Towards Efficient Work-stealing in Virtualized Environments

Yaqiong Peng, Song Wu, Hai Jin
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
{*pengyaqiong, wusong, hjin*}*@hust.edu.cn*

***Abstract*—Nowadays, work-stealing, as a common user-level task scheduler for managing and scheduling tasks among worker threads, has been widely adopted in multithreaded applications. With work-stealing, worker threads attempt to steal tasks from other threads' queue when they run out of their own tasks. Though work-stealing based applications can achieve good performance due to the dynamic load balancing, these steal attempting operations may frequently fail especially when available tasks are scarce, thus wasting CPU resources of busy worker threads and consequently making work-stealing less efficient in competitive environments, such as traditional multiprogrammed and virtualized environments. Although there are some optimizations for reducing the cost of steal-attempting threads by having such threads yield their computing resources in traditional multiprogrammed environments, it is more challenging to enhance the efficiency of work-stealing in virtualized environments due to the semantic gap between the virtual machine monitor (VMM) and virtual machines (VMs).**

In this paper, we first analyze the challenges of enhancing the efficiency of work-stealing in virtualized environments, and then propose Robinhood, a scheduling framework that reduces the cost of virtual CPUs (vCPUs) running steal-attempting threads and the scheduling delay of vCPUs running busy threads. Different from traditional scheduling methods, if the steal attempting failure occurs, Robinhood can supply the CPU time of vCPUs running steal-attempting threads to their sibling vCPUs running busy threads, which can not only improve the CPU resource utilization but also guarantee better fairness among different VMs sharing the same physical node. We implement Robinhood based on BWS, Linux and Xen. Our evaluation with various benchmarks demonstrates that Robinhood paves a way to efficiently run work-stealing based applications in virtualized platform. It can reduce up to 64% and 30% execution time of work-stealing benchmarks compared to Cilk++ and BWS respectively, and outperform credit scheduler and co-scheduling for average system throughput by 1.91× and 1.3× respectively, while guaranteeing the performance fairness among applications in virtualized environments.

***Keywords*-virtualization; work-stealing; multicore; parallel program optimization;**

I. Introduction

A. Motivation

Nowadays, work-stealing, as a common user-level task scheduler for managing and scheduling tasks among worker threads, has been widely adopted in multithreaded applications, such as "document processing, business intelligence, games and game servers, CAD/CAE tools, media processing, and web search engines" [1]. In work stealing, each worker thread maintains a task queue for ready tasks. When a worker thread runs out of its local tasks, it turns into a *thief* and attempts to steal some tasks from a randomly selected worker thread's (*victim*) queue. If there are available tasks in the victim's queue, the thief can successfully steal some tasks from the victim and then run the tasks. Otherwise, the thief randomly selects another victim. Due to the dynamic load balancing, work-stealing has been advocated as powerful and effective approach to achieving good performance of parallel applications [2], [3] in dedicated environments, where applications run with one thread per core. However, previous research works on work-stealing have demonstrated that the unsuccessful steals performed by thieves may waste computing resources in competitive environments [4], [1], such as traditional multiprogrammed environments and virtualized environments.

To mitigate this problem of work-stealing in traditional multiprogrammed environments, Cilk++ and Intel TBB implement a yielding mechanism, namely having a steal-attempting thread yield its core if the steal is unsuccessful and there are other ready threads on the core. Ding et al. implement a better yielding mechanism in a *balanced work-stealing scheduler* (BWS), which allows a steal-attempting thread to yield its core to a preempted, busy thread in the same application [1]. Moreover, BWS "monitors and controls the number of awake, steal-attempting threads for each application" [1]. To the best of our knowledge, BWS is the best available work-stealing scheduler for traditional multiprogrammed environments. In the following, we look into the issues of work-stealing in virtualized environments.

With the prevalence of cloud computing, virtualized data centers, like Amazon EC2 [5], have increasingly become the underlying infrastructures to host work-stealing based applications. We assume that a work-stealing based application is running alone on a VM like many previous research works [6], [7], [8], and its number of threads is not more than the number of vCPUs (common configuration

in work-stealing). From the perspective of the guest OS, the application runs with one thread per vCPU and thus its threads do not compete for a vCPU. A steal-attempting thread still occupies its vCPU after using the yielding mechanism in Cilk++ or BWS because the thread is the only ready thread on its vCPU and none preempted busy threads exist in the same application. Due to the semantic gap between VMM and VMs, the VMM does not know whether vCPUs are running steal-attempting threads or busy threads. As a result, vCPUs running steal-attempting threads can waste CPU resources, which could otherwise be used by vCPUs running busy threads. Inspired by that BWS does in traditional multiprogrammed environments, can we have a vCPU running a steal-attempting thread yield its physical CPU (pCPU) to a sibling vCPU running a busy thread? To achieve this goal, we must overcome the semantic gap challenge in virtualized environments. Moreover, merely extending the yielding mechanism in BWS to virtualized environments may incur the vCPU stacking issue [9], as analyzed in Section II.

B. Our Approach

In this paper, we present Robinhood, a vCPU scheduling framework that enhances the efficiency of work-stealing in virtualized environments. Robinhood consists of two components: **Identifying Acceleration Candidate** and **vCPU Acceleration**. The first component relies on the work-stealing runtime and guest OS to identify vCPUs running *poor workers* (see the definition in Section II.C) as acceleration candidates with a new hypercall, which bridges the semantic gap between VMM and VMs. The second component is used to accelerate the candidate vCPUs by supplying them with the CPU time of vCPUs running steal-attempting threads. At a high-level, if a vCPU *A* identifies a preempted vCPU *B* as acceleration candidate and *A* has remaining time slice, Robinhood suspends *A* and migrates *B* to the pCPU of *A*. And then *B* consumes the remaining time slice of *A*. In order to avoid vCPUs from the same VM stack on one pCPU, Robinhood records the relative position of *B* in its original run queue before migration, and pushes *B* back to the position when the time slice offered by *A* is over.

