TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 02/11 pp7-16
Volume 20, Number 1, February 2015

FlexCore: Dynamic Virtual Machine Scheduling Using VCPU Ballooning

Tianxiang Miao and Haibo Chen*

Abstract: As multi-core processors become the de-facto configuration in modern computers, the adoption of SMP Virtual Machines (VMs) has been increasing, allowing for more efficient use of computing resources. However, because of existence of schedulers in both the hypervisor and the guest VMs, this creates a new research problem, viz., double scheduling. Although double scheduling may cause many issues including lock-holder preemption, vCPU stacking, CPU fragmentation, and priority inversion, prior approaches have either introduced new problems and/or addressed the problem incompletely. In this paper, we describe the design and implementation of FlexCore, a new scheduling scheme using vCPU ballooning, which dynamically adjusts the number of vCPUs of a VM at runtime. This essentially eliminates unnecessary scheduling in the hypervisor layer, and thus, boosts performance significantly. An evaluation using a complete KVM-based implementation shows that the average performance improvement for PARSEC applications on a 12-core Intel machine is approximately 52.9%, ranging from 35.4% to 79.6%.

Key words: virtualization; SMP virtual machine; multicore processor; vCPU ballooning

1 Introduction

In recent years, because of the ability of cloud computing to provide scalable, highly efficient virtual computing resources under the pay-as-you-go model, cloud computing has gained significant momentum in both academia and industry. Virtualization[1, 2], a key enabling technology for cloud computing, facilitates efficient resource allocation to end users in the form of virtual machines.

As Moore's law continues to evolve in the form of multi-core hardware, commodity processors are now equipped with a greater number of CPU cores. This makes running SMP Virtual Machines (SMP-VMs) for exploiting the abundant computing resources a necessity[3]. By consolidation, or by running several VMs on top of a single physical machine, cloud service providers can further improve hardware resource utilization[4]. However, this leads to a new problem for virtualized platforms, viz., double scheduling[5]: (1) the OS scheduler inside the guest VM schedules processes on vCPUs and (2) meanwhile the hypervisor schedules vCPUs on physical CPUs (pCPU).

Double scheduling may lead to severe performance degradation of SMP-VMs. In essence, this phenomenon is caused by inappropriate scheduling issued by the hypervisor because most operations inside a guest VM are opaque to the hypervisor, and thus, the hypervisor has little knowledge about the internal running status, which is referred to as "semantic gap"[6]. For instance, even though one vCPU enters the critical section with kernel preemption disabled, it could be preempted rudely when the vCPU runs out of CPU time slices or an external interrupt is presented. This may greatly increase synchronization latency given that another vCPU is very likely waiting for the spinlock. This performance issue may amplify notably in SMP-VMs

• Tianxiang Miao and Haibo Chen are with Institute of Parallel And Distributed System (IPADS), the School of Software, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: cwndmiao@gmail.com; haibochen@sjtu.edu.cn.

∗ To whom correspondence should be addressed.

Manuscript received: 2014-11-24; revised: 2014-12-01; accepted: 2014-12-07

running parallel applications given that an excessive number of runnable vCPUs contend for available pCPUs.

However, commodity hypervisors do not consider carefully this performance issue caused by double scheduling. To mitigate such performance loss, researchers have proposed several co-scheduling strategies[7–9] to make cooperating vCPUs run concurrently. However, these strategies either introduce new problems such as CPU fragmentation and priority inversion or offer non-significant performance boost. This leaves considerable room for optimization.

In a previous workshop paper, we described vCPU ballooning (shortened as vCPU-Bal) abstractly[5]. Similar to memory ballooning[10], vCPU-Bal dynamically adjusts the number of running vCPUs assigned to each VM, making each vCPU monopolize one pCPU separately. Thus, the performance loss due to pCPU contention is eliminated. The main idea underlying this strategy is to weaken the role played by the hypervisor scheduler, thereby reducing double-scheduling to single-scheduling. In this paper, we describe the design and implementation of FlexCore, which leverages vCPU-Bal to address the challenges caused by double scheduling in a production-quality hypervisor (i.e., KVM).

Our previous workshop paper lacked the necessary details pertaining to the implementation of such a technique in real-world hypervisors. Herein, we identify several challenges presented by the use of such a technique and describe solutions to address them in the context of a popular hypervisor. The challenges include detection of vCPU contention, efficient VM-hypervisor communication, and vCPU hotplugging.

We implemented FlexCore based on KVM-QEMU 1.7.0 with Linux-3.13.3. An evaluation conducted using a 12-core Intel machine shows that FlexCore boosts performance by 52.9% on average when running the PARSEC benchmark suite.

The rest of this paper is organized as follows. Section 2 reproduces the double-scheduling problem in detail and describes why it leads to performance degradation. Section 3 describes the design and implementation of the FlexCore system. We then present our performance evaluation results in Section 4. Finally, Section 5 compares FlexCore with other solutions, and Section 6 provides the conclusions of this study.

