Towards Fair and Efficient SMP Virtual Machine Scheduling

Jia Rao Xiaobo Zhou

Department of Computer Science
University of Colorado at Colorado Springs
{jrao, xzhou}@uccs.edu

Abstract

As multicore processors become prevalent in modern computer systems, there is a growing need for increasing hardware utilization and exploiting the parallelism of such platforms. With virtualization technology, hardware utilization is improved by encapsulating independent workloads into virtual machines (VMs) and consolidating them onto the same machine. SMP virtual machines have been widely adopted to exploit parallelism. For virtualized systems, such as a public cloud, fairness between tenants and the efficiency of running their applications are keys to success. However, we find that existing virtualization platforms fail to enforce fairness between VMs with different number of virtual CPUs (vCPU) that run on multiple CPUs. We attribute the unfairness to the use of per-CPU schedulers and the load imbalance on these CPUs that incur inaccurate CPU allocations. Unfortunately, existing approaches to reduce unfairness, e.g., dynamic load balancing and CPU capping, introduce significant inefficiencies to parallel workloads.

In this paper, we present *Flex*, a vCPU scheduling scheme that enforces fairness at VM-level and improves the efficiency of hosted parallel applications. Flex centers on two key designs: (1) dynamically adjusting vCPU weights (*FlexW*) on multiple CPUs to achieve VM-level fairness and (2) flexibly scheduling vCPUs (*FlexS*) to minimize wasted busy-waiting time. We have implemented Flex in Xen and performed comprehensive evaluations with various parallel workloads. Results show that Flex is able to achieve CPU allocations with on average no more than 5% error compared to the ideal fair allocation. Further, Flex outperforms Xen's credit scheduler and two representative co-scheduling approaches by as much as 10X for parallel applications using busy-waiting or blocking synchronization methods.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555246

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling, Synchronization; D.4.8 [*Operating Systems*]: Performance—Measurements

Keywords Virtual Machine Scheduling; Multicore Systems; Parallel Program Optimization

1. Introduction

Cloud computing, unlocked by virtualization technologies, is bringing a transformative change in enterprise architectures. By consolidating multiple independent workloads, each in a virtual machine (VM), onto a fewer number of machines, enterprises benefit from improved hardware utilizations and significant energy savings. On the other hand, virtualization provides important flexibilities to end users. Virtual servers can be on-the-fly reconfigured to meet the growth of hosted applications. Public infrastructure-as-a-service (IaaS) clouds such as Amazon EC2 [1] and Microsoft Azure [35] allow users to lease virtual servers from cloud providers with a pay-as-you-go charging model. However, cloud providers are facing a dilemma – consolidating more VMs on their hardware infrastructure to generate more revenue and achieving good performance for hosted applications to avoid the loss of customers. Since it is a common practice in today's cloud providers to multiplex VMs onto hardware resources, e.g., CPU cores, the question left to providers is how to provide guaranteed performance to users.

Performance guarantee has twofold meanings in a public cloud service: *fairness* and *efficiency*. First, performance should be predictable and proportional to user payments. At least, users belonging to the same service category with the same pay rate should receive a fair amount of shared resources. Second, the overhead introduced by virtualization should be minimized to approximate the execution efficiency in dedicated systems. There is existing work addressing the unfairness issue due to shared CPU caches [12, 17, 22], network I/O interface [3, 19] and shared storage [23]. Other work attempted to improve the efficiency for virtualized I/O processing [37, 38]. With the prevalence of parallel programming, symmetric multiprocessing (SMP) VMs with multiple virtual CPUs (vCPUs) are widely adopted in cloud ser-

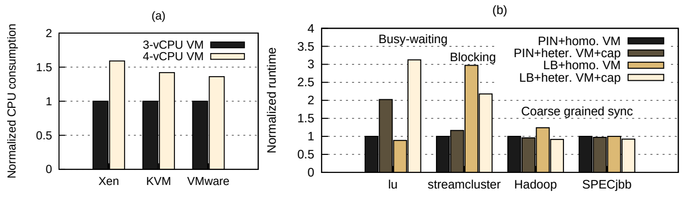

Figure 1. (a) VM-level unfairness for two heterogeneous VMs sharing 4 pCPUs. (b) Effect of fairness mechanisms, load-balancing (LB) and CPU capping (cap), on parallel performance.

vices. Literature has identified the synchronization overhead in SMP VMs as the major performance bottleneck for parallel programs in virtualized environments. Lock-holder preemption (LHP) [32] and vCPU stacking [27] are two unique issues arise with SMP scheduling. To mitigate these issues, researchers have proposed relaxed scheduling [29], balance scheduling [27], demand-based coordinated scheduling [11], and scheduling approaches based on spinlock-detection [7, 9, 32, 34].

In this work, we discover a hidden issue in consolidating SMP VMs – the scalable design of having independent schedulers on individual CPUs makes it difficult to achieve fair CPU allocation for SMP VMs. Existing mechanisms to enforce fairness, e.g., dynamic load balancing and CPU capping, however exacerbate SMP synchronization overhead. Figure 1(a) shows the actual CPU allocations for two heterogeneous VMs sharing four CPUs. SMP VMs can become heterogeneous in the number of vCPUs if some vCPUs are idle and go offline. The workloads running in the VMs are simply infinite loops that fully utilize a VM's processor resources. VMs were assigned the same weights and vCPUs were grouped together (e.g., using Linux `cgroup` in KVM) sharing the per-VM share. As shown in Figure 1(a), popular virtualization platforms, such as Xen [36], KVM [6] and VMware [33], are unable to achieve fair allocation at the VM level. The VM with a larger number of vCPUs gain advantages in getting more CPU resources. We attribute the unfairness to the use of per-CPU schedulers and the load imbalance on individual CPUs that leads to inaccurate CPU allocations. Load-balancing (`LB`) and CPU capping (`cap`) are popular approaches to enforce fairness, but introduce significant synchronization overhead to parallel workloads with different synchronization methods.

Figure 1(b) shows the performance of parallel workloads under different fairness mechanisms in Xen. Parallel workloads ran in the 4-vCPU VM and was co-located with a homogeneous (`homo.`) 4-vCPU VM or a heterogeneous (`heter.`) 3-vCPU VM. Details of the workloads are explained in Section 5. The baseline is when two homogeneous (`homo.`) VMs are co-located and their vCPUs are pinned (`PIN`) to CPUs. From the figure, we can see that when CPU capping is in place, heterogeneity in VM configuration incurs significant slowdowns (as much as 3X) to workloads with busy-waiting synchronization (e.g., spinlocks). When `LB` is used for fairness, it penalizes workloads with non-busy-waiting synchronization (e.g., mutexes). In addition, workloads with coarse-grained synchronization are resilient to these fairness mechanisms.

In this work, we propose a unified approach, *Flex*, for enforcing VM-level fairness and improving the execution efficiency of parallel programs in virtualized environments. Flex relies on two independent components: (1) *FlexW* achieves VM-level fairness by dynamically adjusting VM weights to account for the allocation inaccuracies caused by load imbalance on individual CPUs; (2) *FlexS* improves parallel execution efficiency by eliminating busy-waiting time and accelerating sequential portions of parallel programs. FlexS employs a novel vCPU migration algorithm to prioritize vCPUs doing useful work without compromising VM-level fairness. To identify busy-waiting vCPUs, we also devise a simple but effective hardware metric-based approach. We find that branches per instruction (BPI) and branch miss prediction rate (BMPR) together accurately pinpoint the spin loops in different implementations of busy-waiting-based synchronization.

We have implemented Flex in Xen and performed comprehensive evaluations with various parallel workloads. Experimental results show that Flex achieves fairness between heterogeneous VMs with on average no more than 5% deviation from the ideal fair allocation. Further, Flex is able to realize a reasonably good level of differentiation and proportional allocation between VMs with different weights. Finally, Flex effectively reduces spinning time for busy-waiting-based workloads and avoids vCPU stacking for blocking-based workloads. Compared to default Xen, Flex achieves similar performance for VMs with advantages in CPU allocation and significantly boosts the performance of VMs with disadvantages. Flex performs closely to two representative co-scheduling and demand-based scheduling approaches in optimizing a single workload. When simultaneously optimizing a mix of two parallel workloads, Flex significantly outperforms both approaches by 30.4% and 35%, respectively. Moreover, Flex only incurs less than 1% of overhead to the critical execution path of Xen.