C. Evaluation and Contributions

We implement Robinhood based on BWS [1], Linux and Xen, and compare its effectiveness to the state-of-the-art schemes including Cilk++ [10] under Credit scheduler of Xen [11] and co-scheduling [12] respectively, and BWS [1] under balance scheduling [9]. Our evaluation with various work-stealing and non-work-stealing benchmarks demonstrates that the advantages of Robinhood include (1) reducing up to 64% and 30% execution time of work-stealing benchmarks compared to Cilk++ and BWS in single-VM scenario, respectively, and (2) outperforming other schemes for average system throughput by up to $1.91\times$, without sacrificing the performance fairness among benchmarks in multiple-VM scenario.

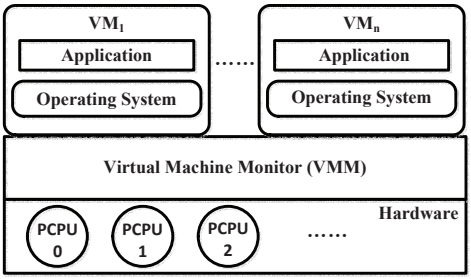


Figure 1: General System Virtualization Architecture

In summary, this paper makes the following contributions:

- We analyze the challenges of enhancing the efficiency of work-stealing in virtualized environments.
- We present a novel vCPU scheduling framework named Robinhood with the goal to reduce the cost of vCPUs running steal-attempting threads and the scheduling delay of vCPUs running busy threads.
- We rigorously implement a prototype of Robinhood based on BWS, Xen and Linux.

D. Outline

The rest of the paper is organized as follows. Section II overviews the background on virtualization and work-stealing software systems, and then looks into the challenges of work-stealing in virtualized environments. Section III and Section IV describe the design and implementation of Robinhood, respectively. Section V provides a comprehensive evaluation of Robinhood. Section VI discusses the related work, and Section VII concludes the paper.

II. Background and Motivation

In this section, we first discuss the basics of system virtualization and work-stealing software systems. Then, we look into the challenges of work-stealing in virtualized environments.

A. Virtualization

Figure 1 shows the general system virtualization architecture, which introduces an additional layer (hypervisor or VMM) between guest operating systems and the underlying hardware. The VMM manages the underlying hardware resources and exports them to VMs running on them. The virtualization technology is widely used to build cloud data centers. Virtualized data centers, like Amazon EC2 [5], provide services for customers to lease VMs and run their applications on the VMs.

B. Work-stealing Software Systems

With the prevalence of multicore processors, many applications use parallel programming models to utilize the

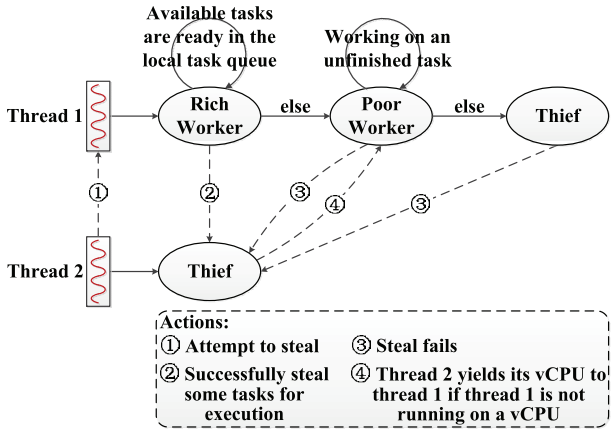


Figure 2: A scenario of BWS in virtualized environments

abundant computing resources. Work-stealing has been integrated into most of parallel programming models because of its effectiveness in managing and scheduling tasks among worker threads. In work stealing, each worker thread maintains a task queue for ready tasks. When a worker thread runs out of its local tasks, it turns into a *thief* with the goal to steal some tasks from a randomly selected worker thread's (*victim*) queue. If there are available tasks in the victim's queue, the thief can successfully dequeue some tasks from the victim's queue and run the tasks. Otherwise, the thief randomly selects another victim. The most popular work-stealing software systems are Cilk [13], [14], Cilk++ [10] and Intel Threading Building Blocks (TBB) [15].

Previous research works on work-stealing have demonstrated that the unsuccessful steals performed by thieves may waste computing resources in traditional multiprogrammed environments [4], [1]. To the best of our knowledge, BWS is the state-of-the-art work-stealing scheduler for mitigating the cost of steal-attempting threads [1], which is implemented on the top of Cilk++. In BWS, when a thread fails to conduct a steal operation, it yields its core to a preempted, busy thread in the same application. Moreover, a thread falls into sleep after a certain times of unsuccessful steal attempts. In the following, we analyze the problem of BWS in virtualized environments, and the challenges to solve the problem.

C. Challenges for Work-stealing in Virtualized Environments

We provide an intuitive scenario to analyze the problem of BWS for virtualized environments, which is shown in Figure 2. For the convenience of our analysis, we first define the following roles of threads in work-stealing:

- **Thief:** Attempting to steal tasks from other threads
- **Poor worker:** Working on an unfinished task and the local task queue is empty
- **Rich worker:** Working on an unfinished task and available tasks are ready in the local task queue

A 8-threaded application is running on a 8-vCPU VM. The application is developed by BWS. Thread 2 in the application runs out of its local tasks, and turns into a *thief*. In order to keep itself busy, thread 2 randomly selects thread 1 as victim, and attempts to steal some tasks from the victim. Currently, thread 1 may be a *rich worker*, *poor worker*, or *thief*. Figure 2 shows the actions of thread 2 for different roles of thread 1.