2 Background and Analysis

2.1 SMP-VM scheduling

Each SMP-VM is equipped with two or more vCPUs to fully exploit abundant computing resources, and thus, to fulfill the requirements for running parallel applications. In the VM consolidation scenario, one physical CPU may be shared by several vCPUs from different VMs. The hypervisor scheduler allocates CPU time slices among the vCPUs according to their weight (shares). To ensure fairness among VMs, the hypervisor scheduler first tags each VM with the same total shares and then divides these shares equally among all vCPUs belonging to each VM. For every instance of context switch, the hypervisor scheduler updates vCPU's shares based on the CPU time it consumed during the last period. To reduce cache contention, the hypervisor maintains one separate scheduler for each pCPU on a multi-core system. Furthermore, to balance workload among pCPUs, the hypervisor moves vCPUs from busy pCPUs to idle ones in a work-stealing manner[11].

However, the "sematic gap"[6] usually impedes the hypervisor from inferring the actual running status in a guest VM to make an optimal decision. The existence of such a "semantic gap" often results in the hypervisor issuing inappropriate scheduling. For example, the hypervisor scheduler is oblivious to whether a vCPU is in an "urgent" state (e.g., holding a spinlock) and always preempts it. This may seriously increase synchronization latency and hurt application performance.

To illustrate this phenomenon, we conducted a simple experiment using two configurations on a 12-core Intel machine: (1) single-VM case (1VM), where only one guest VM exists in the system, and (2) two-VM case (2VM), where two guest VMs run simultaneously. In these two configurations, each VM is assigned with 12 vCPUs and 8 GB memory, and it executes one of the four parallel applications in the PARSEC benchmark suite[12]. The evaluation results are shown in Fig. 1. Although VMs in the two-VM case occupy half the computing resources compared with the VM in the single-VM case, the total time spent in the two-VM case is 3.2×, 4.1×, 2.7×, and 2.6× longer. More specifically, the time spent in the kernel mode in the two-VM case is 3.7×, 8.7×, 9.0×, 6.3× longer.

To figure out why the performance deteriorates, we used the perf[13] profiling tool to carry out function-level

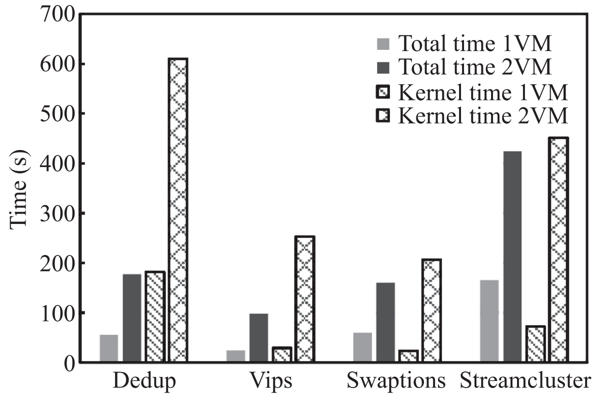


Fig. 1 Comparison on total execution time and kernel time between 1VM case and 2VM case.

sampling and analysis. (Because inline optimization is used in kernel compiling to reduce function-call overhead, we measure the elapsed time in spinlock using lockstat[14]). The result is shown in Fig. 2. The majority of CPU time is wasted in function-call Inter-Processor Interrupt (IPI) and spinlock acquisition, resulting in little CPU time being spent on the actual computing work. This is the main cause of the aforementioned performance deterioration. The following two paragraphs state the internal process in detail.

2.2 Function-call IPI delay

IPI is widely used in SMP OS kernels to notify other CPUs of some specific events such as TLB invalidation and rescheduling[15], and IPI receivers need to call the corresponding handler in response. In Linux, the function-call IPI sender enters the busy waiting state after issuing a request until all handlers in the IPI receivers return[16]. This mechanism is highly efficient in native hardware, because hardware-based IPI is of high priority and consumes only 100–200 CPU cycles[17]. In addition, the IPI receiver often performs small tasks such as appending an entry in the per-CPU linked list, which makes the IPI handler return quickly.

However, in virtualized environments, IPI is emulated by virtual APIC (vAPIC). vCPU sends a virtual IPI by writing an MSR register, which causes traps to the hypervisor. Then, the hypervisor marks its receivers. The next time the IPI receiver is scheduled and enters the non-root mode, the pending interrupt is injected. Under the VM consolidation scenario, it is likely that more than one vCPUs remain pending in the run queue. Therefore, the IPI receiver is required to wait until it gets scheduled, and the IPI sender must keep the busy waiting state for a long time until all receivers return. Moreover, a virtual IPI is always broadcast and has multiple receivers, which makes things worse. As shown in Fig. 3, vCPU0 broadcasts a function-call IPI to its receivers at T_1 and then enters the busy waiting state. However, vCPU1 and vCPU2 cannot be scheduled immediately because other vCPUs from different VMs are occupying pCPUs. Thus, the busy waiting state continues until T_2, at which point other vCPUs finish execution and the IPI handlers return. Here $T_{\text{busy\_waiting}}$ is wasted in meaningless busy waiting.

