Robinhood: Towards Efficient Work-stealing in Virtualized Environments

Yaqiong Peng, Song Wu, *Member, IEEE,* Hai Jin, *Senior Member, IEEE*

Abstract—Work-stealing, as a common user-level task scheduler for managing and scheduling tasks of multithreaded applications, suffers from inefficiency in virtualized environments, because the steal attempts of thief threads may waste CPU cycles that could be otherwise used by busy threads. This paper contributes a novel scheduling framework named Robinhood. The basic idea of Robinhood is to use the time slices of thieves to accelerate *busy threads with no available tasks* (referred to as poor workers) at both the guest *Operating System* (OS) level and *Virtual Machine Monitor* (VMM) level. In this way, Robinhood can reduce the cost of steal attempts and accelerate the threads doing useful work, so as to put the CPU cycles to better use. 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 applications in virtualized environments. Compared to Cilk++ and BWS, Robinhood can reduce up to 90% and 72% execution time of work-stealing applications, respectively.

Index Terms—virtualization, work-stealing, multicore, parallel program optimization

1 Introduction

1.1 Motivation

Work-stealing has been widely used to manage and schedule concurrent tasks of 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 (*victim*). If the victim has available tasks in its local task queue, the thief can successfully steal some tasks from the victim and then run the tasks. Otherwise, the thief randomly selects another thread as a victim. Due to the dynamic load balancing, work-stealing has been advocated as a powerful and effective approach to achieve good performance of parallel applications [2, 3]. However, previous research works on work-stealing have demonstrated that the unsuccessful steals performed by thieves probably waste CPU cycles in competitive environments [1, 4], such as traditional multiprogrammed environments and virtualized environments.

To mitigate this problem of work-stealing in traditional multiprogrammed environments, Cilk++ and Intel *Threading Building Blocks* (TBB) implement a yielding mechanism, namely having a thief yield its core after an unsuccessful steal [5, 6]. However, the yielded core may not be allocated to a busy thread, when the thief is the only ready thread or there are several thieves on the core [1]. Ding et al. implement a state-of-the-art yielding mechanism for mitigating the cost of steal attempts in a *Balanced Work-stealing Scheduler* (BWS), which allows a thief to yield its core to a preempted, busy thread within the same application [1]. Moreover, a thief in BWS falls into sleep after a certain times of unsuccessful steal attempts. To the best of our knowledge, although these strategies enhance the efficiency of work-stealing in traditional multiprogrammed environments, none of them have shown their effectiveness in virtualized 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 [7], have increasingly become the underlying infrastructures to host multithreaded applications. We refer to a *multithreaded application based on a work-stealing software system* as a *work-stealing application*. When work-stealing applications run on *Virtual Machines* (VMs), thieves probably waste CPU cycles at two levels: (1) thieves consume *virtual CPUs* (vCPUs) that should have been used by busy threads at the guest OS level; (2) vCPUs running thieves consume *physical CPUs* (pCPUs) that should have been used by vCPUs running busy threads at the VMM level. Although the yielding mechanism in BWS can help thieves yield their vCPUs to preempted, busy threads within the same applications, it probably slows down the execution of busy threads with no available tasks (as analyzed in Section 2.2.2). Moreover, a thief may still occupy its vCPU after using the yielding mechanism, due to that no preempted busy threads exist in the same application (e.g. when the application runs alone on a VM, and its number of threads is no more than the number of vCPUs).

1.2 Our Approach

To put CPU cycles to better use, we present Robinhood, a scheduling framework that enhances the efficiency of work-stealing in virtualized environments. When a thief fails to steal tasks from a busy thread with no available tasks, and the busy thread has been preempted by guest OS, Robinhood uses *the remaining time slice of the thief* (allocated

• *Y. Peng, S. Wu and H. Jin is with the 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. The corresponding author is Song Wu. E-mail: wusong@hust.edu.cn*

by guest OS) to accelerate the busy thread. When a thief fails to steal tasks from a busy thread with no available tasks, and the busy thread is running on a vCPU that has been preempted by VMM, Robinhood uses *the remaining time slice of the vCPU running the thief* (allocated by VMM) to accelerate the vCPU running the busy thread. A *first-migrate-then-push-back* policy is proposed for acceleration at both the guest OS level and VMM level.

Robinhood applies to work-stealing applications. In multicore era, many applications use parallel programming models to utilize the abundant computing resources. Work-stealing has been integrated into most of parallel programming models, such as Cilk [8, 9], Cilk++ [5], Intel TBB [6], and OpenMP. Therefore, work-stealing applications span a wide spectrum, providing a wide application range for Robinhood.

1.3 Evaluation

We implement Robinhood based on BWS, Linux and Xen, and compare its efficiency to the state-of-the-art strategies including Cilk++ [5] under Credit scheduler of Xen [22] and co-scheduling [24] respectively, and BWS under balance scheduling [13]. Our evaluation with various work-stealing and non-work-stealing applications demonstrates that the advantages of Robinhood include (1) reducing up to 90% and 72% execution time of work-stealing applications compared to Cilk++ and BWS, respectively, and (2) outperforming other strategies for system throughput by up to $4.8\times$, without sacrificing the performance of non-work-stealing applications. Moreover, Robinhood only incurs less than 1% of overhead to Xen.

1.4 Contributions

In summary, this paper makes the following contributions:

- We discuss the behaviors of existing strategies for work-stealing applications in virtualized environments, and analyze the problems.
- To put CPU cycles to better use, we present a novel scheduling framework named Robinhood, and discuss the challenges to realize Robinhood.
- We rigorously implement and evaluate a prototype of Robinhood based on BWS, Linux and Xen.

1.5 Outline

The rest of the paper is organized as follows. Section 2 analyzes the problems of typical strategies for work-stealing applications in virtualized environments. Section 3 presents the key idea of Robinhood, followed by analyzing the challenges to realize Robinhood. Section 4 and Section 5 describe the design and implementation of Robinhood, respectively. Section 6 provides a comprehensive evaluation of Robinhood. Section 7 discusses the related work, and Section 8 concludes the paper.

2 Problem Analysis

In this section, we first identify three roles of threads in work-stealing applications for the convenience of the problem analysis. Then, we analyze the problems of existing strategies for work-stealing applications in virtualized environments.

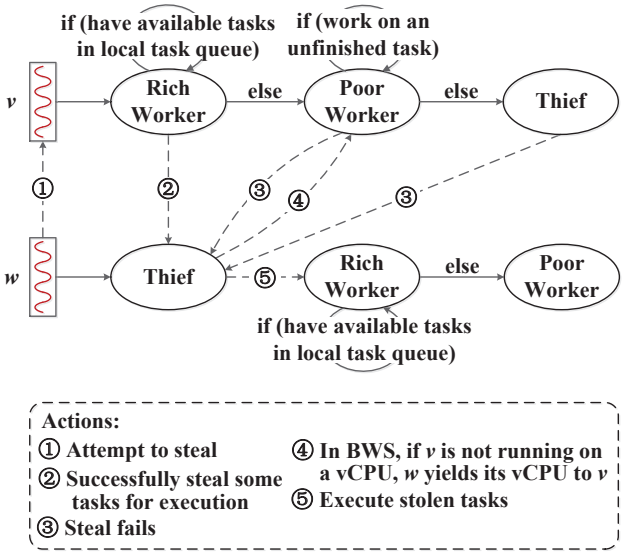


Fig. 1: Actions between a thief w and victim v

2.1 Identifying Thread Roles

We define the following roles of threads in work-stealing applications:

- **Rich worker:** a thread which works on an unfinished task and has available tasks in its local task queue
- **Poor worker:** a thread which works on an unfinished task but has no available tasks in its local task queue
- **Thief:** a thread which runs out of its local tasks and attempts to steal tasks from other threads

Generally, the work-stealing scheduler assigns plentiful tasks to each thread at start. Therefore, each thread is a rich worker at initial state. As a thread runs or is stolen by other threads, its local task queue increasingly lacks of available tasks. When the thread has no available tasks in its local task queue, if it still has an unfinished task to work on, it is a poor worker. Otherwise, it is a thief. As shown in Figure 1, when a thief w randomly selects a thread v as victim and attempts to steal tasks from v, w takes different actions according to the roles of v. Only if v is a rich worker, w can steal some tasks from v. Then, w executes a stolen task, and thus turns into a rich worker or poor worker, depending on whether available tasks are ready in its local task queue. If w fails to steal tasks from v, w continuously selects another victim until it successfully steals some tasks or the computation ends. Besides, a poor worker can generate new available tasks (e.g., by executing a *spawn* or *parallel for* statement in Cilk++), and then turns into a rich worker.

