemption
⑤ Hypervisor lets vCPU 1 runs for extra time
⑥ vCPU 1 finishes and updates eCS count
⑦ Hypervisor penalizes vCPU 1 later

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

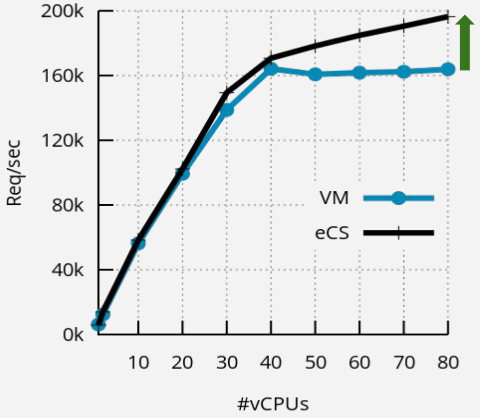
Evaluation

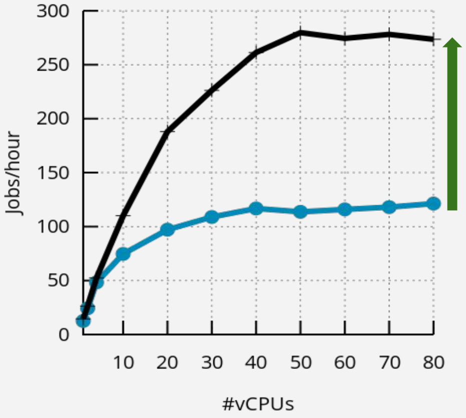
- Does eCS improves VM's performance?
- Does hypervisor maintain system eventual fairness?

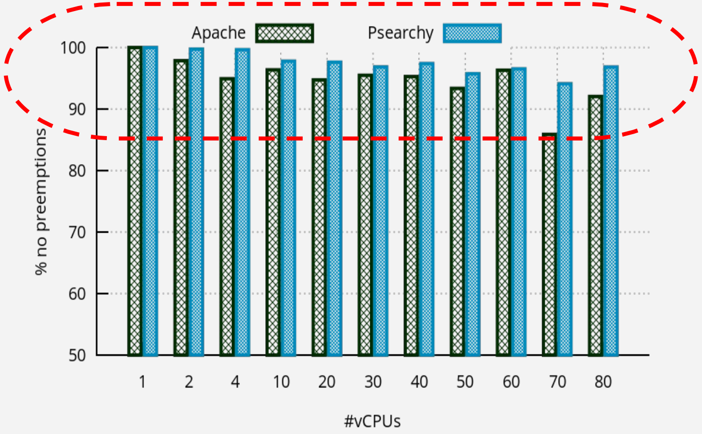
- Setup: 8-socket, 80-core NUMA machine

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

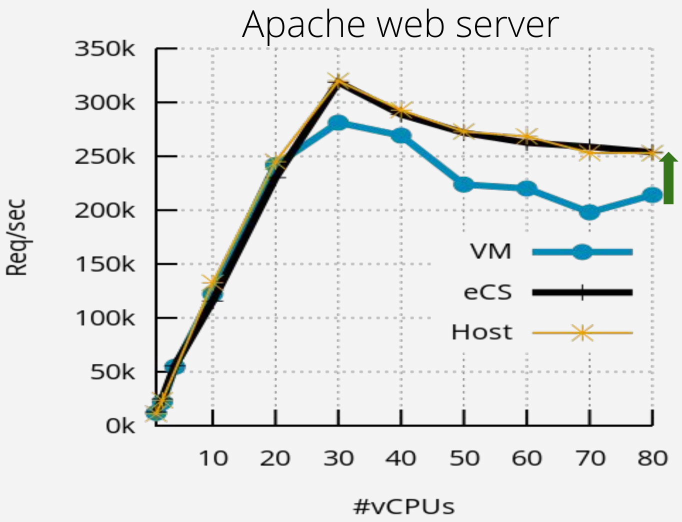

Apache web server

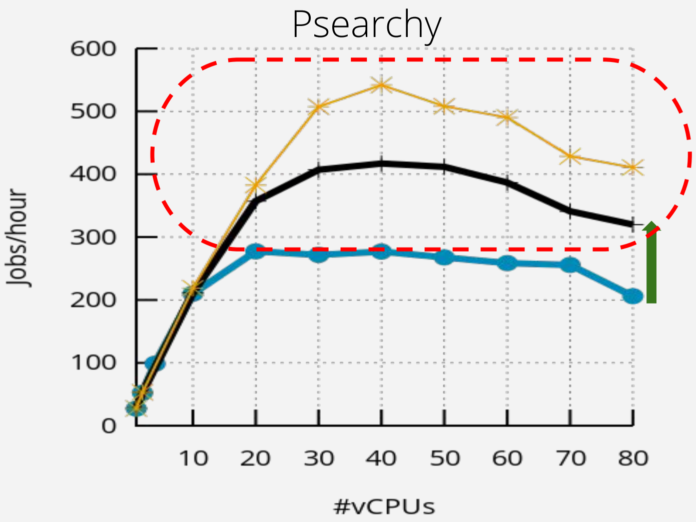

Psearchy


Preemptions avoided

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!