2.3 Spinlock holder preemption

To avoid race condition, a spinlock is usually used for synchronizing multiple CPUs, and a thread repeatedly checks whether the lock is available before acquiring it[18]. In general, a spinlock critical section is short (e.g., approximately tens of CPU cycles), making spinlock a highly efficient synchronization primitive for native hardware.

Unfortunately, the busy waiting state may be extended significantly in virtualized environments. Although the spinlock holder disables

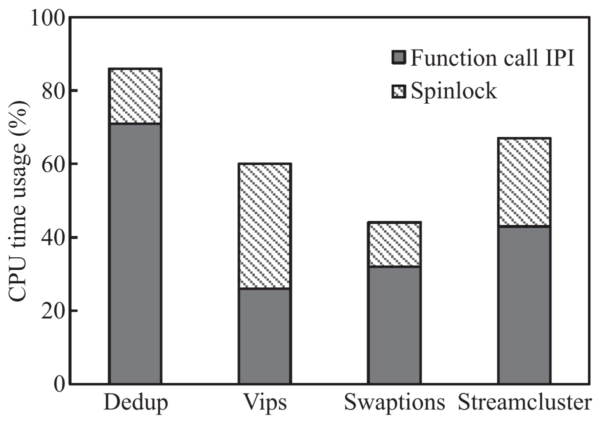


Fig. 2 Breakdown on total execution time in 2VM case.

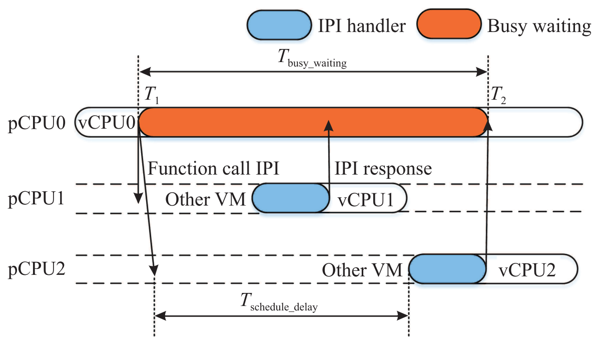


Fig. 3 Illustration on function-call IPI delay.

kernel preemption before entering the critical section, the vCPU may still be preempted once its CPU time slices are consumed or an external interrupt comes. As a result, the time spent busy waiting by other vCPUs will be extended significantly because they must wait until the lock holder is rescheduled and continues the interrupted critical section. Figure 4 shows an example. vCPU0 is the lock holder, and it is preempted soon after entering critical section at T_1. Meanwhile, vCPU1 on pCPU1 is spinning to acquire the same lock. However, this action cannot succeed until T_2, at which point vCPU0 gets a chance to run again and releases the lock. Thus, the synchronization latency on vCPU1 is amplified to $T_{\text{lock\_wait}}$, and all such CPU cycles are wasted in useless spinning.

2.4 vCPU-Bal strategy

We attribute the two aforementioned phenomena to the "semantic gap" between the guest VM and the hypervisor. The hypervisor issues an inappropriate scheduling decision because it is not aware of the internal running status of the guest VM. In function-call IPI delay, the hypervisor does not schedule IPI receivers immediately, which leads to long busy waiting time for the IPI sender. In spinlock holder preemption, the hypervisor preempts the vCPU in the critical section regardless of its "urgent" state, leading to performance degradation.

Owing to the characteristics of virtualization, a hypervisor can access only opaque states, and it is difficult for the hypervisor to abstract enough semantic information from the perceived raw data. vCPU-Bal tries to solve this problem via a novel approach[5]. By weakening the hypervisor scheduler's role, vCPU-Bal converts double-scheduling into single-scheduling. vCPU-Bal is able to adjust the number of available vCPUs assigned to the guest VM and make each vCPU occupy one pCPU separately when the entire system is overwhelmed by heavy contention. Hence there is only one runnable vCPU in the hypervisor scheduler's per-CPU run queue, which means that this vCPU gets the chance to run at any time. With vCPU-Bal, IPI receivers can finish processing as soon as possible, making IPI senders pass this barrier quickly, and spinlock acts as it does with native hardware.

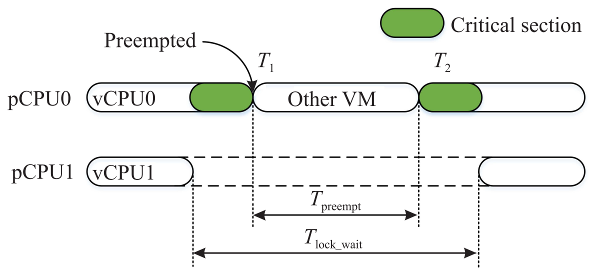


Fig. 4 Illustration on spinlock holder preemption.

vCPU-Bal offers the following advantages:

- By making each vCPU occupy a pCPU in a monopolistic manner, vCPU-Bal can eliminate function-call IPI delay and spinlock holder preemption, thus reducing the CPU time wasted in the busy waiting state.
- Reducing the overhead in performing vCPU context switch, and making it friendly to CPU cache and TLB.
- Avoiding the scalability bottleneck of the guest OS itself when assigned with an excessive number of cores.