The rest of the paper is organized as follows. Section 2 introduces the basics of SMP VM scheduling and parallel synchronization, discusses challenges in attaining VM-level fairness and parallel efficiency. Section 3 and Section 4 describe the design and implementation of Flex, respectively. Section 5 presents the experiment results. Section 6 discusses related work and Section 7 concludes this paper.

2. Background and Motivation

In this section, we first describe the basics of SMP virtual machine scheduling and discuss parallel workloads with different synchronization mechanisms. Then, we demonstrate the difficulty of attaining fairness between SMP VMs in a multicore scenario. Finally, we show that existing scheduling strategies for fairness are inaccurate and introduce significant inefficiencies to parallel workloads.

2.1 SMP Virtual Machine Scheduling

Symmetric MultiProcessing (SMP) VMs, each configured with two or more virtual CPUs (vCPUs), allow users to simultaneously access multiple processors. Therefore, SMP VMs are widely used for hosting parallel workloads. In a virtualized environment, there exists a double scheduling scenario [25], where a guest operating system (OS) schedules processes on vCPUs and the hypervisor schedules vCPUs on physical CPUs (pCPUs). In systems with co-located VMs, multiple vCPUs could run on the same pCPU and the hypervisor allocates CPU cycles according to the *share* of each vCPU. To enforce fairness between VMs, the hypervisor assigns equal shares to individual VMs and the shares are further evenly distributed to vCPUs. Thus, VMs with a smaller number of vCPUs will receive a larger per-vCPU share. When a vCPU finishes running, its share is updated based on how long it ran on the pCPU.

Besides CPU shares, most systems also use an upper bound (e.g., a cap) to limit the maximum amount of CPU a VM is able to consume, even if the systems have idle CPU cycles. Such a resource limit effectively prevents rogue VMs from monopolizing all the resources and realizes performance isolation between VMs. In multicore systems, for scalability considerations, CPUs run independent copies of the scheduler. Load imbalance on different CPUs compromises the overall throughput and responsiveness. Thus, the schedulers perform load-balancing to evenly distribute vCPUs onto pCPUs. In general, there are two approaches to load balancing: *push migration* and *pull migration*. In push migration, the load-balancer periodically checks load balance in the memory hierarchy and pushes vCPUs from a busy node (e.g., a scheduling domain in Linux CFS scheduler) to a less-busy one if an imbalance is found. Pull migrations occur when a pCPU becomes idle and steals (or pulls) a waiting vCPU from a busy pCPU.

2.2 Parallel Program

Parallel programs break large problems into smaller tasks and solve them concurrently. The performance of parallel programs depends critically on the efficiency of task synchronization. We introduce different synchronization methods and discuss their issues in virtualized environments.

2.2.1 Task Synchronization

Task synchronization is needed to ensure correctness if multiple tasks share data with each other. There are many synchronization primitives designed for different purposes, such as `mutex/semaphore`, `spinlock` and `barrier`, typically there are two ways [1] to deal with a task that waits to access the critical section.

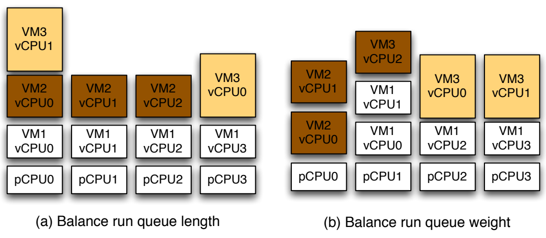

Figure 2. vCPU-to-pCPU mappings for three heterogeneous VMs under different load-balancing policies. All VMs have the same share and the size of each vCPU represents the per-vCPU share.

Busy-waiting (spinning). The task simply stays in a busy loop and repeatedly checks if the lock is available. Spinning is efficient if synchronization is expected to be very short and the task remains active to avoid expensive context switches. However, long spinning will lead to wasted CPU cycles that would be otherwise used by other tasks. Programs using busy-waiting synchronization are more susceptible to lock-holder preemptions (LHP) in virtualized environments.
Non-busy-waiting (blocking). The task voluntarily goes to sleep when fails to acquire the lock and is later awoken by the scheduler once the lock is released. Blocking effectively avoids wasted CPU cycles but requires frequent context switches. As shown in [11], spinlocks are also used to protect the queue that holds sleeping tasks when the lock is released and the scheduler dequeues (awakes) one task from the queue. Thus, blocking synchronizations are not immune to the inefficiencies in virtualized environments.

2.3 Challenges

Next, we discuss the challenges in attaining fairness and efficiency when scheduling SMP VMs in multicore systems.

2.3.1 Enforcing VM-level Fairness

Fair SMP VM scheduling requires that the aggregate CPU allocation to all vCPUs of a VM be proportional to its weight (share) in competition with other VMs sharing the same set of pCPUs. If all VMs have the same weight, each VM should receive the same amount of CPU time no matter how many vCPUs a VM has. As shown in Figure 1(a), existing VM schedulers fail to enforce VM-level fairness between heterogeneous VMs with different numbers of vCPUs. Although vCPU weights determine the relative CPU allocations on a pCPU, the absolute allocation depends on the total weight on the pCPU. In Figure 2, we demonstrate how the divergence on pCPU total weights leads to unfair allocation at VM level. We place three heterogeneous VMs, each with 4vCPU, 3vCPU and 2vCPU, respectively, on four pCPUs.

[1] Although a hybrid approach is possible, e.g., `spin-then-block`, we treat it as an application of the two fundamental approaches.

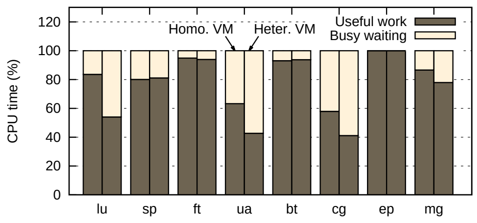

Figure 3. Imbalanced vCPU time allocation due to the co-location of heterogeneous VMs. Fairness capping introduces excessive busy-waiting time for NPB applications.

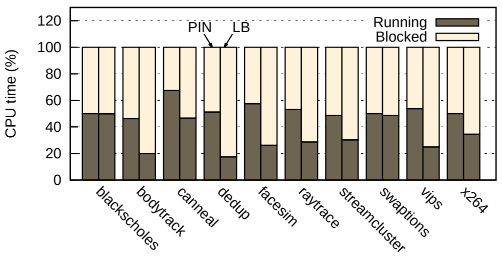

Figure 4. Dynamic load balancing (LB) penalizes PARSEC applications with blocking synchronization.

Figure 2(a) shows a typical vCPU mapping policy that balances the run queue size on each pCPU. For fairness, all VMs are given equal shares and their per-vCPU weights are inversely proportional to their vCPU counts.

Since there is an odd number of vCPUs on an even number of pCPUs, load imbalance inevitably exists. Thus, CPU allocations on vCPUs are no longer proportional to their weights. For example, `VM3-vCPU1` receives less CPU time than `VM3-vCPU0` because its allocation is diluted by a larger total weight on `pCPU0`. More importantly, the allocations on `VM3-vCPU1` , `VM1-vCPU1` and `VM2-vCPU1` are not proportional to their weights, breaking VM-level fairness. A possible solution is to balance the run queue weight (as shown in Figure 2(b)) to minimize vCPU weight dilution. However, perfect weight balance on different pCPUs is not always possible and such balancing may introduce the vCPU stacking issue [27], which limits the effective parallelism and increases the synchronization latency in SMP VMs. In production systems, VM management software often pre-calculates the fair share of individual VMs and applies a resource *cap* to each VM, preventing it from consuming excessive CPU time. With a cap, vCPUs are stopped if the total CPU allocation of a VM exceeds its fair share. However, the use of resource cap turns the system into non-work-conserving mode and may lead to wasted resources.

