SMP VIRTUAL MACHINE

Haibo Chen

Slides from the ASPLOS talk by Kim et al.
Problems of Scheduling SMP VM

SMP VMs on a multicore server

Hypervisor schedules vCPUs on pCPUs
OS schedules processes on vCPUs
Double Scheduling Problems

Schedule dilemma

- Virtualization needs transparency
- Sematic gap between two schedulers
  - e.g., spinlock preemption

Scalability problem

- Virtualization enlarges the sync round-trip
  - e.g., mutex releasing
Schedule Dilemma

VCPU preemption

VCPU stacking
Coordinated Scheduling

Uncoordinated scheduling
→ A vCPU treated as an independent entity

Coordinated scheduling
→ Sibling vCPUs treated as a group
   (who belongs to the same VM)

Uncoordinated scheduling makes inter-vCPU synchronization ineffective
Prior Efforts for Coordination

**Coscheduling** [Ousterhout82]
- Synchronizing execution

- vCPU execution
- pCPU
- pCPU
- pCPU
- Time

**Relaxed coscheduling** [VMware10]
- Balancing execution time
- Stop execution for siblings to catch up

- pCPU
- pCPU
- pCPU
- pCPU
- Time

**Need for VMM scheduling based on synchronization (coordination) demands**

- Good CPU utilization & coordination, but not based on synchronization demands

- Better coordination through explicit information, but relying on user or OS support
Overview

- Demand-based coordinated scheduling

- Identifying synchronization demands
- With non-intrusive design
- Not compromising inter-VM fairness
Coordination Space

- Time and space domains
- Independent scheduling decision for each domain

**Time**
When to schedule?

**Space**
Where to schedule?

**Preemptive scheduling policy**
- Coscheduling
- Delayed preemption

**pCPU assignment policy**
Outline

• Motivation

• **Coordination in time domain**
  • Kernel-level coordination demands
  • User-level coordination demands

• Coordination in space domain
  • Load-conscious balance scheduling

• Evaluation
Synchronization to be Coordinated

- Synchronization based on “busy-waiting”
  - Unnecessary CPU consumption by busy-waiting for a descheduled vCPU
    - Significant performance degradation
  - Semantic gap
    - “OSes make liberal use of busy-waiting (e.g., spinlock) since they believe their vCPUs are dedicated”
      → Serious problem in kernel

- When and where to demand synchronization?
- How to identify coordination demands?
Kernel-Level Coordination Demands

- Does kernel really need coordination?
  - Experimental analysis
    - Multithreaded applications in the PARSEC suite
    - Measuring “kernel time” when uncoordinated

Solorun (no consolidation)  A 8-vCPU VM  Corun (w/ 1 VM running streamcluster)

Kernel time ratio is largely amplified by x1.3-x30  → “Newly introduced kernel-level contention”
Kernel-Level Coordination Demands

• Where is the kernel time amplified?

Dominant sources
1) TLB shutdown
2) Lock spinning

How to identify?
How to Identify TLB Shootdown?

- TLB shootdown
  - Notification of TLB invalidation to a remote CPU

“Busy-waiting for TLB synchronization” is efficient in native systems, but not in virtualized systems if target vCPUs are not scheduled. (Even worse if TLBs are synchronized in a broadcast manner)
How to Identify TLB Shootdown?

- TLB shootdown IPI
  - Virtualized by VMM
  - Used in x86-based Windows and Linux

“A TLB shootdown IPI is a signal for coordination demand!”
→ Co-schedule IPI-recipient vCPUs with its sender vCPU
How to Identify Lock Spinning?

- Why excessive lock spinning?
  - “Lock-holder preemption (LHP)”
    - Short critical section can be unpredictably prolonged by vCPU preemption

- Which spinlock is problematic?