Thread 2 can successfully steal some tasks from thread 1 only if thread 1 is a rich worker. Besides, if thread 1 is a poor worker not running on a vCPU, thread 2 will yield its vCPU to thread 1, which also contributes to the progress of its application. Assuming that thread 1 is running on a preempted vCPU, thread 2 can not immediately yield its vCPU after failing to steal tasks from thread 1. The reason is that thread 1 is making progress from the perspective of the guest OS running in the VM. In fact, the vCPU running thread 1 is preempted by the scheduler in VMM. In this situation, thread 2 will attempt to steal tasks from other threads, and the steal attempts may be continuously unsuccessful. As a result, vCPUs running thread 2 can waste CPU resources, which could otherwise be used by the vCPU running thread 1.

How to enable steal-attempting threads to yield their computing resources to busy threads from the same applications in virtualized environments? We can easily come up with two possible approaches: (1) needing the VMM to disclose whether a vCPU is currently running on a pCPU to the guest OS, and having steal-attempting threads yield their vCPUs to busy threads running on preempted vCPUs, and (2) needing the work-stealing runtime and guest OS to disclose whether a vCPU is running a steal-attempting thread or busy thread to the VMM, and having vCPUs running steal-attempting threads yield their pCPUs to vCPUs running busy threads.

The main issue of the first approach is that it is very hard for the guest OS to ensure that a yielded vCPU has remaining time slice. In other words, when a vCPU is about to be yielded to a busy thread, the vCPU may be preempted by the scheduler in VMM and thus the busy thread can not make progress. Therefore, we adopt the second approach.

In order to adopt the second approach, we face two challenges. The first challenge is how to expose the knowledge of vCPUs running steal-attempting threads or busy threads to VMM because VMM has little knowledge on threads in the guest OS. Moreover, it can incur the vCPU stacking (sibling vCPUs contend for a pCPU) by having vCPUs running steal-attempting threads yield their pCPUs to preempted vCPUs running busy threads from the same VMs. However, previous literature has identified the negative impact of vCPU stacking on the performance of parallel programs in virtualized environments [9]. Therefore, the second challenge is how to avoid vCPU stacking while adopting the second approach.

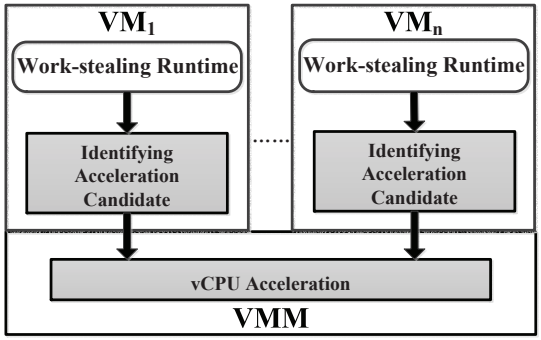


Figure 3: Architecture of Robinhood

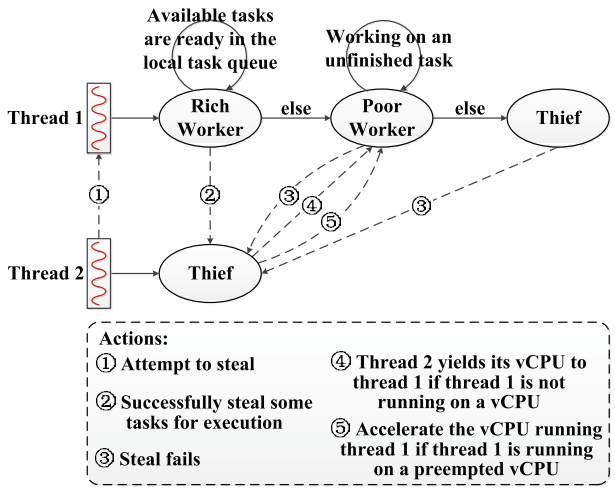


Figure 4: A scenario of Robinhood in virtualized environments

To overcome the above challenges, we propose a new scheduling framework named Robinhood, which will be presented in the next section.

III. The Design of Robinhood

In this section, we describe the design of Robinhood. We begin with the overview of Robinhood, followed by introducing the details.

A. Overview

Figure 3 overviews the architecture of Robinhood, including the following two components:

- **Identifying Acceleration Candidate:** Robinhood relies on the work-stealing runtime and guest OS to identify vCPUs running poor workers as acceleration candidates. With Robinhood, once a thief finds that the selected victim is a poor worker, it traps into the guest OS kernel to check the running status of the worker. If the worker is currently running on a vCPU v, the thief initiates a hypercall down to the VMM with the vCPU ID of v as the argument. Once receiving the hypercall, the VMM can find the data structure representing the vCPU v by its ID, and then initiates the second component to make acceleration decisions. The first component is used to bridge the semantic gap between VMM and VMs.
- **vCPU Acceleration:** If a thief identifies a vCPU as acceleration candidate, we refer to the vCPU running the thief as the yielder for the candidate vCPU. Robinhood checks the running status of the candidate vCPU and its yielder. If the candidate vCPU is preempted by the scheduler in VMM and its yielder has remaining time slice, Robinhood accelerates the candidate vCPU by using a first-migrate-then-push-back mechanism to supply the remaining time slice of the yielder to the candidate vCPU.

Figure 4 shows the scenario of Robinhood similar to that in Figure 2. Different from BWS, Robinhood can further contribute to the progress of work-stealing applications by accelerating vCPUs running poor workers.

In the following subsections, we present the details of vCPU acceleration component, including its design principles and the first-migrate-then-push-back acceleration policy.