3 Design and Implementation

This section describes the design and implementation of FlexCore. Since KVM[2] is released as an open-source kernel module in Linux and is easy to install and debug, we implement FlexCore in KVM. However, vCPU-Bal is not coupled with a specific virtualization platform. Hence, the design challenges we address here should be applicable to other hypervisors, such as Xen[1].

A typical execution of a parallel application in FlexCore involves the following phases:

(1) Before vCPU ballooning: Since the workload on vCPUs is not heavy or there is not much coordination among vCPUs, the guest VM runs with the original number of vCPUs assigned to it.

(2) vCPU ballooning: As more vCPUs become runnable and interact with other vCPUs via spinlock and function-call IPI, the entire system is overwhelmed by busy waiting. Thus, vCPU ballooning is utilized.

(3) After vCPU ballooning: Each vCPU monopolizes one physical core. The performances of spinlock and function-call IPI are restored to their native hardware levels. Most CPU cycles are spent performing real computing work.

To support vCPU-bal, FlexCore needs to address three challenges, viz., detecting vCPU contention, providing efficient VM-hypervisor communication, and supporting vCPU hotplugging.

Figure 5 shows the overall architecture of FlexCore on KVM. FlexCore has three main components: vCPU-Bal control center, vCPU-Bal kernel agent, and the communication channel between them. The following sections describe them in detail.

3.1 vCPU-Bal control center

The vCPU-Bal control center resides in the hypervisor layer. It monitors the running status of all vCPUs in system and issues vCPU ballooning and core binding commands based on vCPU execution statistics.

Intel added a new feature called Pause-Loop Exiting (PLE) in its recently shipped products to support hardware-assisted virtualization[19]. PLE defines two parameters: PLE_Gap, a software-configurable field, as an upper bound on the amount of time between two successive executions of PAUSE in a loop and PLE_Window, donated as an upper bound toward the amount of time a guest is allowed to execute in a PAUSE loop. Once a PAUSE instruction is executed by a guest VM, if the time interval between the last PAUSE instruction exceeds PLE_Gap, the processor regards it as a new waiting loop. If not, the processor increases the counter and triggers a VMExit once the counter reaches PLE_Window. FlexCore relies on PLE to detect vCPU contention.

In both implementations of function-call IPI and spinlock, a PAUSE instruction is appended after one unsuccessful attempt to reduce cache contention overhead. If the failure count exceeds PLE_Window, this situation is detected via the following VMExit. Hence, a high number of PLE times during one period indicates that a massive amount of CPU time is wasted in the busy waiting state caused by double scheduling, and it is necessary to perform vCPU ballooning.

In the FlexCore implementation, we allocate pCPUs in proportion to VM weights in the system. The formula listed below shows the calculating process. In the formula, $N_{\text{vCPU}i}$ refers to the number of vCPU assigned to the i-th guest VM, W_{VM_i} donates the weights of the i-th guest VM, and T_{pCPU} indicates the total number of physical CPUs in the system.

$$N_{\text{vCPU}i} = \frac{W_{\text{VM}_i} \cdot T_{\text{pCPU}}}{\sum\limits_{i=1}^{n} W_{\text{VM}_i}}.$$

Suppose two VMs co-exist in a 12-core machine such that each of them is configured with 12 vCPUs and their weights are 512 and 256, respectively. Then, vCPU-Bal control center will adjust the number of vCPUs from 12 to 8 for the VM with higher weights and from 12 to 4 for the other VM.

To track the system running status, we added a PLE handler in the hypervisor, which bookkeeps PLE times at 1-s intervals in last three seconds. Moreover, the vCPU-Bal control center forks a kernel thread after its initialization. This kernel thread traverses all vCPUs in the system every $T_{\text{check\_interval}}$ seconds and marks them as contended if the PLE times divided by the scheduled times exceed a predefined threshold over the 3 s period. If more than half the vCPUs of a VM in the