If a thread is a rich worker or poor worker, it does useful work to forward the progress of its application. If a thread is a thief failing to steal tasks, it consumes resources that should have been used by threads in other two roles. Although BWS provides a state-of-the-art yielding mechanism for mitigating the cost of thieves in traditional multiprogrammed environments, thieves can still significantly waste CPU cycles in virtualized environments.

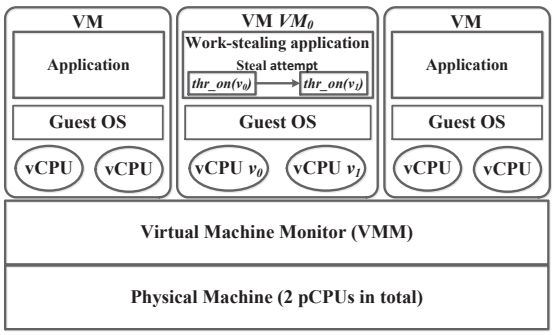


Fig. 2: A scenario of work-stealing applications in virtualized environments

2.2 Problem: Wasted CPU Cycles due to Steal Attempts

When work-stealing applications run on VMs, thieves probably waste CPU cycles at both the guest OS level and VMM level. To simply our analysis, we first assume that an application runs alone on a VM like many previous works [10, 11, 12], and provide an intuitive scenario to illustrate the problems of existing strategies for work-stealing applications. Then, we further analyze the problems when work-stealing applications co-run with other applications within VMs.

2.2.1 Illustrative Examples

Figure 2 shows the scenario: A 2-threaded application is running on a VM VM_0. The application is developed and supported by a work-stealing software system. VM_0 consists of two vCPUs: v_0 and v_1. The *thread running on* v_0 (referred to as $thr\_on(v_0)$) turns into a thief, and attempts to steal some tasks from the *thread running on* v_1 (referred to as $thr\_on(v_1)$). In addition, VM_0 runs simultaneously with other two 2-vCPU VMs on a 2-pCPU physical machine.

In face of the scenario, we discuss the scheduling sequences of vCPUs under three typical strategies: *Proportional fair share scheduling*, *Co-scheduling*, and *BWS*. Generally, most VMMs adopt a proportional fair share scheduling such as Xen and KVM. This scheduling strategy allocates CPU cycles in proportion to the number of shares (weights) that VMs have been assigned. Therefore, we discuss the proportional fair share scheduling in face of work-stealing applications, in order to represent the most common cases. Co-scheduling schedules and de-schedules vCPUs of a VM synchronously, which can alleviate the performance degradation of parallel applications running on SMP VMs. BWS is the state-of-the-art work-stealing scheduler for mitigating the cost of thieves. We do not discuss the strategy that puts co-scheduling and BWS together, because BWS blocks thieves and may cause the *CPU fragmentation* [11, 13] problem for co-scheduling.

When $thr\_on(v_1)$ is a rich worker, $thr\_on(v_0)$ can successfully steal some tasks from $thr\_on(v_1)$, and then forwards the progress of the work-stealing application by executing stolen tasks. As a result, v_0 does not waste CPU cycles on steal attempts under all strategies. When $thr\_on(v_1)$ is also a thief, the work-stealing application ends. Therefore, we focus on the case that $thr\_on(v_1)$ is a poor worker. Figure 3 shows the scheduling sequences of vCPUs under different strategies, when $thr\_on(v_1)$ is a poor worker. We assume that $thr\_on(v_1)$ can generate new tasks and turns into a rich worker after v_1 consumes CPU cycles for a period of 2*T_{TS}. T_{TS} and T_{Period} indicate the time slice and scheduling period of a vCPU in this scenario, respectively. The time slice indicates the time for a vCPU to run on a pCPU in each scheduling period. When a vCPU runs out of its time slice, it is preempted by VMM.

As shown in Figure 3(a)-(b), under proportional fair share scheduling and co-scheduling, v_0 waste CPU cycles on steal attempts for a period of 2*T_{TS} in the interval [T_0, T_5). The reason is that $thr\_on(v_1)$ does not have available tasks for $thr\_on(v_0)$ to steal until T_5, and thus $thr\_on(v_0)$ continuously performs unsuccessful steal attempts on v_0. As a result, the work-stealing application only makes useful progress by $thr\_on(v_1)$ for a period of 2*T_{TS} in the interval [T_0, T_5).

From the perspective of guest OS, the work-stealing application runs with one thread per vCPU and thus its threads do not compete for a vCPU. When using the yielding mechanism in BWS, $thr\_on(v_0)$ still occupies v_0 and performs unsuccessful steal attempts, because no preempted busy threads exist in the same application. Compared to the above two strategies, BWS can reduce the wasted CPU cycles from $2 * T_{TS}$ to T_{TS} by making $thr\_on(v_0)$ sleep after T_1. However, as shown in Figure 3(c), the CPU cycles between T_3 and T_4 is used by vCPUs of other VMs. This part of CPU cycles are originally allocated to v_0 without BWS. Therefore, VMs running work-stealing applications tend to relinquish CPU cycles to VMs running other types of applications under BWS. In other words, although BWS can reduce the cost of vCPUs on steal attempts to some extent, it does not urgently forward the progress of work-stealing applications in virtualized environments. As a result, the work-stealing application still makes useful progress by $thr\_on(v_1)$ for a period of 2*T_{TS} in the interval [T_0, T_5).

Goal 1 Assuming that the first time slice of both v_0 and v_1 is used to execute $thr\_on(v_1)$, what will happen? In this assumption, $thr\_on(v_1)$ can consume CPU cycles for a period of 2*T_{TS} in the interval [T_0, T_2), and thus generates new tasks. Consequently, v_0 would not waste CPU cycles or relinquish pCPU0 to vCPUs of other VMs. Therefore, it is necessary to accelerate vCPUs running poor workers.

2.2.2 Considering Multiprogramming within VMs

A VM leased by users may simultaneously run multiple applications, resulting in multiprogramming within the VM (e.g., multiple applications run in a virtual desktop). When work-stealing applications co-run with other applications within VMs, thieves do not only waste CPU cycles at the VMM level as illustrated above, but also at the guest OS level. Although the yielding mechanism in BWS can help thieves yield their vCPUs to preempted, busy threads within the same applications, it probably slows down the execution of poor workers. To better explain the problem, we provide an example to show the yielding action in Figure 4. T'_{TS} and T'_{Period} indicate the time slice and scheduling period of a thread at the guest OS level, respectively. Given an ***entity*** v (indicating a vCPU or thread), the ***scheduling path*** of v means when and where v is scheduled by scheduler. Figure 4(a) shows the original scheduling path of a poor

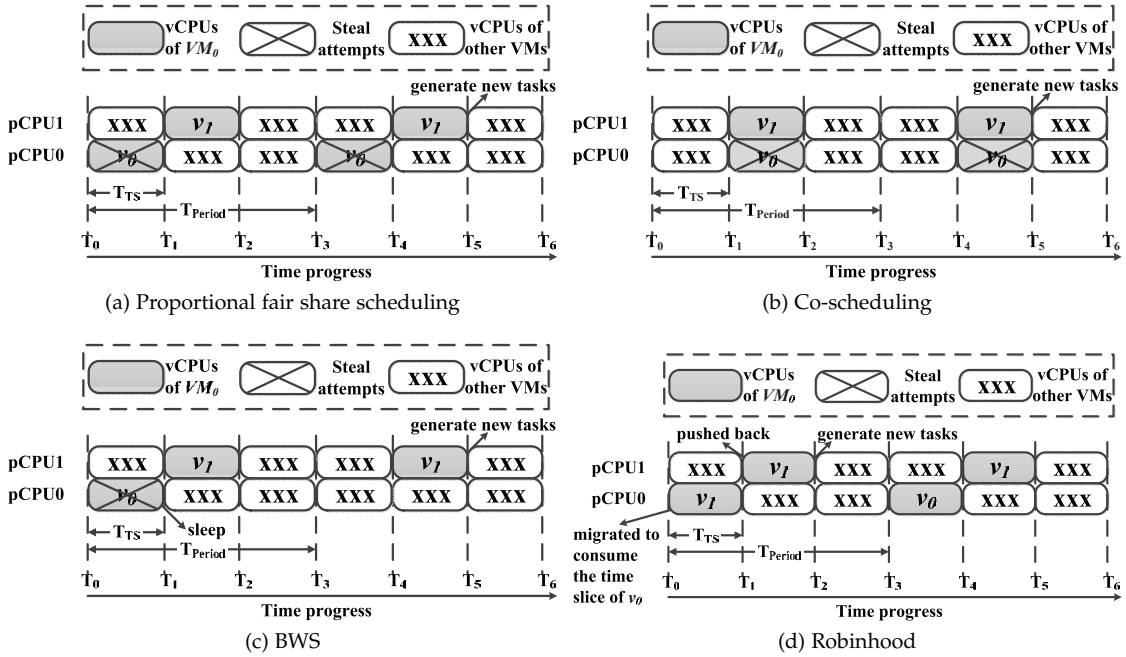


Fig. 3: Scheduling sequences of vCPUs under different strategies in the scenario illustrated by Figure 2, when $thr\_on(v_1)$ is a poor worker