B. Design Principles of vCPU Acceleration

Robinhood accelerates a candidate vCPU depending on the running status of the candidate and the remaining time slice of its yielder. Taking Figure 4 as an example again, if the vCPU running thread 1 is currently running on a pCPU, it has no need to migrate the vCPU to the pCPU of thread 2 for acceleration. Once Robinhood determines not to accelerate the candidate, thread 2 will find another victim to steal tasks, which follows the original path of the work-stealing runtime.

If Robinhood determines to accelerate a candidate vCPU, the vCPU will be migrated to the pCPU of its yielder. Then the scheduler in VMM de-schedules the vCPU's yielder and schedules the vCPU immediately. How much CPU time should be allocated to the candidate vCPU for this acceleration? Generally, most of the VMMs adopt a proportional share scheduling strategy such as Xen and KVM. This strategy allocates CPU time in proportion to the number of shares (weights) that VMs have been assigned. Therefore, the CPU time allocated to a candidate vCPU is equal to the remaining time slice of its yielder by Robinhood, which abides the inter-VM fairness.

Moreover, previous literature has identified the negative impact of vCPU stacking on the performance of parallel programs in virtualized environments [9]. In order to avoid vCPU stacking, Robinhood integrates balance scheduling into its design. For the acceleration path, Robinhood records the relative position of the candidate vCPU in its original run queue before migration, and pushes the vCPU back to the position when its allocated time slice is over so that no stacking could happen. This acceleration policy is called first-migrate-then-push-back.

C. First-Migrate-Then-Push-Back Acceleration Policy

Algorithm 1 First-Migrate-Then-Push-Back Scheduling

Input: *curr*: the current vCPU running on the pCPU where the scheduler resides; *runq*: the run queue of the pCPU where the scheduler resides; *default_time_slice*: the default time slice in the scheduler
Output: scheduling decision

```
1:  /* The CPU time consumed by an accelerated vCPU is paid by its yielder, and the accelerated vCPU is not inserted into runq when its assigned time slice is over */
2:  if curr → yielder != NULL then
3:      burn_runtime(curr → yielder, runtime(curr))
4:  else
5:      burn_runtime(curr, runtime(curr))
6:      insert(curr)
7:  end if
8:  time_slice = get_time_slice(curr)
9:  remaining_time_slice = time_slice − runtime(curr)
10: /* If the current vCPU determines to yield its pCPU to an accelerated candidate and has remaining time slice, the candidate is migrated to this pCPU without being removed from its original run queue */
11: if curr → acct_candidate != NULL
    and remaining_time_slice> 0 then
12:     acct_candidate = curr → acct_candidate
13:     original_cpu = get_pcpu(acct_candidate)
14:     set_original_pcpu(acct_candidate, original_cpu)
15:     set_pcpu(acct_candidate, get_pcpu(curr))
16:     set_time_slice(acct_candidate, remaining_time_slice)
17:     acct_candidate → yielder = curr
18:     next = acct_candidate
19: else
20:     next points to the non-accelerated vCPU that is closest to the head of runq
21:     remove(next)
22:     set_time_slice(next, default_time_slice)
23: end if
24: curr → acct_candidate = NULL
25: /* Push the accelerated vCPU back to its original run queue */
26: if curr → yielder != NULL then
27:     set_pcpu(curr, get_original_pcpu(curr))
28:     curr → yielder = NULL
29: end if
30: Context switch to next
```

Algorithm 1 shows the detail of the first-migrate-then-push-back policy, which resides in the main schedule function in the VMM. We add two variables to the per-vCPU data structure: *acct_candidate* and *yielder*. When the current vCPU *curr* identifies a vCPU *v* as acceleration candidate and *v* is preempted by the scheduler in VMM, the variable *acct_candidate* in *curr* points to *v*, and then *curr* raises a soft IRQ interrupt on its pCPU and forces a call to the main schedule function.

Generally, VMM monitors the CPU shares of each vCPU in certain forms. For example in the Xen, the CPU shares of each vCPU is monitored in *credits*. As a vCPU runs, it consumes its CPU shares. When a vCPU runs out of its CPU shares, it indicates that the vCPU has exceeded its fair share of CPU time. With the first-migrate-then-push-back policy, if a vCPU is supplied with extra CPU time by its yielder, the vCPU consumes the CPU shares of the yielder (lines 2-3). If *curr* has remaining time slice, *v* is migrated to the pCPU of *curr* without being removed from its original run queue (lines 11-18) in order to retain the relative position of *v* in its original run queue. Otherwise, the scheduler will run the non-accelerated vCPU that is closest to the head of runq. If the yielder value of a vCPU is NULL, it means that it is a non-accelerated vCPU. Otherwise, the vCPU is accelerated on another pCPU and can not be scheduled on its original pCPU until the acceleration is over. When the acceleration is over, an accelerated vCPU is pushed back to its original run queue by setting the value of its yielder to NULL (lines 26-29).

IV. The Implementation of Robinhood

We have implemented Robinhood based on BWS, Linux and Xen. We choose BWS because it is the best available implementation of work-stealing scheduler for traditional multiprogrammed environments, and BWS is based on Cilk++. The latest release of BWS is available on the website [16]. We choose Linux and Xen because of their broad acceptance and the availability of their opensource code.

In the following, we will describe how to implement the two components of Robinhood in detail.