Spinlock wait time breakdown

<table>
<thead>
<tr>
<th>Lock</th>
<th>Wait Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bodytrack</td>
<td>100%</td>
</tr>
<tr>
<td>canmeal</td>
<td>82%</td>
</tr>
<tr>
<td>dedup</td>
<td>93%</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>82%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>93%</td>
</tr>
<tr>
<td>swaptions</td>
<td>82%</td>
</tr>
<tr>
<td>vips</td>
<td>93%</td>
</tr>
<tr>
<td>x264</td>
<td>82%</td>
</tr>
</tbody>
</table>

- Other locks
  - Runqueue lock 93%
  - Pagetable lock 82%
  - Semaphore wait-queue lock
  - Futex wait-queue lock
How to Identify Lock Spinning?

- **Futex**
  - Linux kernel support for user-level synchronization (e.g., mutex, barrier, conditional variables, etc)

```c
mutex_lock(mutex)
/* critical section */
mutex_unlock(mutex)
```

```c
futex_wake(mutex) {
  spin_lock(queue->lock)
  thread=dequeue(queue)
  wake_up(thread)
  spin_unlock(queue->lock)
}
```

```c
mutex_lock(mutex)
```

```c
futex_wait(mutex) {
  spin_lock(queue->lock)
  enqueue(queue, me)
  spin_unlock(queue->lock)
  schedule() /* blocked */
  /* wake-up */
  /* critical section */
  mutex_unlock(mutex)
  futex_wake(mutex) {
    spin_lock(queue->lock)
    /* critical section */
    mutex_unlock(mutex)
```
How to Identify Lock Spinning?

- Why preemption-prone?

  - Prolonged by VMM intervention
  - Multiple VMM interventions for one IPI transmission
  - Repeated by iterative wake-up

  No more short critical section! → Likelihood of preemption

  - Preemption by woken-up sibling
    → Serious issue

Remote thread wake-up → Wait-queue unlock → Wait-queue lock spinning

<table>
<thead>
<tr>
<th>vCPU0</th>
<th>vCPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to Identify Lock Spinning?

- Generalization: “Wait-queue locks”
  - Not limited to futex wake-up
  - Many wake-up functions in the Linux kernel
    - General wake-up
      - \_wake_up()\n    - Semaphore or mutex unlock
      - \texttt{rwsem\_wake()}, \texttt{\_mutex\_unlock\_common\_slowpath()}, …
  - “Multithreaded workloads usually communicate and synchronize on wait-queues”

“A Reschedule IPI is a signal for coordination demand!”

\(\rightarrow\) Delay preemption of an IPI-sender vCPU until a likely-held spinlock is released
User-Level Coordination Demands

- Coscheduling-friendly workloads
  - SPMD, bulk-synchronous, etc.
  - Busy-waiting synchronization
    - “Spin-then-block”

Coscheduling (balanced execution)

Uncoordinated (skewed execution)

Uncoordinated (largely skewed execution)

More blocking operations when uncoordinated
User-Level Coordination Demands

- Coscheduling
  - Avoiding more expensive blocking in a VM
    - VMExits for CPU yielding and wake-up
      - Halt (HLT) and Reschedule IPI
  - When to coschedule?
    - User-level synchronization involves reschedule IPIs

“A Reschedule IPI is a signal for coordination demand!”
→ Co-schedule IPI-recipient vCPUs with a sender vCPU

Providing a knob to selectively enable this coscheduling for coscheduling-friendly VMs
Urgent vCPU First (UVF) Scheduling

- Urgent vCPU
  - 1. Preemptively scheduled if fairness is kept
  - 2. Protected from preemption once scheduled
    - During “Urgent time slice (utslice)”
Outline

• Motivation
• Coordination in time domain
  • Kernel-level coordination demands
  • User-level coordination demands

• Coordination in space domain
  • Load-conscious balance scheduling

• Evaluation
vCPU-to-pCPU Assignment

- **Balance scheduling** [Sukwong11]
  - Spreading sibling vCPUs on different pCPUs
    - Increase in likelihood of coscheduling
    - No coordination in time domain

Uncoordinated scheduling

Balance scheduling

vCPU stacking

Likelihood of coscheduling

No vCPU stacking

\[
\text{Uncoordinated scheduling} \quad < \quad \text{Balance scheduling}
\]
vCPU-to-pCPU Assignment

