Towards Fair and Efficient SMP Virtual Machine Scheduling

Jia Rao and Xiaobo Zhou

University of Colorado, Colorado Springs

http://cs.uccs.edu/~jrao/
Executive Summary

• **Problem:** *unfairness* and *inefficiency* in consolidating SMP VMs
  
  ‣ Existing VM schedulers favor VMs w/ more virtual CPUs
  
  ‣ Fairness mechanisms hurt parallel performance

• **Flex** is a scheduling framework that:
  
  ‣ Adaptively adjusts vCPU weights for VM-level fairness
  
  ‣ Flexibly schedules vCPUs to minimize unnecessary spinning
  
  ‣ **Results:** 5% error to the ideally fair allocation, 30%+ performance improvement for parallel workloads, ~1% overhead in Xen
SMP VM Consolidation

• Abundant hardware parallelism in DC

• SMP VMs are prevalent in the cloud
  ‣ 26 out of 29 instance types in Amazon EC2 have more than two vCPUs
  ‣ Support parallel applications
  ‣ Heterogeneous consolidation is common, e.g., Amazon EC2
Unfair VM CPU Allocation

- CPU allocation
  - **NOT** proportional to VM weights
  - VMs w/ more vCPUs gain advantage
  - **Common** issue in all hypervisors

![Diagram showing CPU allocation and normalized CPU consumption for Xen, KVM, and VMware]
Causes of Unfairness

- Per-CPU scheduling
  - Independent scheduler on each CPU
  - Each allocates CPU based on relative vCPU weights
  - Per-vCPU weight dependent on VM weight and the number of vCPUs
  - Scalable but hard for VM-level fairness

Equal weight VMs
Box size reflects per-vCPU weight

Allocations on vCPU $\Rightarrow$ VM-level fairness
IFF
Same total weight on each CPU
Existing Solutions

• Capping VM CPU consumption (Cap)
  ‣ Requires pre-calculation of the fair share
  ‣ Non-work-conserving
  ‣ Introduce significant inefficiencies to parallel applications

• Load balancing (LB)
  ‣ Tries to achieve equal weights on CPUs
  ‣ Balance weight vs. balance run queue length
Cap on Busy-waiting-based Workloads

- Busy-waiting synchronization
  - Tasks stay in a busy loop waiting for lock release
  - Avoids contexts switches
- Lock holder preemption (LHP)
  - Preemption of vCPU holding locks
  - Long synchronization latency

Capping exacerbates the LHP issue in virtualized environments
LB on Blocking-based Workloads

- Blocking synchronization
  - Tasks go to sleep if failing to acquire the lock
  - Avoids wasted CPU cycles
- vCPU stacking
  - vCPUs belonging to one VM pile on the same CPU
  - No parallelism + Long sync latency

LB exacerbates the vCPU stacking issue

Since blocking vCPUs frequently switch between READY and RUNNING states, they are more likely “victims” of work-stealing based load balancing. Gradually, stolen vCPUs pile on a few CPUs.
Related Work

- Fairness in multicore systems
  - [Li-PPoPP09] - no VM-level fairness

- Minimizing sync latency in SMP VMs
  - [Sukwong-Eurosys11] - avoids vCPU stacking
  - [Kim-ASPLOS13], [Weng-HPDC11] - not effective in user-level synchronization
  - Pause loop exit (PLE) - needs hardware support

- Spin detection
  - [Wells-PACT06] - store-based spin detection, not accurate to apps with different store rates, e.g., LU in NAS parallel benchmark
Flex for Fairness and Efficiency

• Flexible vCPU weight (FlexW)
  ‣ Monitors VM CPU consumption
  ‣ Calculates fair shares based on VM weights
  ‣ Adjusts vCPU weights to compensate the difference

• Flexible vCPU scheduling (FlexS)
  ‣ Stops spinning vCPUs to avoid wasted CPU cycles
  ‣ Switches the preempted vCPU with one on another CPU that is doing useful work
  ‣ Ensures that no vCPUs from the same VM stack on one CPU
FlexW Design

- Determine the fair share
  - $P$ - number of shared CPUs, $w_i$ - VM weight
  - Ideally fair share according to generalized processor sharing (GPS)
    \[ S_{i, GPS}(t_1, t_2) = \frac{w_i}{\sum w_j} (t_2 - t_1) \cdot P \]
- Adjust VM weights
  - $w_i^r$ - VM weight
  - calculate the lag \[ lag_i(t_1, t_2) = \frac{S_{i, GPS}(t_1, t_2) - S_i(t_1, t_2)}{S_{i, GPS}(t_1, t_2)} \]
  - compensate the lag with real-time weights
    \[ w_i^r = w_i^r + w_i \cdot lag_i(t_1, t_2) \]
FlexS Design