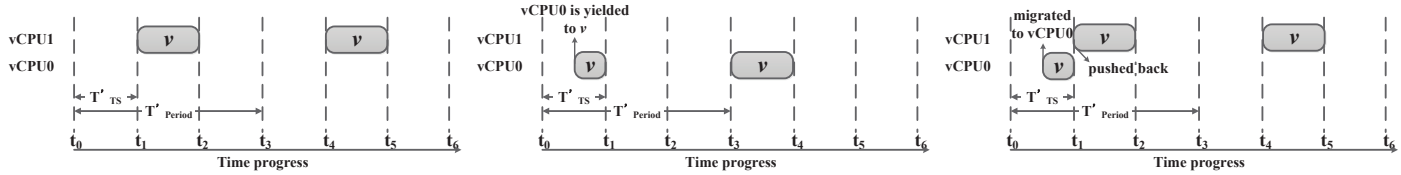


Fig. 4: Scheduling path of a poor worker v at the guest OS level

worker v without BWS at the guest OS level. We assume that a thief running on vCPU0 attempts to steal some tasks from v in the interval $[t_0, t_1)$. As shown in Figure 4(b), the thief yields vCPU0 to v to complete the remaining time slice in BWS. Then, v follows the original scheduling path of the thief on vCPU0. Compared to the original scheduling path of v, v consumes less CPU cycles in BWS. Because a multithreaded application can not end until its last thread finishes, it probably degrades the performance of a work-stealing application by slowing down poor workers.

Goal 2 Therefore, it is necessary to reduce the cost of thieves while not slowing down poor workers at the guest OS level.

3 Our Strategy

In this section, we present the key idea of Robinhood, followed by discussing the challenges to realize it.

3.1 First-Migrate-Then-Push-Back Acceleration Policy

The key idea of Robinhood is to use the time slices of thieves to accelerate poor workers by a *first-migrate-then-push-back* acceleration policy at both the guest OS level and VMM level. When using an entity v_0 to accelerate an entity v_1, the *first-migrate-then-push-back* policy takes two steps: (1) migrating v_1 to the CPU of v_0, and allocating the remaining time slice of v_0 to v_1 for execution; (2) resuming the original scheduling path of v_1.

To compare Robinhood with other strategies, we take the scenario in Section 2.2.1 as an example again. Figure 3(d) shows the possible scheduling sequences of vCPUs under Robinhood, when $thr\_on(v_1)$ is a poor worker.

Once $thr\_on(v_0)$ fails to steal tasks from $thr\_on(v_1)$ at T_0, Robinhood immediately suspends v_0, and migrates v_1 to the pCPU of v_0 (pCPU0). Then, v_1 consumes the remaining time slice of v_0 in the first scheduling period. When the time slice is over at T_1, v_1 is pushed back to its original pCPU (pCPU1), and follows its original scheduling path on pCPU1. In this way, $thr\_on(v_1)$ can consume two time slices from T_0 to T_2, and thus generate new tasks to be a rich worker at T_2. Then, $thr\_on(v_0)$ can successfully steal some tasks from $thr\_on(v_1)$ when v_0 is scheduled at T_3. Different from other strategies, whatever $thr\_on(v_1)$ is a rich worker or poor worker, v_0 does not waste CPU cycles on steal attempts under Robinhood. Moreover, Robinhood does not impact the original scheduling path of other vCPUs.

Besides, Figure 4 shows the example that applies the *first-migrate-then-push-back* policy to the guest OS level. Be-

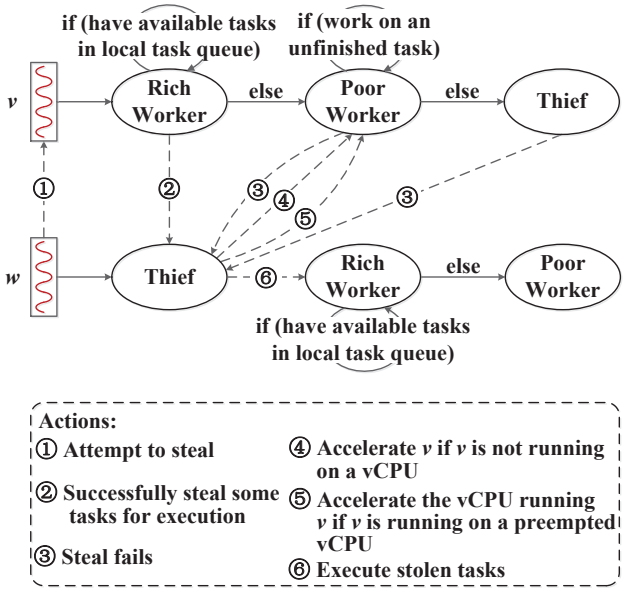


Fig. 5: Actions between a thief w and victim v in Robinhood

cause Robinhood supplies the remaining time slice of the thief to v and retains the original scheduling path of v, Robinhood does not slow down poor workers like BWS, and even accelerates them.

3.2 Two-level Acceleration: A high-level overview of Robinhood

To overview Robinhood at a high-level, we describe the actions between thieves and the corresponding victims in Figure 5. Assuming that w randomly selects v as victim and attempts to steal tasks from v, Robinhood takes different actions according to the roles and running status of v. If v is a rich worker, w can forward the progress of the application by successfully stealing tasks from v. Otherwise, if v is a poor worker not running on a vCPU, Robinhood uses the *first-migrate-then-push-back* policy to accelerate v at the guest OS level. Otherwise, if v is a poor worker running on a preempted vCPU, Robinhood uses the *first-migrate-then-push-back* policy to accelerate the vCPU running v at the VMM level.

3.3 Challenges

To realize Robinhood, we face three challenges.

Challenge 1 *How to bridge the semantic gap between VMM and VMs*?

Robinhood needs VMM to schedule vCPUs running poor workers to the pCPUs of vCPUs running thieves. However, VMM does not know whether vCPUs are running thieves or poor workers due to the semantic gap between VMM and VMs [14]. To bridge the semantic gap, we add new support for the work-stealing runtime and guest OS to disclose whether a vCPU is running a thief or poor worker to VMM.

Challenge 2 *How to minimize the impact of entity migrations incurred by Robinhood on NUMA multicore systems*?

Nowadays, NUMA multicore systems are common in real cloud datacenters. On NUMA multicore systems, it

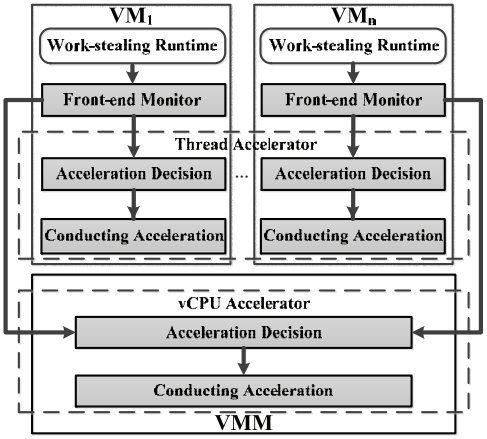


Fig. 6: Architecture of Robinhood

is much more expensive to access memory on a different NUMA node than the local node [15]. Because Robinhood uses migrations to conduct acceleration, it probably increases the cases that entities access remote memory by migrating entities across different NUMA nodes for acceleration. Therefore, we need to add NUMA-aware support for Robinhood.

Challenge 3 *How to resume the original scheduling path of entities*?

Robinhood needs to resume the original scheduling path of an accelerated entity. To achieve this goal, it is necessary to retain the position of entities in their original run queues before conducting acceleration and ensure that their CPU shares are not consumed during the acceleration phase. However, when an entity is migrated from one CPU to another CPU, existing schedulers delete the entity from the run queue of its former CPU, in order to prevent these CPUs scheduling the entity at the same time. Moreover, as an entity runs, existing schedulers generally reduce the CPU shares of the entity in order to maintain fair share of CPU cycles. We need to add new support for schedulers to preserve the original scheduling path of entities, while guaranteeing an entity to be scheduled by one CPU at a time and abiding the fairness principle.

To overcome the challenges, we design and implement a prototype of Robinhood.