A. How to Identify Acceleration Candidate

Robinhood needs the work-stealing runtime and guest OS to disclose whether a thread is a poor worker running on a preempted vCPU. BWS provides a new system call in Linux kernel, which allows the calling thread (*yielder*) to yield its core to a designated thread (*yieldee*) in the same application only if the yieldee is currently preempted by the scheduler in OS. We modify the system call in Robinhood. In the modified version, if the yieldee is currently preempted by the scheduler in guest OS, the yielder yields its vCPU to the yieldee as the original path in BWS. Otherwise, the yieldee's vCPU is identified as acceleration candidate. Then the guest OS initiates a new hypercall down to the VMM with the yieldee's vCPU ID as the argument, and then uses the argument to find the vCPU data structure (called *vcpu* in Xen) representing the yieldee's vCPU for making acceleration decisions. The hypercall is implemented by us on the top of Xen.

B. How to Implement vCPU Acceleration

We implement the acceleration component based on the credit scheduler of Xen. Under credit scheduler, each pCPU maintains a local run queue for runnable vCPUs, and the run queue is sorted by vCPU priority (*boost*, *under*, *over* and *idle* from the high to low). When inserting a vCPU into a run queue, it is placed behind all other vCPUs of

equal priority to it. The credit scheduler is a proportional fair share schedulers. Each VM is assigned a *weight*, and vCPUs of each VM are distributed a parameter called *credits* in proportion to the weight of the VM at the end of each accounting period (default 30ms). As a vCPU runs, it consumes its credits. When a vCPU runs out of its credits, it indicates that the vCPU has exceeded its fair share of CPU time and its priority is set to *over*. If a vCPU still has credits, its priority is set to *under*. On each pCPU, the head of the run queue is selected as the next vCPU to run at every scheduling decision.

Based on the credit scheduler, when Robinhood determines to accelerate a vCPU V_1, Robinhood forces a call to the main scheduler function in Xen. Instead of selecting the head of the run queue to run next, Robinhood sets V_1 to the next vCPU to run, and sets the time slice of V_1 equalling to the remaining time slice of V_1's yielder. During the acceleration process, V_1 consumes the credits of its yielder. In order to save the information of yielders and acceleration candidates, we add two member variables: *acct_candidate* and *yielder* to the data structure *vcpu* in Xen.

V. Evaluation

With the implementation of Robinhood, we carry out our experiments on a machine consisting of one eight-core 2.6GHz Intel Xeon E5-2670 chip with hyper-threading disable. We use Xen-4.2.1 as the hypervisor and Redhat Enterprise Linux 6.2 with the kernel version 3.9.3. In this section, we first introduce characteristics of the selected benchmarks and our experimental methodology, then present the experimental results.

A. Benchmarks

Table I describes the benchmarks we use in our experiments. All the benchmarks except LOOP are developed with Cilk++ and provided by authors of [1]. BWS and Robinhood completely retain the same *application programming interfaces* (APIs) of Cilk++. LOOP is used as a representative of non-work-stealing multi-threaded programs. We match the number of threads in these benchmarks with the number of vCPUs in the VMs like many previous research papers [6], [8].

B. Experimental Methodology

We compare Robinhood with the following related schemes:

- *Baseline*: Cilk++ under the default Credit scheduler in Xen hypervisor.
- *CS+Cilk++*: Cilk++ under co-scheduling which schedules all the vCPUs of a VM simultaneously.
- *BWS*: The best available implementation of work-stealing scheduler [1] for multiprogrammed environments.

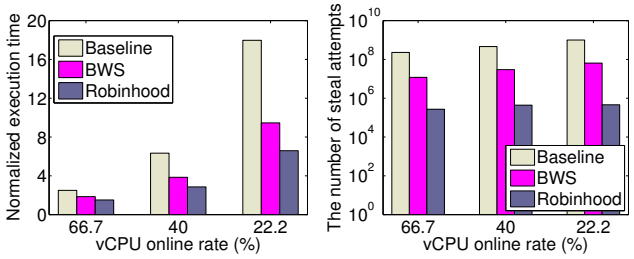


Figure 5: Normalized execution time of CLIP and the number of steal attempts

To study performance comprehensively, we first test the performance of work-stealing benchmarks on a single VM with different vCPU *online rate*, and compare Robinhood with the baseline and BWS. The vCPU online rate indicates the percentage of time of vCPUs in a VM being mapped to pCPUs. Then we test the performance of work-stealing and non-work-stealing benchmarks running simultaneously on multiple VMs under baseline, CS+Cilk++, BWS and Robinhood in terms of fairness and throughput. When testing multiple VMs, we evaluate BWS along with balance scheduling in order to make the comparison with Robinhood fair. Moreover, we also compare Robinhood with another alternative approach, which haves a vCPU running steal-attempting thread yield its pCPU to a sibling vCPU running busy thread without pushing back the latter to its original pCPU.

C. Testing a single VM

In this testing scenario, each of work-stealing benchmarks is run on VM *vm1*, which is configured with 4 vCPUs and 4096MB memory. We borrow the experimental approach from [6] to adjust the vCPU online rate of *vm1*. In order to provide an intuitive comparison, we normalize the execution time of a benchmark running on a VM with a vCPU online rate ($<$100%) to the execution time of the same benchmark running on the same VM under baseline when the vCPU online rate is 100%.

The execution time of CLIP and the number of steal attempts under baseline, BWS and Robinhood are shown as Figure 5. We can observe that Robinhood can significantly reduce the number of steal attempts in CLIP compared to Cilk++ and BWS when the vCPU online rate decreases. As a result, Robinhood reduces up to 55% and 26% execution time of CLIP compared to Cilk++ and BWS, respectively.