2.3.2 Maximizing Parallel Efficiency

Synchronization overhead has been accused of limiting the scalability and performance of parallel programs in a virtualized environment [7, 11, 27, 32]. Lock-holder preemption (LHP) [32] and vCPU stacking [27] are two uniques issues that arise with SMP VM scheduling. We show that mechanisms for enforcing fairness inadvertently exacerbate these synchronization issues.

The LHP issue refers to the preemption of the vCPU that holds an important lock in the guest OS. Such preemptions seriously increase synchronization latency to milliseconds as opposed to a few microseconds in physical systems. In the case of spinlock holder preemption, CPU time is wasted when other sibling vCPUs busy-wait for the preempted vCPU to release the lock. We find that fairness mechanisms, such as vCPU capping, can lead to excessive busy-waiting or wasted CPU time. We ran the NASA parallel benchmark (NPB) [5] in a 4-vCPU VM co-located with a heterogeneous 3-vCPU VM on four pCPUs. The background 3-vCPU VM ran three `while(1)` loops to burn CPU time. To enforce fairness, we applied a cap equivalent to the time of two CPUs to each VM. The OpenMP version of NPB was used and compiled with option `OMP_WAIT_POLICY` set to active. With this setting, threads stay in spin loops while waiting for other threads. For comparison, we also ran the NPB benchmark with a homogeneous 4-vCPU VM in the background. In this setting, no vCPU capping is needed. We instrumented NPB source code to calculate the total busy-waiting time spent by all threads.

Figure 3 shows that the co-location of heterogeneous VMs caused excessive busy-waiting in NPB applications. In heterogeneous co-location, the vCPU that had less contention on CPU time spent significant portion of time waiting for other slower vCPUs. When vCPU capping is in place, such wasted time is counted towards the fair share of the VM. As such, busy-waiting wastes the time could otherwise be used by useful work. Note that applications that dynamically adjust work assignment for application threads (e.g., `ft`, `bt`, `sp`) and applications without synchronization (e.g., `ep`) are resilient to such imbalance caused by vCPU capping.

As discussed in Section 2.3.1, load-balancing helps enforce VM-level fairness by averaging the run queue size or weight on different pCPUs so that actual CPU consumptions approximate VM weights. For example, pCPUs with less load will become idle earlier and steal (pull) runnable vCPUs from busier pCPUs. However, our findings reveal that such vCPU migrations cause severe slowdowns to parallel applications with blocking synchronization. We ran the PARSEC [30] benchmark in a 4-vCPU VM along with a 3-vCPU background VM running `while(1)` loops. PARSEC is composed of multithreaded programs that use `Pthread` blocking synchronization primitives. If failing to enter the critical section, a thread blocks itself and goes to sleep. Figure 4 draws the time that threads spent in running and blocked states under two scheduling policies. `PIN` binds vCPUs to pCPUs and ensures that no vCPUs belonging to the same VM reside on one pCPU. It is similar to *balance scheduling* [27] and may incur imbalanced CPU allocation. `Load-balancing (LB)` allows vCPU migrations for fairness. From the figure, we can see that `LB` incurs excessive vCPU blocking to PARSEC applications. When PARSEC threads block themselves, the vCPUs that carry these threads become idle and also get blocked by the hypervisor. Thus, the corresponding pCPUs become idle and start to steal vCPUs from other pCPUs. Since vCPU migrations only steal runnable (not actually running) vCPUs on the run queue, it is likely that vCPUs belonging to PARSEC applications are stolen as vCPUs running the `while(1)` loop never block and are likely in the running state during the steal. As a result, load-balancing causes severe vCPU stacking issues for application with blocking synchronization.

[**Summary**] In this section, we have shown that it is difficult to achieve VM-level fairness in multicore systems and existing solutions are likely to cause LHP and vCPU stacking issues. These findings motivated us to develop a SMP VM scheduling scheme that separates fairness enforcement from the rest of the scheduler and is carefully designed for improving parallel efficiency. To this end, we design FlexW, a vCPU accounting scheme that dynamically adjusts VM weights to realize fair allocation at the VM level, and FlexS, a flexible vCPU scheduling algorithm that eliminates wasted busy-waiting time.

Algorithm 1 Flexible vCPU Weight Adjustment

```
1: Variables: Virtual CPU $v$; Weight of the $i_{th}$ VM $w_i$; Real-time
   weight of the $i_{th}$ VM $w_i^r$; Number of shared CPUs $P$;
2:
3: /* System-wide accounting for period $(t_1, t_2)$ */
4: procedure CPU_ACCOUNTING($void$)
5:     $acct\_count$++
6:     for each VM do
7:         for each vCPU do
8:             cpus_and(workers, workers, v $\rightarrow$ cpu_affinity)
9:         end for
10:    end for
11:    $P = cpus\_weight(workers)$
12:    for each VM do
13:        $S_{i,GPS}(t_1,t_2) = \frac{w_i}{\sum w_j}(t_2 - t_1) \cdot P$
14:        $lag_i(t_1,t_2) = \frac{S_{i,GPS}(t_1,t_2) - S_i(t_1,t_2)}{S_{i,GPS}(t_1,t_2)}$
15:        $w_i^r = w_i^r + w_i \cdot lag_i(t_1,t_2)$
16:        if $acct\_count >$ FAIR_WINDOW then
17:            $acct\_count = 0$
18:            $w_i^r = w_i$
19:        end if
20:    end for
21: end procedure
```

3. Flex Scheduling for Fairness and Efficiency

Based on our findings, we attribute the unfairness in a heterogeneous VM co-location scenario to the inaccurate CPU allocation caused by diluted vCPU weights. We also find that excessive busy-waiting and vCPU stacking are the culprits of parallel performance slowdowns. Therefore, we try to answer the following questions when designing Flex. (1) *How to adaptively change vCPU weights to achieve VM-level fairness*? (2) *How to schedule vCPUs to eliminate busy-waitings*? (3) *How to avoid vCPU stacking*?

3.1 Overview

Flex centers on two key designs: flexible vCPU weight adjustment (FlexW) and flexible vCPU scheduling (FlexS). FlexW is a system-wide CPU accounting daemon that periodically monitors the actual CPU consumptions of all VMs. It calculates the desired fair share of individual VMs based on their weights. If there is a difference between actual allotted CPU time and the fair share, FlexW adjusts VM weights to compensate the discrepancy. FlexS is part of the vCPU scheduling module running on individual pCPUs. It non-intrusively detects busy-waiting vCPUs according to hardware metrics and preempts such vCPUs to avoid wasted CPU time. Before scheduling a vCPU from other VMs, FlexS tries to steal a sibling vCPU that is doing useful work from other pCPUs' run queues. In the following subsections, we present the details of each design.

3.2 Flexible vCPU Weight Adjustment

Algorithm 1 shows the flexible vCPU weight adjustment algorithm. For each accounting period, FlexW first determines the number of pCPUs shared by all VMs. Note that VMs may share different sets of pCPUs and form multiple accounting groups, each of which requires fair allocation within the group. In this work, we assume a single accounting group that shares the same set of pCPUs and leave enforcing fairness in multiple groups to future work. Then the total CPU time for shared P CPUs during time period (t_1, t_2) becomes $(t_2 - t_1) \cdot P$. Next, FlexW calculates the fair allocation $S_{i,GPS}(t_1, t_2)$ (line 12) under the idealized *Generalized Processor Sharing (GPS)* [21] algorithm using a VM's original weight w_i. The lag of a vCPU is the normalized difference between the fair allocation $S_{i,GPS}(t_1, t_2)$ and its actual consumed CPU time $S_i(t_1, t_2)$ (line 13). A positive lag indicates that the vCPU has received less time than under GPS [14] and vice versa. Finally, FlexW determines the real-time weight w_i^r of the VM based on its lag (in percentage) relative to the fair allocation and uses the adjusted weight in the next accounting interval. Note that we bring the lag to the same scale of weights by multiplying it with the original weight w_i.