- **Balance scheduling** [Sukwong11]
- **Limitation**
  - Based on “global CPU loads are well balanced”
  - In practice, VMs with fair CPU shares can have

![Diagram showing different # of vCPUs and different TLP](image)

**TLP can be changed in a multithreaded app**

TLP: Thread-level parallelism
Proposed Scheme

- Load-conscious balance scheduling
  - Adaptive scheme based on pCPU loads
  - When assigning a vCPU, check pCPU loads

If load is balanced
→ Balance scheduling

If load is imbalanced
→ Favoring underloaded pCPUs

Handled by coordination in time domain

CPU load > Avg. CPU load
→ overloaded
Outline

• Motivation

• Coordination in time domain
  • Kernel-level coordination demands
  • User-level coordination demands

• Coordination in space domain
  • Load-conscious balance scheduling

• Evaluation
Evaluation

- Implementation
  - Based on Linux KVM and CFS

- Evaluation
  - Effective time slice
    - For coscheduling & delayed preemption
    - 500us decided by sensitive analysis
  - Performance improvement
  - Alternative
    - OS re-engineering
Evaluation

- SMP VM with UP VMs
  - One 8-vCPU VM + four 1-vCPU VMs (x264)

### Performance of 8-vCPU VM

- **Baseline**: Basic performance without any optimizations.
- **Balance**: Load-conscious balance scheduling.
- **LC-Balance**: Load-conscious balance scheduling with additional optimizations.
- **LC-Balance+Resched-DP**: LC-Balance with delayed preemption for reschedule IPI.
- **LC-Balance+Resched-DP+TLB-Co**: LC-Balance+Resched-DP with coscheduling for TLB shootdown IPI.

#### Workloads of 8-vCPU VM

- **blackscholes**: Non-synchronization-intensive
- **freqmine**: Non-synchronization-intensive
- **raytrace**: Non-synchronization-intensive
- **swaptions**: Non-synchronization-intensive
- **streamcluster**: Non-synchronization-intensive
- **facesim**: Futex-intensive → 5-53% improvement
- **bodytrack**: Futex-intensive → 5-53% improvement
- **fluidanimate**: Futex-intensive → 5-53% improvement
- **X264**: Futex-intensive → 5-53% improvement
- **carnes**: Futex-intensive → 5-53% improvement
- **dedup**: Futex-intensive → 5-53% improvement
- **ferret**: Futex-intensive → 5-53% improvement
- **vips**: Futex-intensive → 5-53% improvement

#### TLB-intensive Workloads

- **TLB-intensive**: Coscheduling for TLB shootdown IPI → 20-90% improvement
Alternative: OS Re-engineering

- Virtualization-friendly re-engineering
  - Decoupling reschedule IPI transmission from thread wake-up

Delayed reschedule IPI transmission

```c
wake_up (queue) {
    spin_lock(queue->lock)
    thread=dequeue(queue)
    wake_up(thread)
    spin_unlock(queue->lock)
}
```

- Modified `wake_up` func
- Using per-cpu bitmap
- Applied to `futex_wakeup` & `futex_requeue`

Delayed reschedule IPI is virtualization-friendly to resolve LHP problems
Conclusions & Future Work

- Demand-based coordinated scheduling
  - IPI as an effective signal for coordination
  - pCPU assignment conscious of dynamic CPU loads

- Limitation
  - Cannot cover **ALL** types of synchronization demands
    - Kernel spinlock contention w/o VMM intervention

- Future work
  - Cooperation with HW (e.g., PLE) & paravirt
Urgent vCPU First (UVF) Scheduling

- Urgent vCPU
  - 1. Preemptively scheduled if fairness is kept
  - 2. Protected from preemption once scheduled
    - During "Urgent time slice (utslice)"
Proposed Scheme

- Load-conscious balance scheduling
  - Adaptive scheme based on pCPU loads

Balanced loads → Balance scheduling

Imbalanced loads → Favoring underloaded pCPUs

• Example

Candidate pCPU set
(Scheduler assigns a lowest-loaded pCPU in this set)
= \{pCPU0, pCPU1, pCPU2, pCPU3\}

pCPU3 is overloaded
(i.e., CPU load > Avg. CPU load)