We also run other work-stealing benchmarks on *vm1* while the configuration is the same as CLIP. As depicted in Figure 6, Robinhood outperforms the baseline and BWS in all aspects while varying benchmarks and the vCPU online rate. The lower the vCPU online rate is, the more apparent the benefit of Robinhood is. To be more specific,

Table I: Benchmarks

Benchmarks	Description
CG	Conjugate gradient, size B
CLIP	Parallelized computational geometry algorithm on a large number of polygon pairs
MI	Matrix inversion algorithm on a 500×500 matrix
LOOP	A micro-benchmark that creates 8 threads, and each thread performs a busy loop until the main thread notifies them to finish

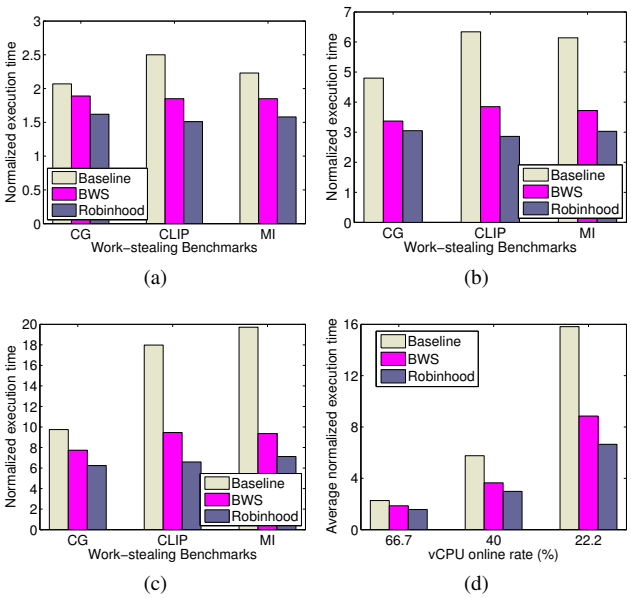


Figure 6: Normalized execution time of work-stealing benchmarks in *vm1*, where the vCPU online rate is: (a) 66.7%, (b) 40%, and (c) 22.2%; the average normalized execution time of all benchmarks is shown in (d)

Robinhood saves up to 56% and 25% average execution time of work-stealing benchmarks compared to Cilk++ and BWS, respectively.

D. Testing multiple VMs

In this testing scenario, we co-run two VMs, each of which runs a benchmark from Table 1 and is configured with 8 vCPUs and 4096MB memory. As different benchmarks have different execution times, we run each benchmark repeatedly in all experiments, so that their executions can be fully overlapped.

In order to study the behaviour of co-running benchmarks under different schemes, we first give some metrics.

Assuming the baseline solo-execution time of the application *App(n)* running on the *nth* VM is $T_s(n)$ and its execution time in a co-running is $T_c(n)$, then the slowdown of *App(n)* in the co-running is defined as follows:

$$\textit{Slowdown(n)} = \frac{T_c(n)}{T_s(n)} \tag{1}$$

The lower slowdown indicates the better performance.

An ideal fair scheduling strategy can guarantee that the CPU time consumed by a VM *VM(n)* is strictly in proportional to its weight. Meanwhile, the VMs in our system have the same configuration. Therefore, each application running on a VM should suffer from the same slowdown if the underlying scheduling vCPU scheduling strategy are ideally fair. Then the system unfairness is defined as follows:

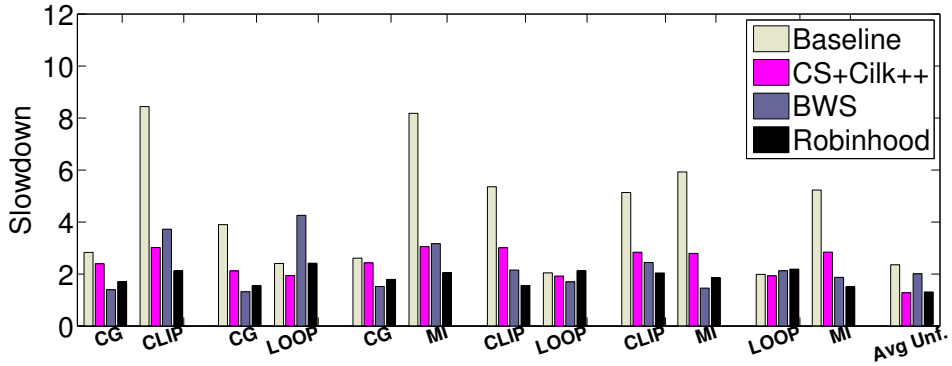
$$Unfairness = \frac{\max_1^n\{Slowdown(n)|\}}{\min_1^n\{Slowdown(n)|\}} \tag{2}$$

We use the weighted speedup [17] to measure system throughput, which is the sum of the speedups of co-running applications:

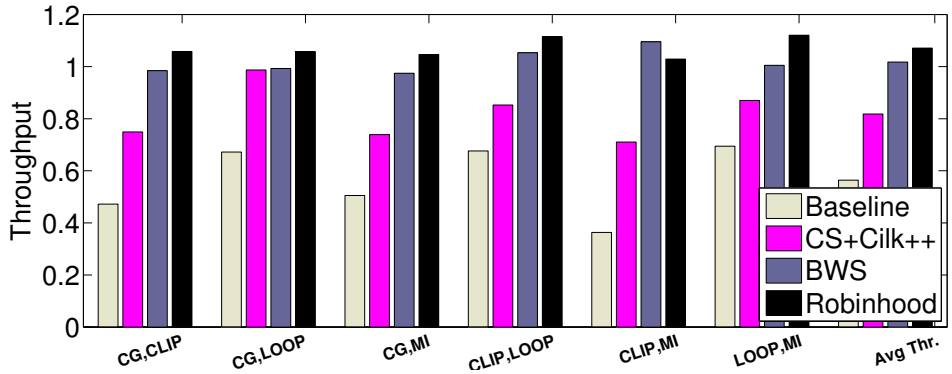
$$\textit{Throughput} = \sum_1^n \frac{T_s(n)}{T_c(n)} \tag{3}$$