There are many practical issues FlexW needs to deal with. The work-conserving property still needs to be preserved when enforcing fairness. Before calculating the fair share for each VM, FlexW checks if the total consumed CPU time equals to the available time (i.e., $(t_2 - t_1) \cdot P$). If VMs do not consume all CPU time, FlexW simply quits the weight adjustment process and set the real-time weights to the original weights. As such, the CPU time a VM could use is only limited when the system is overcommitted. Another issue is *infeasible weight* [21], where a VM's peak CPU consumption is smaller than its fair share. For example, a 2-vCPU VM's weight becomes infeasible if the fair share for this VM is equivalent to the time of 3 CPU. To this end, FlexW only calculates the fair share of VMs with feasible weight and uses the peak consumption as the fair share of VMs with infeasible weight.

Although FlexW resets VMs' real-time weights once the total CPU demand is below the available CPU time, it is possible that some VMs demand less than their fair share and others consume more than their share. In this case, FlexW still considers the system under work-conserving mode and keeps updating VMs' real-weights. As a result, the VMs that voluntarily demand less CPU will have ever-increasing real weights and will monopolize the CPU once their demands increase. Our solution is to keep a limited history of allocations and enforce fairness within a `FAIR_WINDOW`. All real-time weights are reset to original weights when switching to the next fair window (line 16). This design effectively prevents VMs from occupying the CPU for too long, but still gives frequent idling VMs higher shares within a fair window when they wake up.

3.3 Flexible vCPU Scheduling

While FlexW enforces fairness between VMs, it does not guarantee efficient SMP VM scheduling. It even introduces load imbalance as some vCPUs receive less CPU time than their siblings. As discussed in Section 2.3.2, load imbalance leads to excessive spin time for busy-waiting-based workloads and vCPU stacking for blocking-based workloads. Co-scheduling [20] has been considered most effective in minimizing synchronization latency, but suffers CPU fragmentation and priority inversion issues. Scheduling and de-scheduling vCPUs together is even harder under fairness constraints. When designing FlexS, we do not attempt to explicitly co-schedule vCPUs. Instead, we believe that synchronization efficiency is maximized if CPU time is only used for useful work during an accounting period. Thus, the objectives of FlexS are to de-schedule busy-waiting vCPUs and schedule vCPUs doing useful work as much as possible.

3.3.1 Identifying Busy-waiting vCPUs

Since busy-waiting avoids expensive context switches, it is widely used in application libraries and OS kernels. There is existing work detecting excessive busy-waiting by instrumenting Linux kernel to report spinning statistics [34] or by non-intrusively monitoring user-kernel mode switches [32] or by counting store instructions [7]. Besides kernel instrumentation being not always possible, existing non-intrusive approaches are not applicable to user-level busy-waiting implementations [32] or not accurate for different workloads [7]. For example, it is difficult to set a threshold for store-based spin detection and some applications (e.g., `lu`)

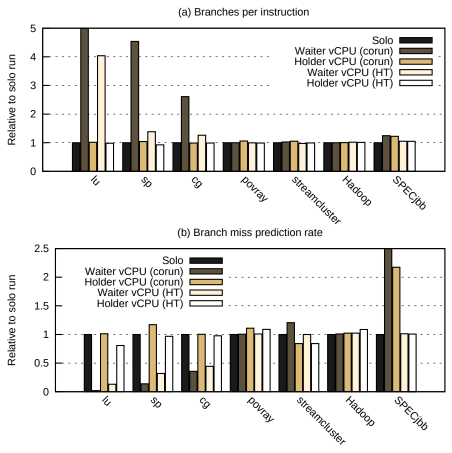

Figure 5. Branches per instruction (BPI) and Branch miss prediction rate (BMPR) together accurately identify busy-waiting vCPUs.

have inherent differences in the frequency of issuing stores on different threads (e.g., the master thread issues 20X times more stores than others). To this end, we design a simple but effective mechanism for busy-waiting detection.

Similar to hardware-assisted spin detection [9], which identifies busy-waiting vCPUs as ones execute excessive `PAUSE` instructions, our approach also finds a common pattern that spans across different spin implementations. We find that spin loops are usually designed to be highly efficient and contain only a few instructions. Such short loops are executed many times until the lock is acquired. Compared to other loops at both user and kernel levels, spin loops show a high branch per instruction (BPI) rate. Further, as the loops are repeatedly executed, the hardware branch predictor has a low branch miss prediction rate (BMPR). More importantly, spin loops always accompany with a lock-holder who is running the normal code. Thus, high BPI rates and low BMPR values along with distinctive behaviors from other vCPUs are good indicators of busy-waiting vCPUs. To validate our spin detection design, we created scenarios that one vCPU is slower than others. When executing parallel workloads, the slower vCPU is likely the lock/mutex *holder* and the faster ones are likely the *waiters*. We made a vCPU slower by running a `while(1)` loop on the same pCPU (i.e., `corun`) or running the loop on the pCPU's hyperthread (i.e., `HT`). `Solo` (without contention) was used as the baseline. We ran different 4-thread workloads to study the correlation between branch related hardware metrics and high-level synchronization. Details of parallel workloads are explained in Section 5. Besides parallel workloads, we also included a multiprogrammed workload with 4 copies of `povray` from the SPEC CPU2006 benchmark [31].

Figure 5(a) and (b) show that BPI climbs and BMPR plummets on spinning vCPUs (e.g., waiters in `lu`, `sp` and `cg`). Interestingly, BPI and BMPR not only qualitatively identify busy-waiting vCPUs but also quantitatively re-

Algorithm 2 Flexible vCPU Scheduling

```
1:  Variables: Virtual CPU $v$; Maximum BPI ($max\_bpi$) in $v$'s VM; Minimum BMPR ($min\_bmpr$) in $v$'s VM.
2:
3:  /* Periodic local vCPU accounting*/
4:  procedure VCPU_ACCOUNTING($v$)
5:      BURN_CREDITS($v$)
6:      Update $v$'s BPI and BMPR values
7:      /* If vCPU is busy waiting, call SCHEDULE() and yield to others */
8:      if (IS_BUSY_WAITING($v$)) then
9:          $v \rightarrow yield = $ TURE
10:         $raise\_softirq$(SCHEDULE_SOFTIRQ)
11:     else
12:         $v \rightarrow yield = $ FALSE
13:     end if
14: end procedure
15:
16: procedure IS_BUSY_WAITING($v$)
17:     /*BPI close to max and BMPR close to min indicate spinning*/
18:     if $\frac{(max\_bpi - v \rightarrow bpi)*10}{max\_bpi} == 0$ and $\frac{(v \rightarrow bmpr - min\_bmpr)*10}{min\_bmpr} == 0$ then
19:         return TRUE
20:     else
21:         return FALSE
22:     end if
23: end procedure
24:
25: procedure SCHEDULE($void$)
26:     /* If just yielded, switch this vCPU with a sibling on another CPU. Otherwise, follow the regular path. */
27:     if $curr \rightarrow yield$ == TRUE then
28:         $next = $ LOAD_BALANCE($curr$, SWITCH)
29:     else
30:         if load is balanced then
31:             $next = $ next vCPU in local run queue
32:         else
33:             $next = $ LOAD_BALANCE($curr$, STEAL)
34:         end if
35:     end if
36:     Context switch to $next$
37: end procedure
```

flect the level of contention. For example, BPI is lower and BMPR is higher in HT than in Corun as the loop in HT does not contend CPU cycles but only shared resources like the reservation station and caches. Because synchronization contributes most to the dynamic portion of program execution, the number of branches executed per instruction stays relatively stable for programs with no or little synchronization no matter there is contention or not. As shown in Figure 5(a), independent copies of povray have similar BPI values in all scenarios. Similar conclusions can also be made for workloads with blocking synchronization (e.g., streamcluster and SPECjbb) as spinlocks are only used when operating on the queue of waiting threads [11]. Since Hadoop workload embraces the embarrassing parallel model and its map tasks are largely independent of each other, its BPI also does not change with contentions. Figure 5(b) shows that BMPR changes more significantly than BPI for blocking workloads. One possible explanation is that the frequent switches between vCPUs due to voluntary vCPU blocking pollute the history of branch predictors leading to higher BMPRs. Nevertheless, due to the lack of short spin loops, BMPR never drops dramatically for these workloads.