4 System Design

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

4.1 System Architecture

Figure 6 overviews the architecture of Robinhood. Therein, *Front-end Monitor* locates in each VM, which is used to capture steal failures of thieves and take different actions according to the status of corresponding victims. If a victim is a preempted, poor worker, *Front-end Monitor* invokes a *system call* (a software trap from an application to OS kernel) to initiate *Thread Accelerator* with the thread IDs of the thief and victim as the arguments. Otherwise, if a victim is a non-preempted, poor worker, *Front-end Monitor* invokes a

hypercall (a software trap from a VM to VMM) to initiate *vCPU Accelerator* with the vCPU IDs of the thief and victim as the arguments (for ***Challenge 1***). *Thread Accelerator* uses the remaining time slices of *thieves* (acceleration initiators) to accelerate *poor workers* (acceleration candidates), while *vCPU Accelerator* uses the remaining time slices of *vCPUs running thieves* (acceleration initiators) to accelerate *vCPUs running poor workers* (acceleration candidates). Both *Thread Accelerator* and *vCPU Accelerator* contain two phases:

- *Acceleration Decision*: Once receiving the arguments passed by *Front-end Monitor*, this component uses the arguments to find the data structures representing the acceleration initiator and acceleration candidate, and then makes acceleration decision according to the running status and NUMA node information of the acceleration initiator and acceleration candidate. One important principle is that forbids entity acceleration across NUMA nodes (for ***Challenge 2***). Only if this phase decides to conduct acceleration, the next phase will be initiated.
- *Conducting Acceleration*: This phase embraces the *first-migrate-then-push-back* acceleration policy. It migrates an acceleration candidate to the vCPU (for *Thread Accelerator*) or pCPU (for *vCPU Accelerator*) of the corresponding acceleration initiator without removing the acceleration candidate from its original run queue, and gives a hint to the scheduler not to schedule the acceleration candidate to its original vCPU or pCPU until the acceleration is over (for ***Challenge 3***). During the acceleration phase, the acceleration candidate consumes the CPU shares of its acceleration initiator (for ***Challenge 3***).

Moreover, as previous literature has identified the negative impact of vCPU stacking (sibling vCPUs contend for one pCPU) on the performance of parallel programs in virtualized environments [13], Robinhood balances all vCPUs of a VM to different run queues like balance scheduling for regular path. For the acceleration path, because Robinhood retains the position of an accelerated vCPU in its original run queue before migration and pushes the vCPU back to the position when the vCPU acceleration phase is over, no stacking could happen.

In the following, we present the details of each phase.

4.2 Front-end Monitor

When a thief w attempts to steal tasks from a randomly selected victim v, w first needs to get the lock of v, in order to guarantee that only one thief may steal tasks from v at a time. Therefore, there are two reasons causing that w fails to steal tasks from v: (1) w fails to get the lock of v; (2) no available tasks are ready in the local task queue of v. As shown in Algorithm 1, *Front-end Monitor* continuously captures steal failures of all thieves due to the second reason by tracking the code concerned with task stealing in work-stealing runtime (line 1), and takes different actions according to the status of corresponding victims.

If v works on an unfinished task but lacks of available tasks in its task queue, it is a poor worker according to the definition in Section 2.1. Then, if v has been preempted by the guest OS, *Front-end Monitor* obtains the thread IDs of w and v, and uses them as the arguments of a system call to initiate *Thread Accelerator* (lines 4-5). Otherwise, *Front-end Monitor* invokes a hypercall to initiate vCPU Accelerator with the vCPU IDs of w and v as the arguments (lines 6-8).

Algorithm 1 Front-end Monitor

```
Input: w: current thread
Output: response for each steal failure of w
1:  for each time w fails to steal tasks from a randomly selected
    victim thread v, due to that no available tasks are ready in
    v's task queue do
2:    /* If v is a poor worker, check the status of v. */
3:    if v works on an unfinished task then
4:      if v has been preempted then
5:        invoke a system call to initiate Thread Accelerator
          with the thread IDs of w and v as the arguments
6:      else
7:        invoke a hypercall to initiate vCPU Accelerator with
          the vCPU IDs of w and v as the arguments
8:      end if
9:    else
10:     /* Attempt to steal tasks from another victim */
11:     random_steal(w)
12:   end if
13: end for
```

How to obtain the vCPU IDs of w and v? Most OSes provide a unique ID for each thread in the OS kernel. Therefore, *Front-end Monitor* can use the thread IDs of w and v to find the corresponding thread data structures. Through the data structures, *Front-end Monitor* can obtain the OS-level IDs of the vCPU running w and the vCPU running v. Generally, each vCPU has a unique ID in the guest OS kernel and VMM, respectively. We observe that the two kinds of vCPU IDs have a certain relationship in most cases. For example, when a Linux-based VM runs on a Xen-based physical machine, the ID of each vCPU in the guest OS kernel is equal to that in the VMM. Therefore, *Front-end Monitor* can obtain the VMM-level vCPU IDs of w and v according to the corresponding OS-level vCPU IDs, which bridges the semantic gap between VMM and VMs.

4.3 Acceleration Decision

Before conducting acceleration, both *Thread Accelerator* and *vCPU Accelerator* make acceleration decision depending on the status and NUMA node information of the acceleration initiator and acceleration candidate. Algorithm 2 shows the details. Before making acceleration decision, *Thread Accelerator* and *vCPU Accelerator* need to acquire the run queue locks of acceleration initiator and acceleration candidate, which is omitted in Algorithm 2. Besides, to record the acceleration initiator and acceleration candidate associated with each entity, we add two variables (*acct_initiator* and *acct_candidate*) to both the per-thread and per-vCPU data structure.

Forbidding entity acceleration across NUMA nodes. As described in previous sections, the acceleration phase contains migration operations for entities. However, the overhead of entity migration across NUMA nodes is usually very high. To minimize the impact of entity migration on

Algorithm 2 Making Acceleration Decision

Input: *acct_initiator*: an acceleration initiator; *acct_candidate*: an acceleration candidate
Output: acceleration decision

```
1: if acct_initiator and acct_candidate reside in the same NUMA node then
2:    /* If acct_candidate is not being accelerated by another entity, and has been preempted by scheduler */
3:    if acct_candidate→acct_initiator == NULL
      && acct_candidate→is_running == FALSE then
4:       time_slice = get_time_slice(acct_initiator)
5:       remaining_time_slice = time_slice - runtime(acct_initiator)
6:       /* If acct_initiator has remaining time slice */
7:       if remaining_time_slice > 0 then
8:          acct_initiator→acct_candidate = acct_candidate
9:          /* If acct_initiator is an accelerated entity, conduct acceleration inheritance mechanism */
10:         if acct_initiator→acct_initiator != NULL then
11:            acct_candidate→acct_initiator =
               acct_initiator→acct_initiator
12:         else
13:            acct_candidate→acct_initiator = acct_initiator
14:         end if
15:         raise_softirq(SCHEDULE_SOFTIRQ)
16:      end if
17:   end if
18: else
19:   return to the user level
20: end if
```

the efficiency of work-stealing, an acceleration initiator is not allowed to accelerate entities from different NUMA nodes (line 1). To emphasize the importance of this intuitive principle, we evaluate the efficiency of Robinhood without NUMA consideration in Section 6.7.

Guaranteeing exclusiveness of the acceleration for entities. The variable *acct_initiator* associated with each entity is used to save the acceleration initiator that selects the entity as an acceleration candidate. When an entity v_0 attempts to select another entity v_1 as an acceleration candidate, if the value of *acct_initiator* associated with v_1 is not NULL, the attempt of v_0 is rejected. In this way, an entity is guaranteed to be accelerated by an acceleration initiator at a time.

If an acceleration candidate is not being accelerated (namely its *acct_initiator* value is NULL) and has been preempted by scheduler, Robinhood computes the remaining time slice of the corresponding acceleration initiator (lines 3-5). If the remaining time slice is greater than zero, Robinhood first saves the information of the acceleration initiator and acceleration candidate into the corresponding variables, and then raises a soft IRQ interrupt on this CPU (lines 7-16). The IRQ interrupt forces a call to the main schedule function, where the acceleration phase will be initiated.

Acceleration inheritance. It is important to note that an accelerated entity v_0 may select another entity v_1 as an acceleration candidate before v_0 completes the time slice supplied by its acceleration initiator v_2. In this case, v_1 must inherit the acceleration initiator from v_0 and its variable *acct_initiator* is set as v_2 instead of v_0, because the remaining time slice of v_0 indeed comes from v_2. We call this mechanism as *acceleration inheritance* (lines 10-11).