Handled by coordination in time domain
(UVF scheduling)
Evaluation

- Urgent time slice (utslice)
  - 1. Utslice for reducing LHP
  - 2. Utslice for quickly serving multiple urgent vCPUs

**Workloads:**
A futex-intensive workload in one VM
+ *dedup* in another VM as a preempting VM

>300us utslice
2x-3.8x LHP reduction

Remaining LHPs occur during local wake-up or before reschedule IPI transmission
→ Unlikely lead to lock contention
Evaluation

- Urgent time slice (utslice)
  - 1. utslice for reducing LHP
  - 2. utslice for quickly serving multiple urgent vCPUs

Workloads:
3 VMs, each of which runs vips
(vips - TLB-IPI-intensive application)

As utslice increases, TLB shootdown cycles increase

500usec is an appropriate utslice for both LHP reduction and multiple urgent vCPUs
Evaluation

- Urgent allowance
  - Improving overall efficiency with fairness

**Workloads:**
vips (TLB-IPI-intensive) VM + two *facesim* VMs

No performance drop

Efficient TLB synchronization
Evaluation

- Impact of kernel-level coordination
  - One 8-vCPU VM + four 1-vCPU VMs (x264)

![Diagram showing unfair contention and performance comparison]

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Balance</th>
<th>LC-Balance</th>
<th>LC-Balance+Resched-DP</th>
<th>LC-Balance+Resched-DP+TLB-Co</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Workload</th>
<th>Baseline</th>
<th>Balance</th>
<th>LC-Balance</th>
<th>LC-Balance+Resched-DP</th>
<th>LC-Balance+Resched-DP+TLB-Co</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ blackscholes</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ bodytrack</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ canneal</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ dedup</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ facesim</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ ferret</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ fluidanimate</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ fremine</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ raytrace</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ streamcluster</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ swaptions</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ vips</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>w/ x264</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Performance of 1-vCPU VM

- LC-Balance: Load-conscious balance scheduling
- Resched-DP: Delayed preemption for reschedule IPI
- TLB-Co: Coscheduling for TLB shootdown IPI

**Balance scheduling → Up to 26% degradation**

Co-running workloads with 1-vCPU VM (x264)
Evaluation: Two SMP VMs

\[ \sum \frac{Time_{solorun}}{Time_{corun}} \]

- a: baseline
- b: balance
- c: LC-balance
- d: LC-balance+Resched-DP
- e: LC-balance+Resched-DP+TLB-Co

\[ \text{Corunner (dedup)} \]
\[ \text{Main workload (at X-axis)} \]

\[ \text{Corunner (freqmine)} \]
\[ \text{Main workload (at X-axis)} \]
Evaluation

- Effectiveness on HW-assisted feature
  - CPU feature to reduce the amount of busy-waiting
    - VMExit in response to excessive busy-waiting
    - Intel Pause-Loop-Exiting (PLE), AMD Pause Filter
    - Inevitable cost of some busy-waiting and VMExit

<table>
<thead>
<tr>
<th>Apps</th>
<th>Streamcluster (futex-intensive)</th>
<th>facesim</th>
<th>ferret</th>
<th>vips</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction in Pause-loop VMExits (%)</td>
<td>44.5</td>
<td>97.7</td>
<td>74.0</td>
<td>37.9</td>
</tr>
</tbody>
</table>

![Graphs showing performance metrics for Streamcluster and ferret applications.](image-url)
Evaluation

- Coscheduling-friendly user-level workload
  - Streamcluster
    - Spin-then-block barrier intensive workload

Normalized execution time (corunning w/ bodytrack)

**More performance improvement as the time of spin-waiting increases**

Resched-Co: Coscheduling for reschedule IPI

**Barrier breakdown**

- Blocking: 38%
- Reschedule IPIs (3 VMExits): 21%
- Additional (departure) barriers: 29%