Based on our observations, we use a simple heuristic to identify busy-waiting vCPUs – *If some vCPUs have the highest BPIs and the lowest BMPRs among their siblings, they are spinning*. FlexS maintains the maximum BPI and minimum BMPR among all vCPUs in a VM. When deciding a vCPU's busy-waiting status, FlexS calculates the distances of its BPI and BMPR to the VM-wide maximum and minimum. If being close enough (within 10%, line 18 in Algorithm 2), FlexS considers the vCPU a busy-waiter and performs flexible scheduling to eliminate busy-waiting time. It is possible that there may exist false-positive detections as FlexS infers busy-waiting vCPUs based on cross-vCPU comparisons. During program initialization or execution phase changes, some vCPUs can show considerably different behaviors than others and be mistakenly identified as spinning. To this end, FlexS clears BPI and BMPR values when a fair window expires so that program phases could be detected and false positives would not affect next detection windows.

3.3.2 Eliminating Busy-waiting Time

The key to eliminating busy-waiting is to stop spinning vCPUs and run vCPUs with useful work. Algorithm 2 shows the design of flexible vCPU scheduling. During periodic local vCPU accounting, FlexS updates a vCPU's BPI and BMPR values with new performance monitoring unit (PMU) readings. If FlexS identifies the vCPU as busy-waiting (function Is_busy_waiting), it raises a soft IRQ interrupt (line 10) on this pCPU and forces a call to the main Schedule function, where the current vCPU will be de-scheduled and the next-to-run vCPU is selected. Since the de-scheduled vCPU voluntarily yields to others, its unfinished time slice should be used to by its sibling vCPUs with useful work. Thus, the Schedule function has two paths when scheduling vCPUs. If a vCPU just yielded (line 27), FlexS pulls a runnable but not busy-waiting sibling (with a false yield flag) of this vCPU from another pCPU. Otherwise, follow the regular path.

To **avoid vCPU stacking**, Flex integrates balance scheduling into its design. When performing load balancing, we ensure that no vCPUs from the same VM will be stacked on one pCPU. For the yielded vCPU path, FlexS switches two vCPUs on two pCPUs so that no stacking could happen. For the regular path, if load is imbalanced (line 33), only vCPUs that have no siblings on the current pCPU could be stolen. However, frequent vCPU migrations will likely violate the fairness enforced by FlexW. This is because switching vCPUs (usually with different weights) changes the total weights of pCPUs. As discussed in Section 2.3.1, such a divergence in pCPU weights causes unfairness. We find that VMs that migrate vCPUs frequently tend to gain advantage in CPU allocation. To **preserve fairness**, we ensure that vCPU migrations do not affect the total weight on individual pCPUs. Specifically, FlexS exchanges the weights of two vCPUs when switching them. Besides the weight, a vCPU's relative position in its run queue also affects its CPU allocation. To this end, FlexS does not insert the switched out vCPU onto another pCPU's run queue, which will always put the vCPU on the tail of the run queue. Instead, the run queue pointers of the switching vCPUs are exchanged. As such, no run queue operations are needed on both pCPUs and the run queue positions are preserved.

4. Implementation

We implemented Flex in Xen (version 4.0.2) and patched Xen with *Perfctr-Xen* [18] to access low-level hardware performance counters. Perfctr-Xen maintains a per-vCPU data structure to store the values of hardware counters. We updated counter values every time (every 10 ms) Xen performs vCPU accounting. Xen uses *credit* to represent weight. For every 10 ms, a per-pCPU accounting routine burns the current running vCPU's credit. A system-wide master routine fires up every 30 ms to replenish credits for VMs based on their weights. The higher the weight, the more credits a VM receives per master accounting period. To calculate BPI and BMPR, we calculated the ratio of hardware events `BR_INST_EXEC` and `INST_RETIRED`, `BR_MISP_EXEC` and `BR_INST_EXEC`, respectively.

FlexW is implemented in the system-wide master accounting routine `csched_acct()`, where it iterates over all VMs and vCPUs. The iteration determines the total credit for the pCPUs shared by vCPUs, the `max_bpi` and `min_bmpr` of each VM, and the total consumed credits for each VM. FlexW computes the credits to be assigned to individual VMs in the next interval based on their real time weights. We added a global flag in Xen to record VM creation/termination and vCPU affinity changes. If nothing has changed since last accounting, we avoid the iteration that calculates the total available credit (Algorithm 1, line 6-9). We empirically set `FAIR_WINDOW` to 10 seconds, after which we reset VMs' real-time weights. To catch program phase changes, we also clear a VM's `max_bpi` and `min_bmpr` when a fair window expires.

The key to the effectiveness of FlexS in minimizing busy-waiting time is to find vCPUs with useful work on remote pCPUs. A per-pCPU run queue lock needs to be acquired from a remote pCPU before any vCPU can be stolen. Both a stealing pCPU and the main `schedule` function on a remote pCPU can contend for the lock. Note that the per-pCPU accounting timers have the same 10 ms interval and the per-pCPU `schedule` timers have intervals of 30 ms (exactly three times of the accounting timers). Thus, a stealing pCPU is likely to collide with another stealing pCPU or the local `schedule` function. We make the per-pCPU accounting timer dynamic to avoid the lock contention. FlexS keeps a counter for each vCPU to record how many times it failed to steal a remote vCPU. FlexS sets the next vCPU accounting timer by subtracting its counter value from the default 10 ms. The counter is cleared when a vCPU succeeds in the stealing. Therefore, vCPUs wait for different amount of time before they try to steal again. vCPUs with higher failure counts will perform the stealing earlier than the ones recently succeeded in stealing. This effectively increases the success rate of stealing and leads to less wasted busy-waiting time. Iterating over all pCPUs for the stealing is not scalable as the number of shared pCPUs increases. To bound the steal time, we borrow the idea of the power of two choices [16]. That is, a vCPU iterates over two pCPUs for stealing before it quits.

5. Evaluation

In this section, we present an evaluation of Flex using various parallel workloads. We study the effectiveness of Flex in enforcing fairness and realizing differentiation in a multicore scenario (Section 5.1). Then we compare the performance of Flex with Xen's default credit scheduler and two representative co-scheduling approaches (Section 5.2). Finally, we study the overhead of Flex (Section 5.3).

We implemented Flex in Xen 4.0.2 and deployed our prototype on Dell PowerEdge T420, equipped with two quad-core Intel Xeon E5620 2.4GHz processors and 16GB memory. To evaluate the fairness and performance of Flex, we created two heterogeneous Xen VMs with 4 vCPUs and 3 vCPUs, respectively. Both VMs were configured with 6GB memory and set to the default weight (256 in Xen). We ran Linux kernel 2.6.32 with para-virtualized spinlocks as the guest OS. Both VMs were set to share four cores in one of the two processors. This setting ensures that all vCPUs share the same last-level cache and NUMA node so that the performance of parallel workloads only depends on the allocation of CPU time. For overhead study, we created up to 8 VMs, each with 4 vCPUs.

We selected the following parallel workloads and measured their performance with different approaches.

- **NASA parallel benchmarks (NPB)** [5] include 9 parallel programs derived from computational fluid dynamics applications. We used the OpenMP implementation of the benchmarks and set the problem size to class C. Environment variable `OMP_WAIT_POLICY` was set to active to use busy-waiting synchronization.
- **PARSEC** [30] is a multithreaded shared memory benchmark. Its contains 13 emerging programs that model after divergent application domains. Pthread non-busy-waiting primitives (e.g., mutexes, condition variables and barri-

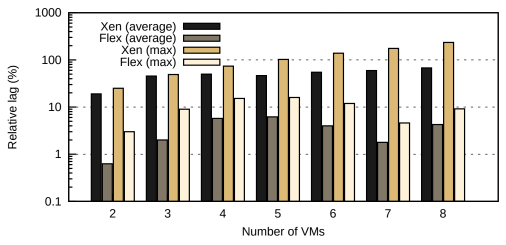

Figure 6. The average and maximum lag of Xen and Flex for different number of VMs.