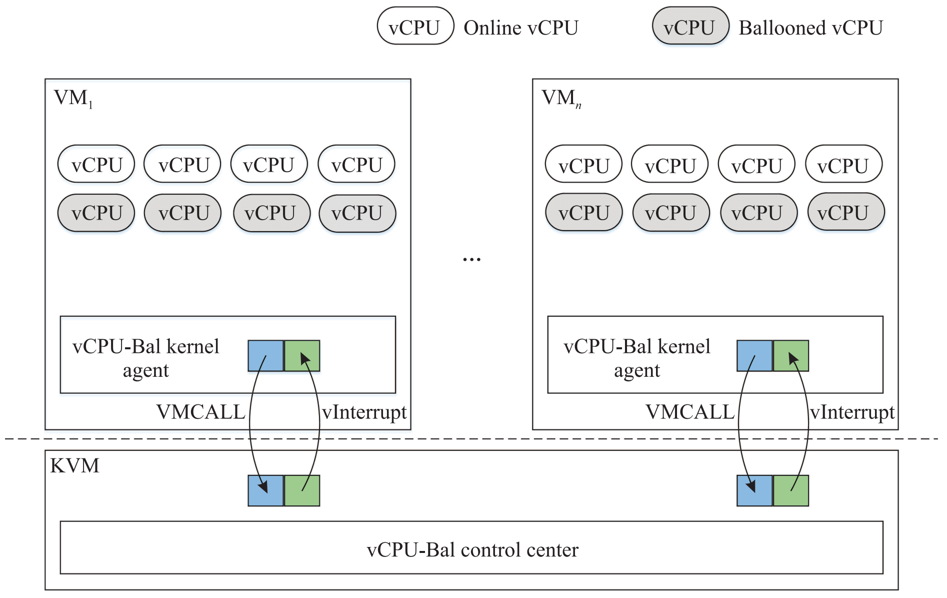


Fig. 5 System architecture of FlexCore on KVM.

system are marked contended, the vCPU-Bal control center will consider the entire system to be in a low performance state. As a result, a virtual interrupt will be delivered to each VM, notifying it to reduce the vCPU number. The entire process is shown in Fig. 6.

3.2 vCPU-Bal kernel agent

The vCPU-Bal kernel agent resides in each VM in the FlexCore system as a guest kernel module. The kernel agent is loaded automatically after the guest OS boots and establishes a communication channel with the vCPU-Bal control center, waiting for its commands and doing the actual work of adjusting the number of vCPUs.

Several challenges are encountered when implementing this kernel agent. First, many per-CPU states exist in Linux, such as per-CPU variable and per-CPU kernel thread, which must be handled with care. Worse, some subsystems in Linux are closely related to the number of online CPUs, such as IPI broadcast to all other CPUs. Finally, the RCU[20] grace period requires that a context switch occurs on all CPUs so that it can reclaim the stale data released before.

```
PLE_HANDLER(vcpu):
  if (VCPUBal_enabled):
    /* account PLE times in every second */
    cur_time ← gettimeofday()
    inc(vcpu.ple_account[cur_time])
  end if

VCPUBAL_MANAGE_THREAD(void):
  while (VCPUBal_enabled):
    sleep for T_check_interval
    foreach vm in vm_list:
      foreach vcpu in vm:
        /* PLE times close to schedule count during last 3
           seconds indicates severe contention in this vcpu */
        if (vcpu.ple_account > vcpu.sched_account * RATIO):
          vcpu.contended ← TRUE
        end if
      end for
      if (more than half of vcpus in vm is contended):
        goto Balloon
      end if
    end for

    Balloon:
    foreach vm in vm_list:
      calculate N_vCPU assigned to vm
      inject virtual interrupt to vm
    end for
    while (have not received halt from all ballooned vCPUs ):
      cpu_relax()
    end while
    bind remaining vCPUs to physical cores
  end while
```

Fig. 6 Algorithm in vCPU-Bal control center.

Thus far, the vCPU-Bal kernel agent has relied on the CPU hotplug feature in Linux[21], the original target of which is hardware maintenance. In the unplugging case, CPU hotplug first marks an outgoing vCPU offline in the global bitmap, so that a global operation such as IPI broadcast ignores the outgoing vCPU. Then, the kernel agent waits for a grace period to safely reclaim the memory resources released by RCU because there is no valid reference to such stale data yet. Thereafter, the kernel agent parks per-CPU kernel threads originally running in the outgoing vCPU and migrates normal threads in its run queue to other alive vCPUs. Finally, the kernel agent executes callback functions registered by other kernel subsystems and issues a HALT instruction to stop the outgoing vCPU. The HALT instruction triggers a VMExit[19] and is detected by the hypervisor, then, the hypervisor can determine whether vCPU unplugging has succeeded.

In addition, we introduce some optimizations to minimize the time cost of vCPU ballooning. The original interface, viz., "cpu_down", is designed to take offline a single vCPU from the guest VM and receive the outgoing vCPU index as its only parameter. However, in most cases, adjustment to the number of vCPUs is larger than one. Moreover, the execution of vCPU ballooning indicates that the entire system is in a heavily contended state. Therefore, the ballooning process should be accomplished as early as possible. If we apply "cpu_down" serially to all outgoing vCPUs, the whole process takes approximately 7 s for 6 vCPUs in our experiment. Through profiling, it is discovered that handling the vCPU's per-CPU kernel threads requires considerable time (approximately 600 ms for each) because "cpu_down" checks the running status of each kernel thread repeatedly until all threads become quiescent (no deferred task remains). Actually, the status checking operation can be parallelized on all outgoing vCPUs because most per-CPU kernel threads are created for deferred task processing and no new task will be delivered because the vCPU is dying. In our implementation, upon receiving the vCPU ballooning command, the kernel agent sends a function-call IPI to all outgoing vCPUs in advance, ordering them to park kernel threads. With this strategy applied, the time cost is reduced to 3.5 s under the same configuration. In theory, the grace period waiting can be parallelized as well. This involves massive modifications to RCU subsystem, but reduces the total ballooning time marginally, as it makes up only a small part of the total ballooning time. Therefore, this challenge will be dealt with in our future work.