• Identifying busy-waiting vCPU

  ‣ Non-intrusive identification without application knowledge

  ‣ Common pattern in different spin implementations
    - Spin loops contain a few instructions
    - Spin loops are executed many times

  ‣ Spin loops show **high** branch per instruction (BPI) and **low** branch miss prediction rate (BMPR)
FlexS Design (cont’)

• Eliminating busy-waiting time
  ‣ Periodically update a vCPU’s BPI and BMPR
  ‣ Busy-waiting vCPU voluntarily yields CPU
  ‣ Find a sibling vCPU to complete the unfinished time slice
  ‣ **Switch** the two vCPU to avoid vCPU stacking and run queue weight changes
Practical Considerations

• Starvation
  • VMs demanding less than its share will have ever increasing real-time weight
    • **Solution:** reset real-time weight every 10s

• Infeasible weight -> peak CPU demand less than the fair share
  • **Solution:** peak demand as the fair share

• False positive in identifying spinning vCPU
  • **Solution:** reset BPI and BMPR every 10s

• Inter-CPU locking overhead due to vCPU migrations
  • **Solution:** only try twice when looking for siblings to switch- the *power of two choices*
Implementation

• Implement Flex in Xen’s credit scheduler
  ‣ weight -> credit
  ‣ **FlexW** in the system-wide csched_acct() routine, adjusts VM credit refill based on real-time weights, invoked every 30ms
  ‣ **FlexS** in the per-CPU schedule() function, adds load_balance_switch() to exchange work with sibling vCPUs
  ‣ Identify spinning vCPU in vcpu_acct() when Xen charges credit to the current running vCPU
Evaluation Methodology

• **Questions**: *VM-level fairness?* and *parallel performance?*

• **Workload**
  - **NAS** Parallel benchmark (OpenMP, busy-waiting sync)
  - **PARSEC** (Pthreads, blocking sync)
  - **Background** interfering loops — isolate from cache contention

• **Scheduling strategies for comparison**
  - **Xen** default credit scheduler
  - **Balance+cap+CO** - [Sukwong-Eurosys11]
  - **Demand+cap** - [Kim-ASPLOS13]
VM-level Fairness

**Heterogeneous VMs:**
1vCPU, 2vCPU, 3vCPU, 4vCPU

Each running while(1) loop

Relative lag = $\left| \frac{S_{i,\text{GPS}}(t_1,t_2) - S_i(t_1,t_2)}{S_{i,\text{GPS}}(t_1,t_2)} \right|$

**Lower is better**

Flex: significant improvement over Xen with no more than 5% unfairness
VM Differentiation

Flex realizes proportional share among VMs
Parallel Performance

Challenge: Flex allocates less CPU time to the 4vCPU VM than Xen

Observation: FlexW alone does NOT guarantee good perf.

Reason: Imbalance wastes CPU time

Results: FlexW+FlexS performs closely to Xen and balance+cap+CO
Parallel Performance

Expected: Flex performs better than Xen

Reason: Flex allocates more CPU time to the 3vCPU VM than Xen
Mix of Parallel Workloads

Results: Flex outperforms balance+cap by 30.4%

Normalized runtime

Dynamic task assignment

Foreground NAS

Background lu

Lower is better
Overhead

System-wide `csched_acct()`

- FlexW overhead: VM weight adjustment
  - Overhead increases with # of VMs performed by the *idle* VM
  - not affecting parallel performance

Per-CPU func. `schedule()`

- FlexS overhead: vCPU stealing
  - constant overhead, not increases with # of VMs
  - frequency of `schedule()` - 30ms
  - less than 1% overhead
Conclusions & Future Work

• Fairness-efficiency tradeoff
  ‥ Straightforward solutions to unfairness lead to poor efficiency

• **Flex**: a holistic solution
  ‥ Adaptively adjusts weight for fairness
  ‥ Flexibly schedule vCPUs to minimize wasted work
  ‥ **Problem**: NOT quite effective for apps with dynamic task assignment

• Future work
  ‥ **Cross-layer** application-cloud coordination
Thank you!

Questions?

http://cs.uccs.edu/~jrao/
Backup Slides Begin Here …
**Observation**: Both Flex and demand+cap improve perf.

**Reason**: Avoiding vCPU stacking helps a lot

**Conclusion**: Flex does not incur much penalty to blocking sync.-based apps