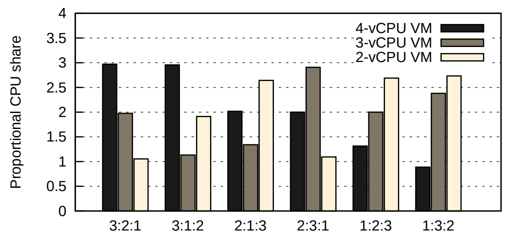

Figure 7. Flex realizes proportional share of CPU between VMs with different numbers of vCPU.

ers) are used for thread synchronization and the native input size was used.

- **Hadoop** is a popular implementation of the MapReduce framework for running data-parallel jobs. We selected a Bayesian classification job in the Hadoop Mahout machine learning library [28]. The job classifies 20,000 newsgroup documents into 20 categories. It consists of three phases of execution, each of which contains multiple map tasks and one reduce task. During each phase, the map tasks are independent of each other with no communications. The next phase starts only when the reduce task finishes in the previous phase.
- **SPECjbb2005** [26] evaluates the performance of server side Java by emulating a three-tier client/server system. It spawns multiple threads to emulate active users posting transaction requests in multiple warehouses within a wholesale company. Synchronization is needed when customer requests and company internal management operations both work on the same database table. Synchronized methods in Java are used to block waiting threads.

If not otherwise stated, we matched the number of threads/maps in these workloads with the number of vCPUs in the VMs.

For comparison, we evaluated three scheduling strategies:

- `Xen`: The default credit scheduler without mechanisms for VM-level fairness.
- `Balance+cap+CO`: For busy-waiting workloads, we used *balance scheduling* [27] with fairness capping. We also set the VM running parallel workloads to a higher priority to co-schedule its vCPUs.
- `Demand+cap`: For non-busy-waiting workloads, we implemented *demand-based coordinated scheduling* [11] with fairness capping. We monitored Xen event channel to prevent the preemption of sender vCPUs of interprocessor interrupts (IPI). Recipients of IPIs are by default prioritized by Xen as wakeup vCPUs are temporally elevated to the `boost` priority [19].

5.1 Fairness and Differentiation

In this subsection, we show the effectiveness of Flex in VM-level fairness and differentiation. We used the *absolute relative lag*, $|\frac{S_{i,GPS}(t_1,t_2)-S_i(t_1,t_2)}{S_{i,GPS}(t_1,t_2)}|$, to measure fairness. We created four types of VMs with 4 vCPUs, 3 vCPUs, 2vCPUs and 1 vCPU, respectively. We changed the number of VMs sharing four cores and ensured that the mix contained heterogeneous VMs as many as possible. Figure 6 shows the maximum and average lag of individual VMs under Xen and Flex. From the figure, we can see that the default credit scheduler fails to enforce fairness at VM-level and the unfairness in terms of both maximum and average lag goes unboundedly to as much as 235% and 67%, respectively. In contrast, Flex achieves one order of magnitude (log scale in the y axis) less lag than Xen. The maximum lag of Flex is bounded by 15% and on average Flex incurs no more than 5% unfairness.

We are also interested in whether Flex can realize service differentiation, where VMs receive CPU time proportional to their weights. We consolidated three VMs, each with 4, 3, 2 vCPUs, respectively, onto four cores. Figure 7 shows the CPU allocation of the three VMs with different combinations of weight. Flex realized almost perfect proportional allocations for weight combination 3:2:1. However, allocations were not accurate when VMs with fewer vCPUs have large weights and VMs with more vCPUs have small weights, e.g., combinations 1:2:3 and 1:3:2. The reason is that Flex enforces proportional allocations for individual fair windows. Once VM real-time weights are reset, VMs with more vCPUs are likely to consume more CPU time, violating the proportionality. It is a trade-off between keeping a limited history of CPU allocation and perfect proportionality. Although not being always accurate, we conclude that Flex realizes a reasonably good level of differentiation.

5.2 Parallel Execution Efficiency

As discussed in Section 2.3.2, imbalanced CPU allocation in SMP VMs incurs excessive busy-waiting or severe vCPU stacking. In this subsection, we show that Flex is able to mitigate such issues and achieve good parallel efficiency. We first study the performance of one SMP VM in two imbalanced scenarios and then evaluate the performance of two SMP VMs with a mix of parallel workloads.

5.2.1 Imbalanced Allocation

We created two imbalanced scenarios, where two heterogeneous VMs, one with 4 vCPUs and one with 3 vCPUs, were co-located on four cores. As shown in Figure 1(a), the VM with more vCPUs gains advantage in getting more

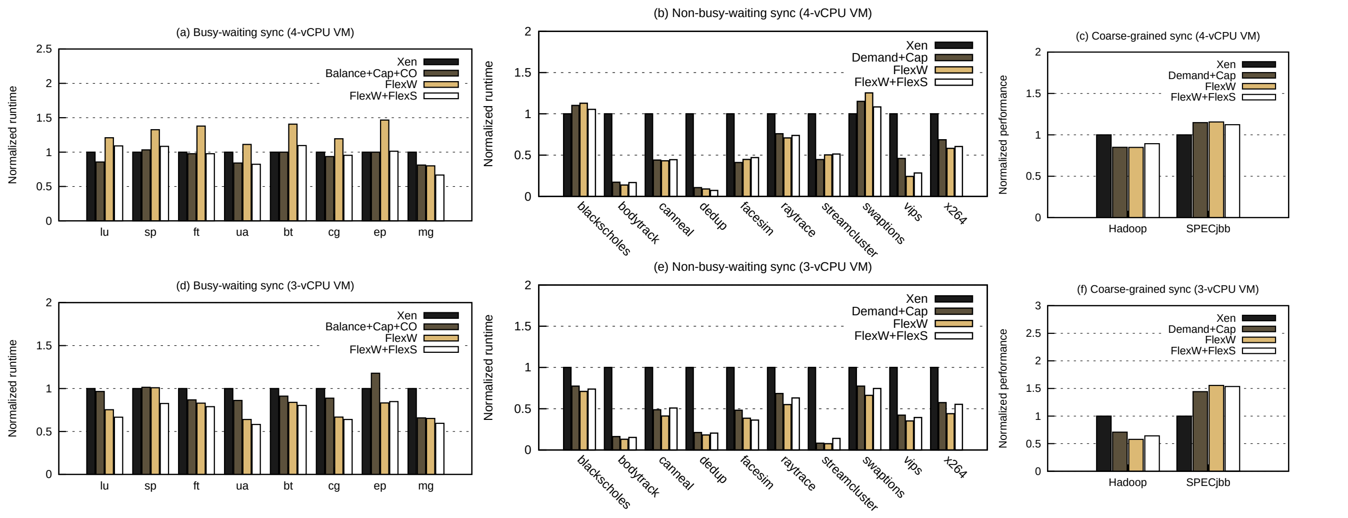

Figure 8. The performance of parallel workloads running with background `while(1)` loops under different policies. Runtime is normalized to baseline Xen and the lower the better. SPECjbb is measured using business operations per second (bops), the higher the better.

CPU time. We ran our parallel workloads in the two VMs in turn and measured their performance under different approaches. We studied two Flex variations: Flex without flexible scheduling (FlexW) and Flex with complete functionalities (FlexW+FlexS). To isolate performance from other factors, e.g., shared cache contention, we ran `while(1)` loops in the other VM.