4.4 Conducting Acceleration

Algorithm 3 Conducting Acceleration

Input: *curr*: current entity; *runq*: the run queue of the CPU where the scheduler resides; *default_time_slice*: the default time slice in the scheduler
Output: scheduling decision

```
1: /* If curr is an accelerated entity, curr consumes the CPU shares of its acceleration initiator, and curr is not inserted into runq. Otherwise, follow the regular path */
2: if curr→acct_initiator != NULL then
3:    burn_runtime(curr→acct_initiator, runtime(curr))
4: else
5:    burn_runtime(curr, runtime(curr))
6:    insert(curr)
7: end if
8: /* If curr is an acceleration initiator and has remaining time slice, the corresponding acceleration candidate is migrated to this CPU without being removed from its original run queue */
9: if curr→acct_candidate != NULL then
10:   acct_candidate = curr→acct_candidate
11:   time_slice = get_time_slice(curr)
12:   remaining_time_slice = time_slice - runtime(curr)
13:   if remaining_time_slice > 0 then
14:      save_original_cpu(acct_candidate)
15:      migrate_to_cpu(acct_candidate, get_cpu(curr))
16:      set_time_slice(acct_candidate, remaining_time_slice)
17:      next = acct_candidate
18:   else
19:      /* Push the acceleration candidate back to its original run queue */
20:      acct_candidate→acct_initiator = NULL
21:      have next point to the non-accelerated entity that is closest to the head of runq
22:      remove(next)
23:      set_time_slice(next, default_time_slice)
24:   end if
25: else
26:   have next point to the non-accelerated entity that is closest to the head of runq
27:   remove(next)
28:   set_time_slice(next, default_time_slice)
29: end if
30: curr→acct_candidate = NULL
31: /* Push the accelerated entity back to its original run queue */
32: if curr→acct_initiator != NULL then
33:   set_cpu(curr, get_original_cpu(curr))
34:   curr→acct_initiator = NULL
35: end if
36: Context switch to next
```

Algorithm 3 shows the details of the *first-migrate-then-push-back* policy, which is based on the proportional fair share scheduler.

Generally, OS kernel and VMM monitor the CPU shares of each entity in certain forms. For example in the Xen, the CPU shares of each vCPU are monitored in *credits*. As an entity runs, it consumes its CPU shares. When an entity runs out of its CPU shares, it has exceeded its fair share of CPU cycles. To conduct the acceleration phase, Robinhood has an accelerated entity consume the CPU shares of its acceleration initiator. Therefore, when scheduler deschedules an entity *curr*, Robinhood checks whether *curr* is an accelerated entity. If *curr* is an accelerated entity, the CPU cycles consumed by it is paid by its acceleration initiator (lines 2-3). Moreover, *curr* is not inserted into the current

run queue *runq* because it will be pushed back to its original run queue.

If *curr* is an acceleration initiator with remaining time slice, Robinhood saves the CPU of the corresponding acceleration candidate (line 14). Then, the acceleration candidate is migrated to the CPU of *curr* without being removed from its original run queue (line 15), in order to retain the position of the acceleration candidate in its original run queue and minimize the migration overhead.

If Robinhood determines to accelerate an entity, the entity will be run by the scheduler (line 17). Otherwise, the scheduler will run the non-accelerated entity that is closest to the head of the local run queue *runq* (lines 21-23, and lines 26-28). If the *acct_initiator* value of an entity is NULL, it means that it is a non-accelerated entity. Otherwise, the entity is accelerated on another CPU and can not be scheduled to its original CPU until the acceleration phase is over. When the acceleration phase is over, namely the accelerated entity is preempted by the scheduler, the entity is pushed back to its original run queue by updating its CPU information and setting the value of its *acct_initiator* as NULL (lines 32-35).

In summary, the acceleration phase needs no extra run queue operations, and the original scheduling path of accelerated entity are preserved.

5 Implementation

We have implemented Robinhood based on BWS, Linux and Xen. We choose BWS because it is the state-of-the-art 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 open source code. Robinhood requires simple changes to BWS, Linux and Xen.

Front-end Monitor. *Front-end Monitor* locates in each VM. It 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 invokes a new system call to initiate *Thread Accelerator*. Otherwise, the yielder invokes a new hypercall to initiate *vCPU Accelerator*. We implement the system call and hypercall based on Linux and Xen, respectively.

Acceleration Decision. Once receiving the system call (or hypercall) initiated by *Front-end Monitor*, *Thread Accelerator* (or *vCPU Accelerator*) uses the arguments to find the thread (or vCPU) data structures representing the corresponding acceleration initiator and acceleration candidate, respectively. From the data structures, Robinhood can obtain the running status and NUMA node information of acceleration initiators and acceleration candidates for making acceleration decision. To record the information of acceleration initiators and acceleration candidates at both the guest OS level and VMM level, we add two variables *acct_initiator* and *acct_candidate* to both the data structures *task_struct* in Linux and *vcpu* in Xen.

Conducting Acceleration. For *Thread Accelerator*, this module is modified from the Completely Fair Scheduler (CFS) in Linux, which monitors the CPU shares of each thread in *virtual runtime*. For *vCPU Accelerator*, this module is modified from the Credit Scheduler in Xen, which monitors the CPU shares of each vCPU in *credits*. Both CFS and Credit Scheduler are proportional fair share schedulers. Therefore, to accelerate an entity v_1, Robinhood makes v_1 consume the CPU shares of its acceleration initiator during the acceleration phase, by adjusting the virtual runtime or credits of v_1's acceleration initiator instead of v_1.

6 Evaluation

With the implementation of Robinhood, we carry out our experiments on a machine consisting of two 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 the following, we first introduce characteristics of the selected benchmarks and our experimental methodology, then present the experimental results.

6.1 Benchmarks

Table 1 describes the benchmarks (including work-stealing and non-work-stealing applications) used in our experiments. All the benchmarks except Loop and SPECjbb2005 are developed with Cilk++ and provided by authors of BWS [1]. BWS and Robinhood completely retain the same *Application Programming Interfaces* (APIs) of Cilk++. Loop and SPECjbb2005 are used to evaluate the impact of Robinhood on non-work-stealing applications.

6.2 Experimental Methodology

We compare Robinhood with the following related strategies:

- *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 for multiprogrammed environments. We evaluate BWS along with balance scheduling in order to make the comparison with Robinhood fair.

We conduct several experiments to compare Robinhood with the above strategies, in order to answer the following questions.

(1) As the *vCPU online rate* (the percentage of time of a vCPU being mapped to a pCPU) decreases, what is the effectiveness of Robinhood in mitigating the performance degradation of work-stealing applications?

- **Section 6.3:** There is a growing need for cloud providers to consolidate more VMs on a physical machine to generate more revenue, while guaranteeing good performance of hosted applications to

Table 1: Benchmarks

Benchmarks	Description	Source
CG	Conjugate gradient, size B.	NPB 3.3
CLIP	Parallelized computational geometry algorithm on a large number of polygon pairs.	BWS
EP	Embarrassingly parallel, size B.	NPB 3.3
MI	Matrix inversion algorithm on a 500×500 matrix.	Cilk++
MM	Multiply two 1600×1600 matrices.	Cilk++
RT	A 3D render on an 800×600 image using ray tracing techniques, which casts 4 rays of light from each pixel to generate a vivid 3D world.	BWS
Loop	A parallel application developed with pthreads.	BWS
SPECjbb2005	Evaluate the performance of server side Java by emulating a three-tier client/server system.	SPECjbb 2005

attract more customers [12]. The more VMs being consolidated on a single machine, the lower the vCPU online rate will be. Therefore, we conduct experiments to study the effectiveness of Robinhood in mitigating the performance degradation of work-stealing applications on a VM with varying vCPU online rate.

(2) Does Robinhood efficiently support the mix of work-stealing applications?

- **Section 6.4:** We also evaluate the mix of work-stealing applications, each of which runs on a VM. In this type of mix, different work-stealing applications interfere with each other. It is important to guarantee the performance of each application and the overall system throughput.

(3) What is the impact of Robinhood on non-work-stealing applications?

- **Section 6.5:** In real cloud environments, there are also many non-work-stealing applications running on VMs. It is important for Robinhood to effectively mitigate the performance degradation of work-stealing applications without sacrificing the performance of non-work-stealing applications. Therefore, we evaluate the impact of Robinhood on non-work-stealing applications.

(4) What is the overhead incurred to Xen by Robinhood?

- **Section 6.6:** Robinhood prolongs the main schedule function in Xen, which takes up the time that can be used to run vCPUs. Therefore, we create up to eight VMs, and evaluate the overhead that Robinhood incurs to the critical execution path of Xen.

(5) Compared to our preliminary work [17], what is the improvement of Robinhood?

- **Section 6.7:** To demonstrate the novelty of this paper, we conduct experiments to evaluate the extensions of Robinhood over our preliminary work.

All the VMs are configured with 16 vCPUs and 4096MB memory. They are set to share all the cores in the physical machine. This setting makes us focus on the vCPU scheduling in VMM, without considering extra problems such as how to place VMs in a physical machine.