vCPU-Bal kernel agent prefers unplugging vCPUs with higher index values. Suppose 12 vCPUs exist in a VM with indexes ranging from 0 to 11, and the control center needs to reduce the vCPU number to 6. In this case, the kernel agent will execute the above process on vCPU11 to vCPU6 in sequence.

Modification to Qemu[22] to accommodate vCPU ballooning is necessary as well because Qemu is closely coupled with guest VMs. Qemu emulates all IO peripherals for guest VMs, and each vCPU stems from a user-space thread forked by Qemu. It must be guaranteed that device interrupts are not delivered to outgoing vCPUs by mistake. By exposing vCPU ballooning information to Qemu, we redirect device interrupts to active vCPUs. Once the HALT instruction is detected in one outgoing vCPU, its corresponding Qemu thread will be suspended on a conditional variable.

3.3 Communication channel

vCPU-Bal control center and vCPU-Bal kernel agent exchange data bi-directionally via the communication channel between them.

In many cases, data exchange is necessary. vCPU-Bal control center needs to tell vCPU-Bal kernel agent when to perform vCPU ballooning and the vCPU number the guest VM should adjust to. vCPU-Bal kernel agent is required to report ballooning progress and deliver debugging information to the control center.

In FlexCore, a XenStore-like[1] method is adopted to achieve this. We add a virtual PCI peripheral in Qemu, and each guest VM loads its virtual driver. During driver initialization, an interrupt vector is reserved, as are the two memory pages used as shared buffer. After initialization, the vector number and buffer page address are passed to vCPU-Bal control center using VMCALL[19] with special parameters. Finally, the control center maps these buffer pages into its own memory address space.

If vCPU-Bal control center wants to deliver messages to one vCPU-Bal kernel agent, the message data is serialized into the first buffer page, and a virtual interrupt is injected for notification. Given that it is impossible to unplug vCPU0 for each VM, the virtual interrupt will always be directed to vCPU0. In contrast, the kernel agent can place its message in the second buffer page and notify the control center using VMCALL.

4 Evaluation

4.1 Configuration

Performance evaluation is conducted on a Dell R810 machine with two Intel Xeon E7-4807 6-core sockets and 32 GB memory. To get rid of weird results caused by hyper-threading, we disable it during evaluation.

The FlexCore system is built on KVM. The kernel version in both the hypervisor and the guest VMs is 3.13.3, and the Qemu version is 1.7.0. At the outset, each VM is configured with 12 vCPUs and 8 GB memory, with the same weight.

PARSEC[12] is a benchmark suite consisting of several parallel applications, and we choose dedup, vips, swaptions, and streamcluster from the suite as test cases. These four applications request services from OS subsystems frequently during running and are expected to reflect running states under real workloads. In the evaluation, the thread number of each application is set to 12, and the native input set is used for testing. To reduce the effect of disk I/O, the entire input set is pre-loaded into memory.

The two configurations used in the evaluation are as follows:

(1) 2VM case: Two guest VMs run concurrently with 24 vCPUs in total. The default scheduler is used in the hypervisor, and the guest OS is unmodified. The PARSEC applications start running at the same time in the guest VMs.

(2) 2VM + FlexCore case: This is almost the same as the 2VM case, except that the vCPU-Bal scheduling strategy is used in the hypervisor and the kernel agent is loaded in every guest VM. More specifically, PLE_Gap is set to 128, and PLE_Window is set to 4096.

4.2 Comparison on execution time

Figure 7 shows the total running time taken by the four PARSEC applications under two configurations, as well as CPU time spent in the kernel mode. The kernel mode time exceeds the total running time for each tested application. This is mainly because the total running time is measured in wall clock, whereas kernel time is a cumulative value of the time spent in the kernel mode by each vCPU. In FlexCore, the total running times of dedup, vips, swaptions, and streamclusters are reduced by 79.6%, 54.1%, 35.4%, and 42.4% compared with its

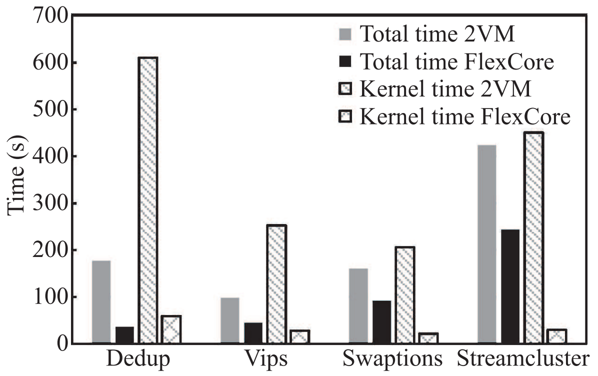


Fig. 7 Comparison on total execution time and kernel time between 2VM case and FlexCore case.

original 2VM case, respectively, resulting in an average reduction of 52.9%. The performance boost is mainly due to the dramatic drops in kernel time in FlexCore, which are 9.7%, 11.1%, 10.7%, and 6.6%, respectively, compared with the 2VM case.