Figures 8 (a) and (d) show the performance of NPB programs when running in the 4-vCPU and 3-vCPU VMs, respectively. The performance of different approaches is normalized to `Xen`. It is expected that default `Xen` delivered good performance in the 4-vCPU VM scenario because the VM attained more than its fair share. With balance scheduling and co-scheduling, `balance+cap+CO` achieved better performance (e.g., `lu`, `ua` and `mg`) than `Xen` using less CPU time (i.e., fair share). However, `balance+cap+CO` realizes co-scheduling by elevating the priority of the parallel workload, which inevitably hurts the performance of co-running workloads. We will show that this strategy performs badly when optimizing a mix of parallel workloads. Figure 8(a) also shows that `FlexW+FlexS` performed closely to `balance+cap+CO` except for `lu`. An examination of `lu` source code revealed that `lu` statically assigns work to threads and is sensitive to load imbalance. Unfortunately, `FlexW` alone was unable to deliver good performance with as much as 50% slowdown compared to `balance+cap+CO`. It indicates that only guaranteeing fair CPU share is not sufficient to achieve good performance. The key is to make efficient use of the fair share.

Figure 8(d) shows that both `FlexW` and `FlexW+FlexS` outperformed `Xen` and `balance+cap+CO` when NPB ran in the 3-vCPU VM. Similar to Xen, `balance+cap+CO` allocated CPU time less than the fair share to the VM. Although capping effectively limited the co-running VM with `while(1)` loops to its fair share, the VM with parallel workloads was unable to consume its fair share as the imbalance diluted its weight and affected the actual CPU allocation. In contrast, both `FlexW` and `FlexW+FlexS` guaranteed fair share by using flexible weights. `FlexW+FlexS` performed consistently better than `FlexW` except for `ep`, which is an embarrassing parallel benchmark with no synchronizations. Thus, we conclude that `FlexS` effectively improved performance by reducing busy-waiting time.

Figures 8(b) and (e) show the performance of PARSEC benchmarks. As discussed in Section 2.3.2, default Xen incurs severe vCPU stacking issues to non-busy-waiting workloads when these workloads co-run with `while(1)` loops. Except for Xen, all the other three approaches avoids vCPU stacking. Therefore, their performance improvement over Xen was quite significant (as much as 10X for `bodytrack`). Because `blackscholes` and `swaptions` had little communication between threads, the vCPUs running these threads were less frequently blocked. The load-balancer in Xen effectively spread the vCPUs and avoided vCPU stacking. Thus, Xen outperformed other approaches in these two benchmarks (in Figure 8(b)) as the VM consumed more than its fair share under Xen. Similarly, the fairness capping in `demand+cap` stopped the `while(1)` VM from monopolizing CPU time and helped the load-balancer spread vCPUs. Among the best-performing approaches, Flex-based approaches outperformed `demand+cap` in the 3-vCPU VM scenario because `demand+cap` also had similar issues (as discussed above) attaining the fair share for the 3-vCPU VM. For the 4-vCPU VM scenario, `demand+cap` outperformed

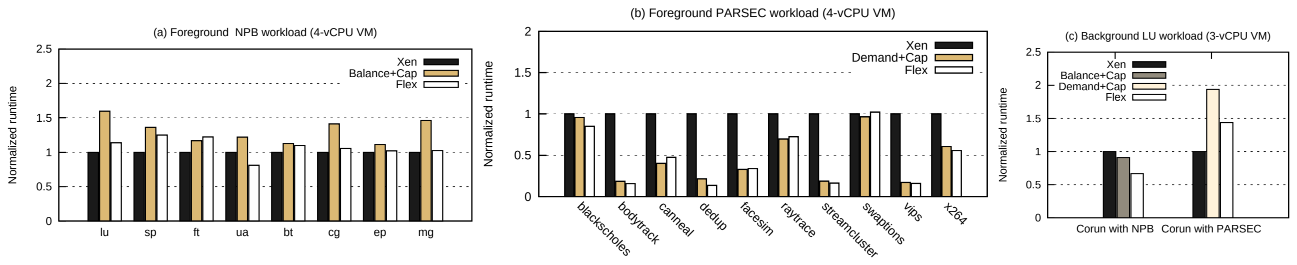

Figure 9. The performance of two parallel workloads running together. NPB and PARSEC run in the foreground and a copy of `lu` benchmark from NPB runs repeatedly in the background.

Flex-based approaches in `facesim` and `streamcluster`. These workloads are communication-intensive and IPI signals used by `demand+cap` provide more useful information about IPI senders and recipients than the hardware-level metrics. Another important observation is that `FlexW` achieved better performance than `FlexW+FlexS` in most workloads. The overhead of frequent vCPU migrations in `FlexS` outweighed its benefit of reducing busy-waiting as spinning only contributes to a small portion of thread synchronization in non-busy-waiting workloads.

Figures 8(c) and (f) show the performance of `Hadoop` and `SPECjbb`. These workloads have coarse-grained synchronization and do not benefit a lot from communication-aware scheduling schemes. The performance improvement of Flex-based approaches and `demand+cap` over Xen was due to the avoidance of vCPU stacking. Interestingly, `FlexW` outperformed `demand+cap` by 15% for Hadoop in the 3-vCPU VM scenario. The reason is that Flex kept increasing the VM's weight during the reduce phase as the CPU demand is below the fair share. This effectively accelerated the reduce task. In contrast, although the vCPU running the reduce task received all the share of the VM (other vCPUs were idle) in `demand+cap`, the actual CPU allocation was less than that in `Flex`. Therefore, Flex is able to accelerate the sequential portion of the `Hadoop` workload.

5.2.2 Mix of Parallel Workloads

In this experiment, we study a more practical scenario, where both co-located VMs ran parallel workloads. In this scenario, a scheduling scheme needs to optimize the performance of both workloads. As discussed above, `lu` employs static work assignment and is sensitive to the scheduling scheme. We used `lu` as the background workload and ran it repeatedly in the 3-vCPU VM. The foreground VM ran the NPB and PARSEC benchmarks in turn. We measured the performance of a scheme by averaging the performance relative to Xen for both foreground and background benchmarks. Since it is not possible to manually prioritize both workloads, we used `balance+cap` in this experiment. We use `Flex` to represent Flex with complete functionalities.

Figures 9(a), (b) and (c) show the performance of the foreground and background workloads. As expected, Xen performed best for the foreground NPB workload as it allocated more than the fair share to it. However, this scarified the performance of `lu` in the background. Compared to `Flex`, Xen incurred 40% performance degradation to `lu` in Figure 9(c). Figures 9(a) and (c) also show that `Flex` achieved consistently better performance than `Balance+cap` in both foreground and background workloads. This confirms the observations in [11] that balance scheduling is not sufficient to optimize parallel performance in virtualized environments. In Figure 9(b), `Flex` performed closely to `demand+cap` in most workloads. However, `demand+cap` performed badly for the background `lu` as it is not designed for workload with busy-waiting loops. Overall, `Flex` outperformed `Balance+cap` and `demand+cap` by 30.4% and 35%, respectively.

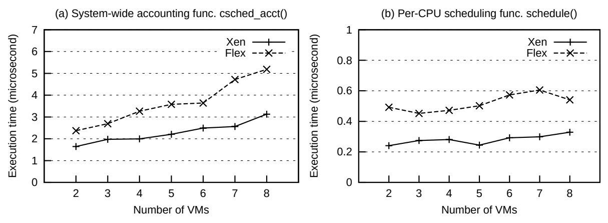

Figure 10. Overhead of Flex.

5.3 Overhead

The overhead of Flex comes from two main sources: (1) the time required to adjust VM weights in the system-wide accounting function `csched_acct`; (2) the time required to perform vCPU stealing in the per-pCPU `schedule` function. Figures 10(a) and (b) show the execution time (in microsecond) of these two functions under Xen and Flex as the number of VMs increases. Since calculating VM weights requires the iteration over all vCPUs, the execution time of `csched_acct` in Flex increases faster than in Xen. However, the increase does not incur overhead to parallel programs as the global accounting is performed by the hypervisor's *idle* VM. The overhead in the main `schedule` function on individual pCPUs can be more significant to parallel performance. A prolonged `schedule` function will take up the time that can be used to run vCPUs. Figure 10 (b) shows that the distance between the execution time of `schedule` in Flex and in Xen does not increase with the number of VMs. Recall that Flex only tries to steal a vCPU from remote pCPUs for two times, its overhead compared to Xen in `schedule` is constant. Considering a scheduling interval of 30ms, Flex only adds less than 1% overhead to Xen.