Figure 7 compares the slowdowns of the benchmarks in each co-running for the baseline, CS+Cilk++, BWS, and Robinhood, and the corresponding unfairness and system throughput. In Figure 7(a), the last four bars show the average unfairness. As shown in Figure 7(a), with the baseline, when two benchmarks co-run, one benchmark may be slowed down by a much larger degree than the other benchmark. Usually, the one that suffers from more serious performance degradation in single-VM scenario, shows a larger slowdown. For example, under the baseline, when CG and CLIP co-run, the slowdown of CLIP is as much as 8.4, while the slowdown of CG is only 2.8. The reason is that the threads in the higher slowdown benchmarks are more likely to run out of tasks and turn into steal-attempting threads wasting CPU time with the baseline. As the CPU time allocated to each VM is in proportion to the weight of that VM, vCPUs running the steal-attempting threads impede the execution of vCPUs running busy thread from the same VMs. LOOP almost suffers from the expected slowdown in each co-running because it is a non-work-stealing benchmark. On average, the unfairness under the baseline is 2.36.

Co-scheduling is to time-slice all the cores so that each application is granted a dedicated use of the cores during its scheduling quanta. Therefore, with CS+Cilk++, the slowdown of each application is closely to the number of co-running applications (2 in our experiments), resulting in the best performance fairness between benchmarks running simultaneously on two VMs (as shown in Figure 7(a)), respectively. However, with CS+Cilk++, vCPUs running steal-



(a) Slowdown and unfairness



(b) Throughput

Figure 7: Two benchmarks run simultaneously on two VMs, respectively

attempting threads still waste computing resources during their scheduling quanta, resulting in a low system throughput as shown in Figure 7(b). On average, the unfairness under CS+Cilk++ is 1.28.

BWS monitors and controls the number of awake, steal-attempting threads, and thus the vCPUs running such threads are also timely blocked. Therefore, the original BWS can also reduce the cost of vCPUs running steal-attempting threads in virtualized environments to some extent. However, with BWS, the priority of blocked vCPUs will be boosted when the blocked vCPUs are waken up, and the scheduler in VMM immediately schedule these vCPUs. As a result, if a VM runs a frequently sleeping application, it may have negative impact on the performance of applications running on other VMs. For example, under BWS, when CG and LOOP co-run, the slowdown of LOOP is as much as 4.26, while the slowdown of CG is only 1.32. On average, the unfairness under BWS is 2.01.

Robinhood both reduces the cost of vCPUs running steal-attempting threads and scheduling delay of vCPUs running busy threads by supplying the CPU time of the former to the latter. When vCPUs running steal-attempting threads yield their pCPUs to vCPUs running busy threads, they do not fall into sleeping. As a result, Robinhood can significantly reduce the difference between slowdowns of co-running benchmarks compared to the baseline and BWS, and perform closely to CS+Cilk++ in terms of fairness. On average, the unfairness under Robinhood is 1.31.

Figure 7(b) shows throughputs of different co-running benchmarks with different schemes. Robinhhood outperforms other schemes in terms of throughput for all cases except the co-running of CLIP and MI under BWS. However, when CLIP and MI co-run under BWS, the system unfairness is 1.68. Robinhood can reduce the unfairness to 1.1 (improve the fairness by 53%) while only reducing the system throughput by only 6% compared to BWS.

E. Experiments with Alternative Approach

We also extend the yielding mechanism in BWS to Xen. With this method, a vCPU running steal-attempting thread yields its pCPU to a sibling vCPU running busy thread without pushing back the latter to its original pCPU. We call this method Yield_to, and Yield_to uses the first component in Robinhood to bridge the semantic gap between VMM and VMs.

In Figure 8, we only plot the result of CLIP and LOOP. The other testing cases have similar figures. Figure 8(a) shows the execution time of CLIP running on *vm1* with different vCPU online rate. We can observe that Robinhood can significantly improve the performance of CLIP by 1.58×

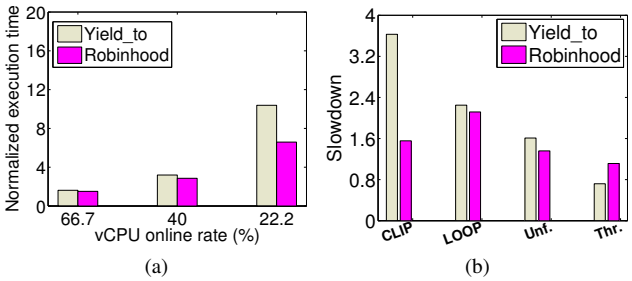


Figure 8: Comparing Robinhood to Yield_to under two scenarios: (a) CLIP runs on *vm1* with different vCPU online rate, and (b) CLIP and LOOP run simultaneously on two VMs, respectively

compared to Yield_to, when the vCPU online rate decreases to 22.2%. Figure 8(b) shows slowdown of CLIP and LOOP when they run simultaneously on two VMs, respectively. The last four bars show the unfairness between CLIP and LOOP, and their overall throughput, respectively. We can observe that Robinhood outperforms Yield_to in all aspects such as improving the performance of CLIP by 2.33×, fairness by 1.18×, and throughput by 1.55×. This is because Yield_to can result in the vCPU stacking issue for VMs running work-stealing applications.

VI. Related Work

Our work is related to the research in system support for vCPU scheduling and work-stealing software systems. We briefly discuss the most related work in turn.

A. System Support for vCPU Scheduling

Virtualization technology is a good way to improve the usage of the hardware while decreasing the cost of the power in cloud data centers. Different from the traditional system software stack, an additional layer (hypervisor or VMM) is added between guest OS and the underlying hardware. Currently, the most popular system virtualization includes VMware [18], Xen [11], and KVM [19].