Dedup, one of the tested parallel applications, deduplicates and compresses the content of an input file using a pipeline of communicating threads. Dedup stresses the virtual memory system as well as the virtual file system in the guest OS. It frequently allocates and frees small trunks of memory to and from the process heap, while spinlocks are used to ensure exclusive access to the process' address space and IPIs are used to broadcast changes to the virtual address mapping. In addition, dedup needs to read and write massive amounts of disk data, which involves a large number of synchronization operations. Thus, double scheduling has a fatal influence on dedup performance. With vCPU ballooning, the busy waiting state is relieved and running efficiency is elevated.

4.3 Time breakdown

Figure 8 shows the time breakdown in percentage for function-call IPI and spinlock acquisition. In the 2VM case, all vCPUs contend for available pCPU resources in the system, and the hypervisor scheduler is oblivious to vCPUs in the "urgent" state, such as IPI receiver and spinlock holder. As in dedup, the time percentages in these two columns are 71% and 15%, respectively. With vCPU-Bal in FlexCore, the number of vCPUs in the system is relatively low, and each of them monopolizes one pCPU, thus eliminating the contention overhead. For dedup in FlexCore, the time percentages are reduced to 14% and 4%. For other tested applications as well, the time percentage drops.

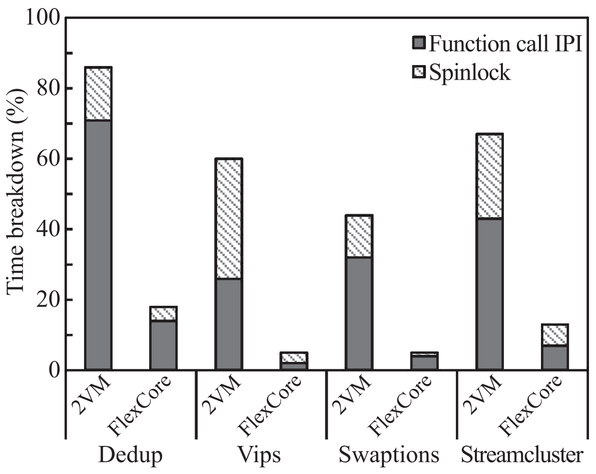


Fig. 8 Comparison on execution time breakdown between 2VM case and FlexCore case.

4.4 Comparison on PLE times

Every time PLE is triggered, a certain amount of CPU time has already been wasted in busy waiting. Thus, the amount of PLE indirectly reflects the system's running efficiency. Table 1 compares the total PLE times during the benchmark under two configurations. As can be inferred from the table, the amount of PLE is reduced significantly.

In fact, the PLE frequency before vCPU ballooning is much higher than that after. For instance, in swaptions, the PLE frequency after vCPU ballooning is approximately 10 kHz, while the PLE frequency before is approximately 500 kHz, which is 50× higher. However, the total amount of PLE is only approximately one fourth of that in the original 2VM case, according to Table 1, because most PLEs are triggered before vCPU ballooning. Moreover, when the vCPU number is reduced, nearly 25% of the total execution time would have elapsed.

4.5 Evaluation conclusion

To conclude from the evaluation, FlexCore can reduce the CPU time wasted in unnecessary busy waiting by

Table 1 Reduction on PLE times in FlexCore.

Application	PLE in 2VM (10^7)	PLE in FlexCore (10^6)	Reduction (%)
Dedup	9.52*	1.90	98
Vips	6.48	9.75	85
Swaptions	8.13	21.90	73
Streamcluster	15.70	36.10	77

Note: *, PLE times are donated in scientific notation.

adjusting the number of vCPUs dynamically and can thus improve guest VM performance by 52.9% on average.

5 Related Work

As virtualization and cloud computing keep evolving, performance issues encountered owing to VM consolidation have attracted considerable attention[23].

Uhlig et al.[24] identified the spinlock holder preemption problem. By exposing lock holding information including remaining critical section length to the hypervisor scheduler, a guest OS gives the hypervisor scheduler a hint. If a vCPU continues to hold the spinlock after using up its time slices, the hypervisor scheduler creates an overdraft and delays the preemption. However, it is difficult to estimate the overdraft, and this scheme may delay other services.