6. Related Work

An important aspect of resource management is enforcing fairness across different entities. Fairness becomes even more critical for resource allocations in public cloud services because as a metered service, users expect the outcome of a cloud proportionate to their payment. It is believed that fairness in allocating computing resources like CPU cycles is well understood [3]. Thus, most work focused on addressing the unfairness in shared CPU caches [17], network I/O interface [3, 19] and shared storage [23]. Li *et al.* [14] were the first to find that scheduling algorithms based on per-CPU run queues are weak in fairness in a multicore scenario. For such algorithms, the allocation error relative to the ideal Generalized Processor Sharing [21] grows unboundedly with the number of threads and CPUs in the system. Li *et al.* proposed distributed weighted round robin (DRWW) which uses inter-CPU thread migrations to approximate GPS with a constant error. DRWW does not consider fairness for a group of threads thus is not applicable to scheduling SMP VMs. Linux `cgroup` [15] is designed for the fairness of a group of processes. However, as shown in Figure 1(a), it is weak in fairness when load is imbalanced on multiple CPUs. In contrast, Flex continuously monitors the actual CPU allocation for each VM and calculates the corresponding lag to the ideal allocation. Based on the lags, Flex adjusts VM weights accordingly. Although Flex does not explicitly address weight dilution due to load imbalance, it uses feedbacks of actual allocations to reduce unfairness.

There is a large body of work dedicated to addressing the synchronization overhead in parallel programs. Coscheduling of threads [20] refers to the scheme that schedules and de-schedules theads belonging to the same parallel application synchronously. Although it maximizes synchronization efficiency, it suffers from CPU fragmentation and priority inversion issues [13]. Demand-based scheduling, a less strict form of coscheduling, only involves the scheduling of communicating threads [4, 8, 24]. In a virtualized environment, there is a semantic gap between the hypervisor and guest OSes. Weng *et al.* [34] proposed to instrument the guest OS to report the statistics of spinlock holders and dynamically schedule them in the hypervisor. Because intrusive approaches are not always possible, work has been done to infer lock holders from user-kernel mode switches [32] or from low-level hardware statistics about store instructions [7]. However, these approaches are not applicable to all scenarios. For example, libraries with synchronization entirely in user mode (e.g., OpenMP) do not incur any user-kernel mode switches or involve blocking. The frequency of store instructions may not be a good indicator of spinning for workloads with inherently different store issue rates in different threads. We found that the spin loops, no matter implemented at what levels, are shorter iterations compared to loops in regular code. Thus, our approach using BPI and BMPR to identify busy-waiting vCPUs is generic and applicable to a wide variety of workloads.

Modern CPUs provide hardware-assisted schemes (e.g., Intel's Pause Loop Exit (PLE) [10] and AMD's Pause Filter [2]) to detect excessive spinning by monitoring the `PAUSE` instruction in a busy-wait loop. Excessive spinning on a vCPU causes a transition from the guest OS to the hypervisor (e.g., via VMEXIT) and relinquishes the CPU to other VMs. However, hardware-based spin detection is only applicable to fully virtualized VMs and incurs frequent expensive VMEXITs to the hypervisor. In contrast, Flex uses a generic and lightweight spin detection mechanism based on hardware performance counters. It is applicable to both fully and para-virtualized VMs and does not require hardware virtualization support or traps into the hypervisor. Further, all the scheduling strategies in our experiments were configured with para-virtualized spinlocks, a software-based spin-then-block mechanism similar to PLE. Our results showed that Flex significantly outperformed the these approaches due to flexible vCPU scheduling.

Flex is closely related to demand-based coordinated scheduling [11], which infers synchronizing threads by observing IPI signals. This approach also addressed the load imbalance issue with load concious balance scheduling. However, it is only applicable to IPI-based synchronization and can not detect spin-based synchronization. Although Flex is designed for eliminating excessive spinning time, its enforcement of short term fairness during the fair window is likely to prioritize the active vCPUs in blocking-based workloads. Moreover, unlike load conscious scheduling [11], which only avoids overloaded CPUs, Flex takes one step forward to coordinate multiple CPUs in switching useful work.

7. Conclusion

Fairness-efficiency trade-offs have always been important issues in resource allocations. In this work, we find the deficiencies of existing hypervisors in enforcing fairness between SMP VMs. Straightforward solutions lead to low efficiency for parallel workloads. This paper proposes a holistic solution, *Flex*, to the fairness and efficiency issues. Flex separates its design into two independent parts. FlexW periodically monitors per-VM actual CPU allocation and adjusts VM weights to approximate the ideal fair allocation. FlexS eliminates excessive busy-waiting in guest OSes by detecting spinning vCPUs from hardware-level branch instruction metrics. FlexS stops busy-waiting vCPUs and opportunistically looks for vCPUs with useful work on other CPUs.

Experiments with Xen and various parallel workloads show that Flex is able to achieve allocations close to the ideal fair allocation and realizes a certain level of differentiation. Flex also shows good performance with parallel workloads using different synchronization methods.

Acknowledgements

We are grateful to the anonymous reviewers for their constructive comments. This research was supported in part by the U.S. National Science Foundation under grants CNS-1320122, CNS-1217979 and CNS-0844983.

References

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] AMD Corporation. *AMD64 architecture programmers manual volume 2: System programming*. 2010.

[3] M. B. Anwer, A. Nayak, N. Feamster, and L. Liu. Network i/o fairness in virtual machines. In *Proc. of VISA*, 2010.

[4] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst.*, 19(3), 2001.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarkssummary and preliminary results. In *Proc. of SC*, 1991.

[6] K. based virtual machine. http://www.linux-kvm.org/.

[7] K. Chakraborty, P. M. Wells, and G. S. Sohi. Supporting overcommitted virtual machines through hardware spin detection. *IEEE Trans. Parallel Distrib. Syst.*, 23(2), Feb. 2012.

[8] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proc. of SIGMETRICS*, 1996.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[11] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. In *Proc. of ASPLOS*, 2013.

[12] P. Lama and X. Zhou. NINEPIN: Non-invasive and energy efficient performance isolation in virtualized servers. In *Proc. of DSN*, 2012.

[13] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of i/o for gang scheduled workloads. In *Proc. of IPPS*, 1997.

[14] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proc. of PPoPP*, 2009.

[15] P. B. Menage. Adding generic process containers to the linux kernel. In *Proc. of OLS*, 2010.

[16] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10), 2001.

[17] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proc. of EuroSys*, 2010.

[18] R. Nikolaev and G. Back. Perfctr-xen: a framework for performance counter virtualization. In *Proc. of VEE*, 2011.

[19] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proc. of VEE*, 2008.

[20] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of ICDCS*, 1982.

[21] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3), 1993.

[22] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proc. of HPCA*, 2013.

[23] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of OSDI*, 2012.

[24] P. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic coscheduling on workstation clusters. In *Proc. of JSSPP*, 1998.

[25] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not vcpus. In *Proc. of APSys*, 2013.

[26] SPEC Java Server Benchmark. http://www.spec.org/jbb2005/.

[27] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proc. of EuroSys*, 2011.

[28] The Apache Mahout machine learning library. http://mahout.apache.org/.

[29] The CPU Scheduler in VMware vSphere 5.1. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[30] The Princeton Application Repository for Shared-Memory Computers (PARSEC) . http://parsec.cs.princeton.edu/.

[31] The SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.

[32] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of VM*, 2004.

[33] VMware. http://www.vmware.com.

[34] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. of HPDC*, 2011.

[35] Windows Azure Open Cloud Platform. http://www.windowsazure.com.

[36] Xen. http://www.xen.org/.

[37] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proc. of HPDC*, 2012.

[38] C. Xu, S. Gamage, H. Lu, R. R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proc. of USENIX ATC*, 2013.