Table 2: The execution time of CG running on VM_0

vCPU online rate (%)	Number of Threads	Baseline	BWS	Robinhood
100	16	10.94s	11.19s	10.93s
	32	11.61s	11.95s	13.23s
	64	22.83s	12.03s	14.94s
50	16	70.54s	33.97s	25.70s
	32	221.76s	38.15s	29.75s
	64	398.75s	61.61s	47.34s
25	16	162.73s	80.49s	57.07s
	32	402.16s	789.92s	488.59s
	64	973.47s	966.39s	879.35s
12.5	16	539.67s	164.01s	120.22s
	32	2088.83s	2131.15s	1691.35s
	64	4152.39s	2290.31s	1970.75s

6.3 Varying vCPU Online Rate

In this testing scenario, each work-stealing application runs on a VM VM_0. We borrow the experimental approach from [10] to adjust the vCPU online rate of VM_0. Recent research from VMware shows that the average vCPU-to-pCPU ratio is 4:1 based on the analysis of 17 real-world datacenters [18]. Consequently, the average vCPU online rate of a VM is 25%. We set the vCPU online rate of VM_0 to be 100%, 50%, 25%, and 12.5%, in order to simulate the real scenario. As these work-stealing applications are CPU-bound workloads, they are not likely to run with more threads than vCPUs. For example, Table 2 shows the execution time of CG running on VM_0, where Robinhood outperforms other strategies in nearly all cases. Under the same vCPU online rate and strategy, CG runs faster with 16 threads than 32 and 64 threads. Therefore, we match the number of threads in work-stealing applications with the number of vCPUs in the VMs like many previous research papers [10, 12].

To provide an intuitive comparison, we define the *slowdown* of an application running on VM_0 as follows:

$$Slowdown = \frac{T}{T_b} \quad (1)$$

where T is the execution time of the application, and T_b is the execution time of the same application running on VM_0 with 100% vCPU online rate under the baseline. The lower slowdown indicates the better performance.

The slowdown of all the work-stealing applications under the baseline, BWS and Robinhood are shown as Figure 7. When the vCPU online rate is 100%, the applications running on VM_0 give similar performance with the three strategies except CLIP and MI. When the vCPU online rate

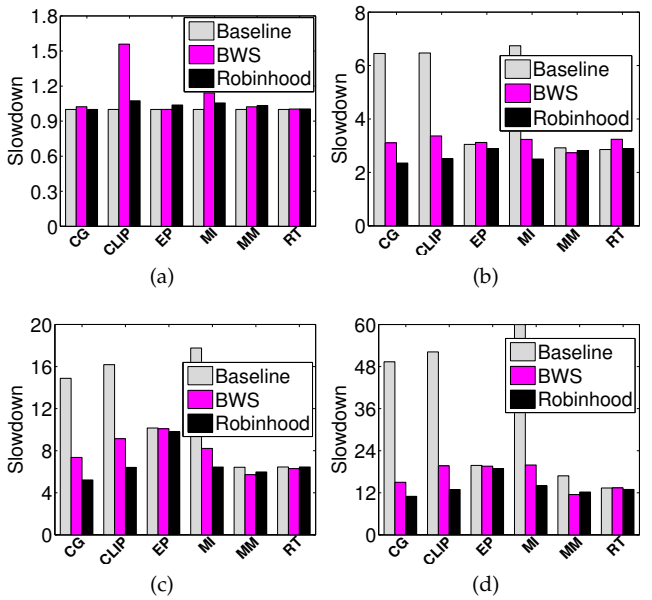


Fig. 7: Slowdowns of work-stealing applications in VM_0, where the vCPU online rate is: (a) 100%, (b) 50%, (c) 25%, and (d) 12.5%

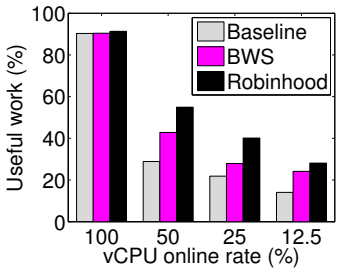


Fig. 8: Useful work ratio of CG under different strategies

decreases, Robinhood outperforms the baseline and BWS for CG, CLIP and MI in all aspects, and gives similar performance of EP, MM, and RT with another two strategies. The reason is that EP, MM, and RT have good scalability, and their threads spend only a small amount of time on stealing with the original Cilk++ [1]. Thus, there is only a limited opportunity for Robinhood to show improvements. In contrast, CG, CLIP and MI have only fair scalability, and thus their workers spend much more time on stealing [1]. As a result, these applications tend to waste CPU cycles and suffer from significant performance degradation under the baseline, when the vCPU online rate decreases.

BWS monitors and controls the number of awake thieves, and thus the vCPUs running such threads are also timely blocked. Therefore, the original BWS can also improve the performance of CG, CLIP and MI to some extent. Robinhood further optimizes the performance of work-stealing applications by accelerating the execution of vCPUs running poor workers. To be more specific, Robinhood saves up to 77% and 34% execution time of work-stealing applications compared to Cilk++ and BWS, respectively.

To further explain the performance power of Robinhood, we evaluate the effectiveness of Robinhood in reducing the cost of vCPUs running thieves. If a work-stealing application spends more time on doing useful work instead of stealing, the cost of vCPUs running thieves from the application will be lower. Therefore, we instrument the useful work ratio of work-stealing applications under the baseline, BWS and Robinhood. Figure 8 shows the useful work ratio of CG with the three strategies. As depicted in Figure 8, compared to the baseline and BWS, Robinhood significantly improves the useful work ratio of CG. This indicates that Robinhood puts the CPU cycles to the best use. The other work-stealing applications have similar figures.

Summary. As the vCPU online rate decreases, Cilk++ suffers from inefficiency for work-stealing applications running on VMs. BWS can improve the efficiency of Cilk++ to some extent by monitoring and controlling the number of awake thieves. Robinhood optimizes BWS by further reducing the cost of vCPUs running thieves and accelerating the execution of vCPUs running poor workers. In this way, the CPU cycles can be put to better use.

6.4 Mix of Work-stealing Applications

In this testing scenario, we simultaneously run four VMs, each of which runs a different work-stealing application from Table 1. As different applications have different execution time, we run each application repeatedly in all experiments, so that their executions can be fully overlapped.

In order to study the behaviour of co-running applications under different strategies, 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 speedup of *App(n)* in the co-running is defined as follows:

$$Speedup(n) = \frac{T_s(n)}{T_n(n)} \tag{2}$$

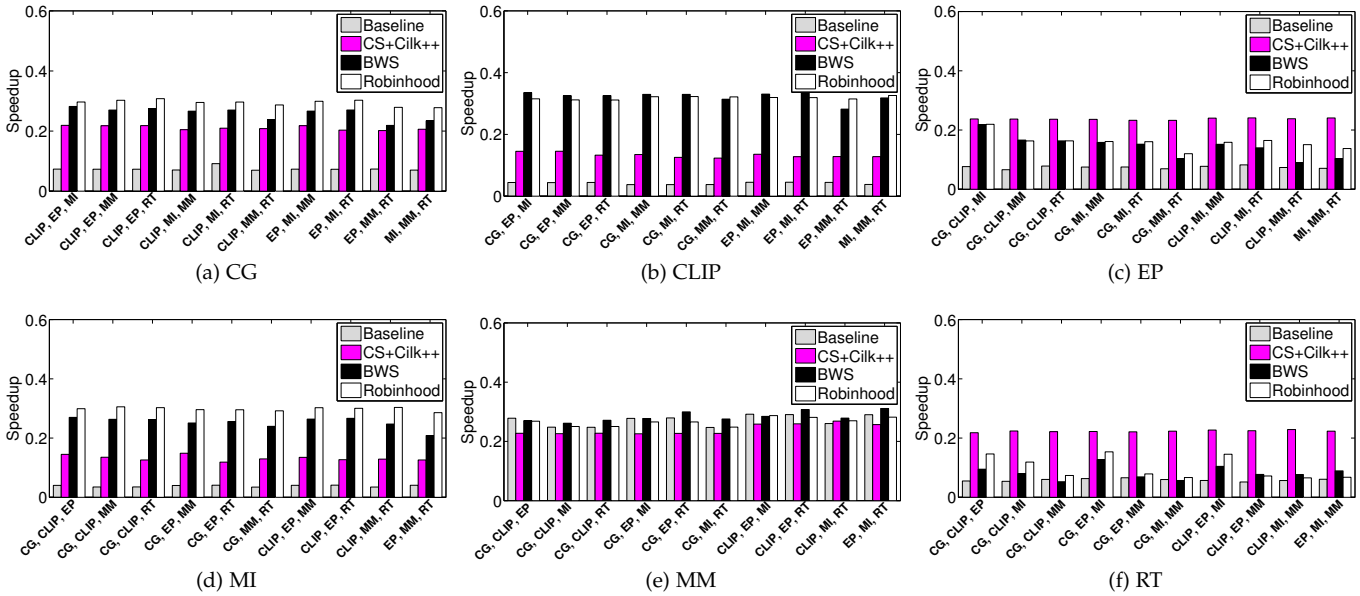
The higher speedup indicates the better performance.