To reduce the performance loss caused by spinlock holder preemption, several co-scheduling strategies were introduced[7–9]. These strategies schedule cooperative vCPUs simultaneously to reduce synchronization latency, for example, sibling vCPUs belonging to the same VM. However, they require the static-time-slice assumption and may cause CPU fragmentation and priority inversion.

Kim et al.[25] discussed demand-based co-scheduling. It infers a guest VM's internal running status based on IPI transmission and schedules the IPI sender and receiver with high priority because they are considered to be in the "urgent" state. This scheme can improve IPI transmission performance, but it does not solve spinlock holder preemption. In addition, priority inversion cannot be eliminated.

Unlike previous solutions, FlexCore tackles the root cause of the problem. By adjusting the number of vCPUs dynamically during runtime, FlexCore proactively avoids performance degradation caused by double-scheduling and does not introduce new problems.

6 Conclusions and Future Work

This study aimed to eliminate the performance degradation caused by double-scheduling. We first identified function-call IPI delay and spinlock holder preemption as the two main causes that lead vCPUs entering long and meaningless busy waiting states. Unlike traditional approaches, FlexCore proactively avoids contention by reducing the number of vCPUs. Our evaluation shows that the average performance improvement for PARSEC applications is approximately 52.9%.

The adjustment of vCPU number is triggered by a comparison between PLE times and scheduled times. Before vCPU ballooning is performed, a certain amount of CPU time has already elapsed. In future, we intend to determine the optimal moment for vCPU ballooning as well as allocate vCPUs among VMs more flexibly.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, kvm: The Linux virtual machine monitor, in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[3] C. Xu, Y. Bai, and C. Luo, Performance evaluation of parallel programming in virtual machine environment, in *Network and Parallel Computing, 2009. NPC'09. Sixth IFIP International Conference on. IEEE*, 2009, pp. 140–147.

[4] H. Lv, Y. Dong, J. Duan, and K. Tian, Virtualization challenges: A view from server consolidation perspective, *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 15–26, 2012.

[5] X. Song, J. Shi, H. Chen, and B. Zang, Schedule processes, not VCPUs, in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, 2013.

[6] P. M. Chen and B. D. Noble, When virtual is better than real operating system relocation to virtual machines, in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on. IEEE*, 2001, pp. 133–138.

[7] 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*, 2009, pp. 111–120.

[8] O. Sukwong and H. S. Kim, Is co-scheduling too expensive for SMP VMs? in *Proceedings of the Sixth Conference on Computer Systems*, 2011, pp. 257–272.

[9] Y. Bai, C. Xu, and Z. Li, Task-aware based co-scheduling for virtual machine system, in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 181–188.

[10] C. A. Waldspurger, Memory resource management in VMware ESX server, *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[11] The CPU Scheduler in VMware ESX 4.1, http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf, 2014.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[13] Linux profiling with Perf, http://lwn.net/Articles/487018/, 2014 .

[14] lockstat in linux kernel, http://lwn.net/Articles/252835/, 2014.

[15] P. Hammalund, J. Crossland, S. Kaushik, and A. Aggarwal, Inter-processor interrupts, U.S. Patent Application 10/631,522, 2003.

[16] The Linux Kernel Archives, https://www.kernel.org/, 2014.

[17] R. Liu, H. Zhang, and H. Chen, Scalable read-mostly synchronization using passive reader-writer locks, in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX Association, 2014, pp. 219–230.

[18] T. E. Anderson, The performance of spin lock alternatives for shared-money multiprocessors, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 1, pp. 6–16, 1990.

[19] *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3A: System Programming Guide*, 2013.

[20] P. E. McKenney and J. D. Slingwine, Read-copy update: Using execution history to solve concurrency problems, in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

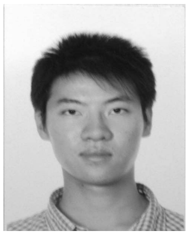
[21] Z. Mwaikambo, A. Raj, R. Russell, and J. Schopp, Linux kernel hotplug CPU support, presented in Linux Symposium, 2004.

[22] F. Bellard, QEMU, a fast and portable dynamic translator, in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

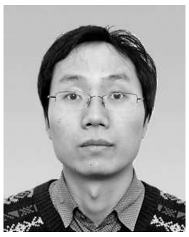
[23] J. Rao and X. Zhou, Towards fair and efficient SMP virtual machine scheduling, in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 273–286.

[24] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, Towards scalable multiprocessor virtual machines, in *Virtual Machine Research and Technology Symposium*, 2004, pp. 43–56.

[25] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, Demand-based coordinated scheduling for smp vms, in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 369–380.



Tianxiang Miao is a master candidate in Shanghai Jiao Tong University. He received his BS degree from NanJing University in 2012. His research interests include full system virtualization and operating system.



Haibo Chen is now a professor in Shanghai Jiao Tong University. He received his BS and PhD degrees in computer science, both from Fudan University in 2004 and 2009, respectively. He seeks solutions to improve the dependability and scalability of computer systems.