One vCPU scheduling method is to schedule vCPUs of a VM asynchronously. This method simplifies the implementation of the vCPU scheduling in the VMM and can improve system throughput. Therefore, it is widely adopted in the implementations of VMMs such as Xen and KVM. Another vCPU scheduling method, namely co-scheduling, is to schedule and de-schedule vCPUs of a VM synchronously. Co-scheduling can alleviate the performance degradation of parallel applications running on SMP VMs, and it is used in previous work [12] and VMware [18]. The limitation of co-scheduling is that the number of the vCPUs should be no more than the number of the pCPUs [12], and CPU fragmentation occurs when the system load is imbalanced [9]. To solve this issue, many researchers propose demand-based co-scheduling [6], [7], which dynamically initiates co-scheduling only for synchronizing vCPUs. This paper focuses on the cases where the system load is balanced. Therefore, we only evaluate co-scheduling as a representative of co-scheduling based approaches. As far as we know, no existing vCPU scheduling policies are designed with the full consideration of the features of work-stealing based applications.

B. Work-stealing Software Systems

Work-stealing is widely used in multithreaded applications because of their effectiveness in managing and scheduling tasks among workers. Over the last few years, there are many studies focusing on enhancing its multiprogrammed performance [20], [4], [1] in traditional environments. The common idea of these studies is to have steal-attempting threads yield their computing resources. Inspired by this idea, we present Robinhood with the goal to enhance the efficiency of work-stealing based applications in virtualized environments. Compared to work-stealing in traditional environments, Robinhood has to face some new challenges, such as the semantic gap between VMM and VMs, and avoiding the vCPU stacking issue.

There are also many studies for enhancing other features of work-stealing algorithms, such as improving data locality [21], and reducing task creation and scheduling overheads [22], [23]. These studies do not address competitive issues for underlying computing resources, which are orthogonal to Robinhood.

VII. Conclusions and Future Work

With the prevalence of virtualization and multicore processors, SMP VMs hosting multithreaded applications have been common cases in cloud data centers. Work-stealing, as an effective approach for managing and scheduling tasks among worker threads in multithreaded applications, may suffer from serious performance degradation in virtualized environments because the execution of vCPUs running steal-attempting threads may waste their attained CPU time and even impede the execution of vCPUs running busy threads.

This paper presents Robinhood, a scheduling framework that enhances the efficiency of work-stealing in virtualized environments. Different from traditional scheduling methods, if the steal attempting failure occurs, Robinhood can supply the remaining time slices of vCPUs running steal-attempting threads to their sibling vCPUs running busy threads, which can not only improve the CPU resource utilization but also guarantee better fairness among different virtual machines sharing the same physical node. Robinhood has been implemented on the top of BWS, Linux and Xen. Our evaluation with various benchmarks demonstrates that Robinhood paves a way to efficiently run work-stealing based applications in virtualized platform. It can reduce up to 64% and 30% execution time of work-stealing benchmarks compared to Cilk++ and BWS respectively, and

outperform credit scheduler and co-scheduling for average system throughput by 1.91× and 1.3× respectively, while guaranteeing the performance fairness among applications in virtualized environments.

In future work, we plan to enhance the support of Robinhood for NUMA multicore systems. A possible approach is to partition vCPUs of each VM into different groups. The number of vCPU groups is equal to the number of NUMA domains on a multicore machine, and each group belongs to a NUMA domain. The vCPU acceleration is only allowed among vCPUs belonging to the same group, in order to reduce the overhead of vCPU migration across NUMA domains.

ACKNOWLEDGMENT

The research is supported by National Science Foundation of China under grant No.61232008 and No.61472151, National 863 Hi-Tech Research and Development Program under grant No.2014AA01A302 and No.2015AA011402, and Research Fund for the Doctoral Program of MOE under grant No.20110142130005.

REFERENCES

[1] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, "Bws: balanced work stealing for time-sharing multicores," in *Proc. EuroSys'12*, 2012, pp. 365–378.

[2] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal, "Efficient load balancing for wide-area divide-and-conquer applications," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPoPP '01. New York, NY, USA: ACM, 2001, pp. 34–43.

[3] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 517–518.

[4] R. D. Blumofe and D. Papadopoulos, "The performance of work stealing in multiprogrammed environments," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1. ACM, 1998, pp. 266–267.

[5] Amazon web services. http://aws.amazon.com/.

[6] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proc. HPDC'11*, 2011, pp. 239–250.

[7] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for smp vms," in *Proc. ASPLOS'13*, 2013, pp. 369–380.

[8] J. Rao and X. Zhou, "Towards fair and efficient smp virtual machine scheduling," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 273–286.

[9] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for smp vms?" in *Proc. EuroSys'11*, 2011, pp. 257–272.

[10] C. E. Leiserson, "The cilk++ concurrency platform," *J. Supercomput.*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

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

[12] C. Weng, Z. Wang, M. Li, and X. Lu, "The hybrid scheduling framework for virtual machine systems," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 111–120.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[15] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[16] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: Balanced work stealing for time-sharing multicores. http://jason.cse.ohio-state.edu/bws/.

[17] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. ASPLOS-IX*, 2000, pp. 234–244.

[18] Vmware. http://www.vmware.com/.

[19] Kvm. http://www.linux-kvm.org/.

[20] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129.

[21] *SLAW: A scalable locality-aware adaptive work-stealing scheduler*, 2010.

[22] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," in *ACM Sigplan Notices*, vol. 44, no. 4. ACM, 2009, pp. 55–64.

[23] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew, "An adaptive task creation strategy for work-stealing scheduling," in *Proc. CGO'10*, 2010, pp. 266–277.