We use the weighted speedup [19] to measure system throughput, which is the sum of the speedups of co-running applications:

$$Throughput = \sum_{i=1}^{n} Speedup(i) \tag{3}$$

Figure 9 compares the speedup of the applications in different co-running cases under the baseline, CS+Cilk++, BWS, and Robinhood. Accordingly, Figure 10 shows throughputs of different co-running applications under the baseline, CS+Cilk++, BWS, and Robinhood. From these two figures, we can observe that the baseline performs the worst for both the performance of work-stealing applications and system throughput.

Co-scheduling is to time-slice all the cores so that each application is granted a dedicated use of the cores during its scheduling quota. Therefore, with CS+Cilk++, the speedup of the most applications is closely to the vCPU online rate, resulting in the better performance than the baseline. To be specific, CS+Cilk++ outperforms other strategies for the performance of EP and RT in all co-running cases, while achieving much worse performance of work-stealing applications that have only fair scalability (CG, CLIP and MI) than BWS and Robinhood. The reason is that vCPUs running



(a) CG (b) CLIP (c) EP

(d) MI (e) MM (f) RT

Fig. 9: Four work-stealing applications run simultaneously on four VMs: Each subfigure takes a work-stealing application as main application. The y-axis presents the speedup of main application in different co-running cases, and the x-axis presents other co-running applications.

thieves waste significant CPU cycles in such applications under CS+Cilk++. As a result, CS+Cilk++ achieves much worse system throughput than BWS and Robinhood, when a co-running case includes more than one work-stealing applications that have only fair scalability.

Because BWS reduces the cost of vCPUs on steal attempts by blocking some thieves, it guarantees better performance of work-stealing applications and system throughput than the baseline and CS+Cilk++ in most cases.

Robinhood both reduces the cost of vCPUs running thieves and accelerates the execution of vCPUs running poor workers by supplying the time slices of the former to the latter. As a result, Robinhood outperforms other strategies for the performance of CG (by up to 77%, 33%, and 22% compared to the baseline, CS+Cilk++, and BWS, respectively) and MI (by up to 89%, 60%, and 37% compared to the baseline, CS+Cilk++, and BWS, respectively) in all the co-running cases. For CLIP and MM, Robinhood performs similar to BWS. For EP, Robinhood only performs worse than CS+Cilk++ (achieving 64% performance of EP compared to CS+Cilk++ on average). Moreover, Robinhood achieves 44% performance of RT compared to CS+Cilk++ on average, and improves the performance of RT by 19% compared to BWS on average. In the aspect of system throughput, Robinhood performs the best in almost all co-running cases. On average, Robinhood outperforms the baseline, CS+Cilk++, and BWS by 60%, 16% and 6%, respectively. Although Robinhood only slightly outperforms BWS for system throughput, Robinhood guarantee better performance of work-stealing applications than BWS.

Summary. Compared to other strategies, Robinhood can guarantee both the performance of each application and the overall system throughput in most mixes of work-stealing applications.

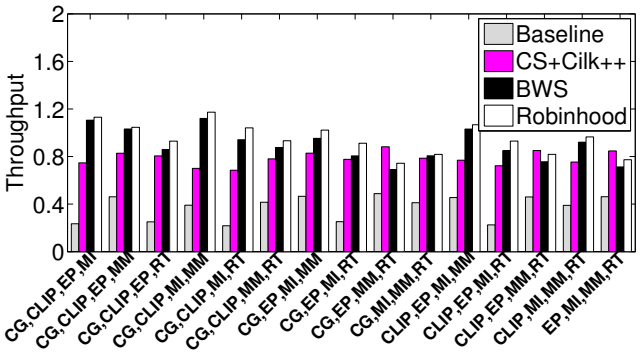


Fig. 10: Throughput of four simultaneously running work-stealing applications

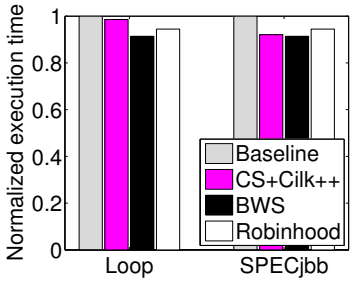


Fig. 11: The impact of Robinhood on non-work-stealing applications

6.5 Impact of Robinhood on Non-work-stealing Applications

In this testing scenario, we use Loop and SPECjbb to represent non-work-stealing applications, and evaluate the impact of Robinhood on them.

We use four VMs sharing the same physical machine

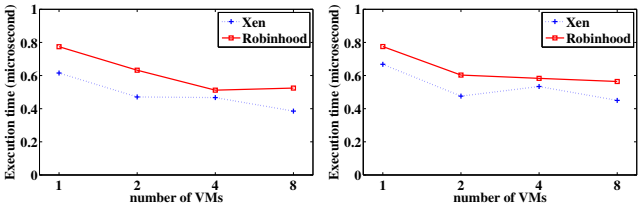


Fig. 12: Overhead of Robinhood to Xen, where the scheduling interval is: (a) 30ms, and (b) 5ms

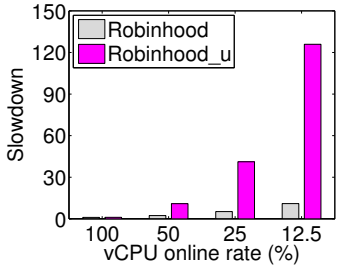


Fig. 13: Slowdown of CG under Robinhood_u and Robinhood

to conduct this test, and measure the performance of non-work-stealing applications under the baseline, CS+Cilk++, BWS and Robinhood. Because CG, CLIP and MI tend to achieve performance improvement from Robinhood, we run CG, CLIP and MI on three VMs as interfering applications, in order to maximize the impact of Robinhood on non-work-stealing applications. Figure 11 shows the results, where the execution time of applications running with CS+Cilk++, BWS and Robinhood is normalized to that with the baseline.

From the test result, we can observe that both Loop and SPECjbb achieve the best performance under BWS. Th reason is that VMs running work-stealing applications tend to relinquish their pCPUs to VMs running non-work-stealing applications under BWS, as analyzed in Section 2.2.1. In contrast, Robinhood retains CPU cycles for VMs running work-stealing applications by supplying the time slices of vCPUs running thieves to sibling vCPUs running poor workers. As a result, Robinhood slightly decreases the performance of Loop and SPECjbb by 3.4% compared to BWS.

Summary. Compared to other strategies, the interferences of Robinhood on non-work-stealing applications are slight and acceptable.

6.6 Overhead of Robinhood to Xen

In this testing scenario, we create up to eight VMs, and evaluate the overhead of Robinhood to Xen. Each VM randomly runs one application of CG, CLIP and MI. The overhead of Robinhood to Xen comes from two main sources: (1) the time required to invoke the hypercall in *Front-end Monitor*; (2) the time required to perform the vCPU acceleration phase in the per-pCPU schedule function. The first source is included in the execution time of work-stealing applications, and it does not incurs overhead to the critical execution path of Xen. From Figure 7, we can see that the first source does not outweigh the benefit of Robinhood on the work-stealing applications when the vCPU online rate decreases. The second source is more significant because it probably takes up the time that could be otherwise used to run vCPUs. Therefore, this subsection focuses on evaluating the second source.

We profile 100 times of the execution time of the schedule function in Credit and Robinhood for different number of VMs, and compute the average values. Figure 12 shows the results. We can observe that the distance between the execution time of the schedule function in Robinhood and in original Xen does not increase with the number of VMs. As shown in Figure 12(a), when the number of VMs is 1, 2, 4, and 8 under the default scheduling interval of 30ms in Xen, the average distance is 0.159us, 0.162us, 0.45us, and 0.139us, respectively. Therefore, Robinhood only adds less than 1% of overhead to Xen.

Although the overhead of Robinhood grows with the reduction in scheduling interval, reducing the scheduling interval also increases the overhead of context switching and reduces the effectiveness of the CPU cache, resulting in significant performance degradation of applications [20]. As a result, scheduling intervals of 5ms or 10ms give a good balance [20]. Therefore, we also test the overhead of Robinhood to Xen under the scheduling interval of 5ms, and show the experimental results in Figure 12(b). From the figure, we can observe that Robinhood only adds less than 1% of overhead to Xen.

Summary. Robinhood incurs negligible overhead to the critical execution path of Xen.

6.7 Comparison with Preliminary Work

Compared to our preliminary work, there are two extensions in this work: (1) adding NUMA support for Robinhood; (2) enhancing the efficiency of Robinhood for multiprogramming within VMs. Therefore, we evaluate the following strategies:

- *Robinhood_u*: Robinhood without NUMA consideration, namely allowing a vCPU to accelerate another vCPU from different NUMA nodes.
- *Robinhood_s*: Robinhood without multiprogramming consideration within VMs, namely using the yielding mechanism in BWS to reduce the cost of thieves at the guest OS level.

Figure 13 shows the slowdown of CG under Robinhood_u and Robinhood, and the other work-stealing applications have similar figures. We can observe that Robinhood_u performs much worse than Robinhood, because the significant overhead incurred by vCPU migration across NUMA nodes outweigh the benefit of vCPU acceleration. Therefore, it is a simple but important principle that forbids vCPU acceleration across NUMA nodes.

Figure 14 shows slowdowns of work-stealing applications in VM_0, where *interfering applications* (CPU-hungry loop application with 16 threads) exist within VM_0. We can observe that Robinhood performs best in all cases, because it accelerates poor worker at both the guest OS level and VMM level. To be specific, Robinhood reduces up to 90%, 72% and

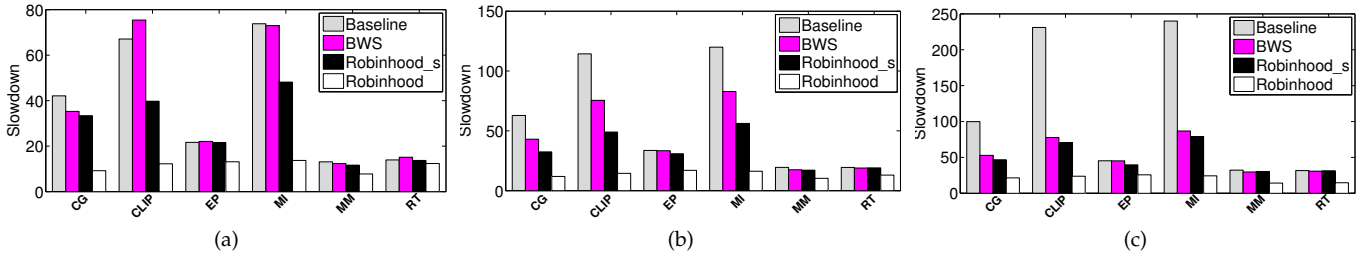


Fig. 14: Slowdowns of work-stealing applications in VM_0 with 25% vCPU online rate, where the number of interfering applications is:(a) 1, (b) 2, and (c) 4.

72% execution time of work-stealing applications compared to the baseline, BWS and Robinhood_s, respectively.

Summary. Compared to preliminary work, this paper extends the use range of Robinhood.

7 Related Work

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

7.1 vCPU Scheduling in Virtualized Environments

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 [21], Xen [22], and KVM [23].

One vCPU scheduling method is to schedule vCPUs of a VM asynchronously. This method simplifies the implementation of the vCPU scheduling in 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 [24] and VMware [21]. The limitation of co-scheduling is that the number of the vCPUs should be no more than the number of the pCPUs [24], and CPU fragmentation occurs when the system load is imbalanced [13]. To solve this issue, many researchers propose demand-based co-scheduling [10, 11], which dynamically initiates co-scheduling only for synchronizing vCPUs. As far as we know, no existing vCPU scheduling policies are designed with the full consideration of the features of work-stealing applications.

7.2 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 the efficiency of work-stealing software systems in traditional multiprogrammed environments [1, 4, 25]. The common idea of these studies is to have thieves yield their computing resources. Inspired by this idea, we present Robinhood with the goal to enhance the efficiency of work-stealing 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 [26], and reducing the overheads of task creation and scheduling [27, 28]. These studies do not address competitive issues for underlying computing resources, which are orthogonal to Robinhood.

Besides, there are some studies about extending work-stealing to distributed memory clusters [29, 30, 31]. Dinan et al. focus on the scalability of work-stealing on distributed memory cluster and design a system [30]. In their approach, when a thief fails to steal tasks from a victim, it continuously selects another victim until it successfully steal some tasks or the computation completes, resulting in wasted CPU cycles on steal attempts. Saraswat et al. proposes a lifeline-based global load balancing that allows work-stealing to run efficiently on distributed memory. In their approach, when a thief is unable to find tasks after a certain times of unsuccessful steals, it is blocked like BWS. However, as discussed in Section 2.2.1, blocked thieves make their applications tend to relinquish CPU cycles in virtualized environments. Therefore, when running on *clusters consisting of VMs* (referred to as virtual clusters), distributed work-stealing applications may also suffer from inefficiency. In future work, we intend to extend Robinhood to improve the efficiency of work-stealing in virtual clusters. A possible approach is to first rely on an efficient cluster scheduler (e.g., Sparrow [32]) to select VMs for running any given distributed work-stealing applications. When a thief fails to steal tasks from a poor worker without making progress, and they belong to a VM, Robinhood uses the *first-migrate-then-push-back* policy to accelerate the poor worker.

8 Conclusion

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

thieves may waste their attained CPU cycles 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 CPU cycles of thieves to busy threads within same applications at both the guest OS level and VMM level, which can put the CPU cycles to better use. 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 applications in virtualized environments.

ACKNOWLEDGEMENTS

This work was supported by National Science Foundation of China under grant No.61472151 and No.61232008, National 863 Hi-Tech Research and Development Program under grant No.2014AA01A302 and No.2015AA01A203, the Fundamental Research Funds for the Central Universities under grant No.2015TS067. The corresponding author is Song Wu.

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 *Proc. PPoPP'01*, 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 *Proc. ICS'09*, 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] C. Leiserson, "The cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.
[6] 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.
[7] Amazon web services. http://aws.amazon.com/.
[8] 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 *Proc. PPoPP'95*, 1995, pp. 207–216.
[9] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. PLDI '98*, 1998, pp. 212–223.
[10] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proc. HPDC'11*, 2011, pp. 239–250.
[11] 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.
[12] J. Rao and X. Zhou, "Towards fair and efficient smp virtual machine scheduling," in *Proc. PPoPP'14*, 2014, pp. 273–286.
[13] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for smp vms?" in *Proc. EuroSys'11*, 2011, pp. 257–272.
[14] H. Liu, H. Jin, X. Liao, B. Ma, and C. Xu, "Vmckpt: lightweight and live virtual machine checkpointing," *Science China Information Sciences*, vol. 55, no. 12, pp. 2865–2880, 2012.
[15] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *Proc. HPCA'13*, 2013, pp. 306–317.
[16] Bws. http://jason.cse.ohio-state.edu/bws/.
[17] Y. Peng, S. Wu, and H. Jin, "Towards efficient work-stealing in virtualized environments," in *Proc. CCGrid'15*, 2015.
[18] V. Soundararajan and J. M. Anderson, "The impact of management operations on the virtualized datacenter," in *Proc. ISCA '10*, 2010, pp. 326–337.
[19] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. ASPLOS-IX*, 2000, pp. 234–244.
[20] Credit scheduler. http://wiki.xensource.com/wiki/Credit_Scheduler.
[21] Vmware. http://www.vmware.com/.
[22] Xen. http://www.xen.org/.
[23] Kvm. http://www.linux-kvm.org/.
[24] C. Weng, Z. Wang, M. Li, and X. Lu, "The hybrid scheduling framework for virtual machine systems," in *Proc. VEE'09*, 2009, pp. 111–120.
[25] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proc. SPAA '98*, 1998, pp. 119–129.
[26] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A scalable locality-aware adaptive work-stealing scheduler," in *Proc. IPDPS'10*, 2010, pp. 1–12.
[27] 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.
[28] 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.
[29] R. D. Blumofe, P. A. Lisiecki *et al.*, "Adaptive and reliable parallel computing on networks of workstations," in *Proc. ATC'97*, 1997, pp. 133–147.
[30] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proc. SC'09*, 2009, pp. 53:1–53:11.
[31] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proc. PPoPP'11*, 2011, pp. 201–212.
[32] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. SOSP'13*, 2013, pp. 69–84.

Yaqiong Peng received her B.S. (2010) degree at Huazhong University of Science and Technology (HUST) in China. Currently she is a Ph.D. candidate student of Computer Science and Technology at Huazhong University of Science and Technology (HUST). Her current research interests include operating systems, virtualization and parallel program optimization.

Song Wu is a professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. from HUST in 2003. He is now the director of Parallel and Distributed Computing Institute at HUST. He is also served as the vice director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST. His current research interests include cloud computing, system virtualization, datacenter management, storage system, in-memory computing.

Hai Jin received the PhD degree in computer engineering from HUST in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at HUST in China. He is currently the dean of the School of Computer Science and Technology at HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a senior member of the IEEE and a member of the ACM. He has coauthored 15 books and published more than 500 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.