Hybrid Co-Scheduling Optimizations for Concurrent Applications in Virtualized Environments

Yulong Yu*, Yuxin Wang†, He Guo*, Xubin He‡
* School of Software
Dalian University of Technology, Dalian, China
† School of Computer Science and Technology
Dalian University of Technology, Dalian, China
‡ Department of Electrical and Computer Engineering
Virginia Commonwealth University, Richmond, VA, USA

Abstract—**Concurrent applications in virtualized environments (VE) have some problems such as Lock Holder Preemption (LHP). Hybrid Co-scheduling is an effective approach to address such problems of concurrent applications in VE. However, the contention and exclusiveness in hybrid co-scheduling, especially when multiple concurrent domains reside in a virtualization system, cause a very serious performance degradation and unfairness. To alleviate the contention meanwhile keeping the advantages of hybrid co-scheduling, we propose two optimization schemes named Partial Co-Scheduling (PCS) and Boost Co-Scheduling (BCS) using finer space granularity. Instead of raising co-scheduling signals for all online CPUs, PCS scheme raises the co-scheduling signals only for the necessary CPUs, while the operations of other CPUs are not affected. BCS scheme boosts the priorities for co-scheduled virtual CPUs (VCPUs) to induce the scheduler to pick the appropriate VCPUs when co-scheduling. We implement both PCS and BCS into Credit Scheduler in Xen 4.0.1, and evaluate their performance compared with original hybrid co-scheduling and co-descheduling under different configurations. The experiment results show that our proposed schemes effectively alleviate the CPU run-time contention and achieve better performance and fairness than existing hybrid co-scheduling approaches.**

Index Terms—**Concurrent Application; Hybrid Co-scheduling; Credit Scheduler; Virtualization; Partial Co-Scheduling; Boost Co-Scheduling**

I. Introduction

The virtualization technology resurges because of the performance enhancement in current personal computers, which brings the possibility to consolidate a large cluster of services into a single server or a small cluster. The total cost reduction makes the virtualization an attractive solution to resource sharing.

With the increased popularity of multi-core processors, more and more applications tend to parallelism for higher performance. The parallel programming tools (such as MPI [1], PVM [2] and OpenMP [3]) help programmers efficiently write concurrent applications. In concurrent applications, multiple threads run simultaneously on different CPU cores to handle their own portions of the task. It is inevitable that these threads need communicate and synchronize to exchange signals and data with each other. The state of the art solutions for synchronization are using locks and semaphores. In a non virtualized environment, all CPU cores in a machine are physical and run online simultaneously, so the locks held by some CPUs will be released soon, and a lock requester is aware of lock releasing as soon as possible. However, this prerequisite that all CPU cores are online at the same time is no longer tenable in virtualized environments (VE), because the CPU cores in a virtual machine (VM) become virtual CPUs (VCPUs), which have to share the physical CPU running time with other VCPUs. The scheduler in the virtual machine monitor (VMM) may schedule a VCPU which is waiting for a lock held by another preempted VCPU or a synchronization signal from other preempted VCPUs, thus this picked VCPU just wastes its time slice on meaningless spinning, seriously affects the system performance.

Some work have been proposed for the lock-holder preemption (LHP) problem and the synchronization problems in VE, such as lock-aware delay preemption [4], spin yield [5], [6], co-scheduling [7], [8] and gang-scheduling [9], [10] (which will be discussed in more detail in Section VII). Co-scheudling is a scheduling scheme that enforces concurrent threads to be scheduled on respective CPU cores simultaneously to avoid the meaningless spinning. Weng et al [8] propose a hybrid co-scheduling framework in VE as an optimization scheme to co-scheduling. Hybrid co-scheduling allows non-concurrent domains to reside in a system to achieve higher CPU utilization by filling the co-scheduling gaps with non-concurrent VCPUs. Some other work on co-scheduling include co-descheduling [6], [11], approximate co-scheduling [6], task-aware co-scheduling [12], [13] and etc. Comparied with other schemes for concurrent applications in VE, co-scheduling have advantages in four aspects, i.e. 1) co-scheduling keeps the prerequisite in native system; 2) no concurrent or synchronization detection is needed; 3) co-scheduling is orthogonal to underlying scheduler in VMM; 4) co-scheduling is not intrusive to guest OS, and easy to implement. In addition, hybrid co-scheduling achieves higher CPU utilization than traditional co-scheduling. Because of these advantages, hybrid co-scheduling is widely used to improve performance for concurrent applications in VE. However, hybrid co-scheduling still suffers from exclusiveness and contention among domains, especially when

there are more than one concurrent domains in a virtualized system. They cause unsatisfactory CPU utilization and many scheduling fragments, which seriously affect the performance of concurrent domains. Our experimental results show that the performance may drop by 50% on average in hybrid co-scheduling when there are two concurrent domains in a system, much slower than non-co-scheduling underlying scheduler.

To address these problems in hybrid co-scheduling, we propose two optimizations using finer space granularity named Partial Co-Scheduling (PCS) and Boost Co-Scheduling (BCS). Both optimizations aim to reduce exclusiveness and contention among CPUs. PCS allows multiple domains to be co-scheduled concurrently by only raising the co-scheduling signals to the CPUs which contain the corresponding co-scheduled VCPU, while other CPUs remain untouched. PCS co-schedules the concurrent VCPUs in time and space precisely, at the cost of increased complexity to pick next VCPU. In many applications, such time precision is unnecessary, therefore from another perspective, BCS is proposed to achieve finer space granularity without precisely co-scheduling in time. Quoting the idea in BOOST mechanism in Credit Scheduler in Xen, BCS promotes the priority of corresponding co-scheduled VCPUs instead of co-scheduling signals to induce the scheduler to pick the appropriate VCPUs for co-scheduling. BCS exchanges precise time alignment to further alleviate the scheduling fragments. It can achieve good performance for most concurrent applications, but for some applications which need strict synchronized, such as MPI cross domain concurrent applications, BCS cannot achieve satisfactory performance that PCS achieves.

We consider that our work on PCS and BCS have the following contributions. 1) Our proposed PCS and BCS not only maintain the advantages of hybrid co-scheduling, but also improve the performance and fairness compared to hybrid co-scheduling when multiple concurrent domains reside in a system; 2) PCS and BCS alleviate exclusiveness and contention among concurrent domains, support multiple concurrent domains to be co-scheduled simultaneously, and achieve higher CPU utilization than hybrid co-scheduling; 3) BCS uses a voluntary approach, which takes advantage of the underlying scheduler, simplifies the implementation, and improves code reliability. We implement prototypes of both optimizations in Credit Scheduler [14] in Xen 4.0.1 [15], [16], and evaluate PCS and BCS and compare them with hybrid co-scheduling [8] and co-descheduling [6] to demonstrate the effectiveness of PCS and BCS. The experimental results show that PCS and BCS maintain the performance of hybrid co-scheduling when one concurrent domain resides in a system. While, when two concurrent domains reside in a system, our performance gain compared to non-co-scheduling underlying scheduler (Credit Scheduler in our case) is about 25% on average; moreover PCS and BCS double the performance compared to existing hybrid co-scheduling and co-descheduling.

The rest of this paper is organized as follows. We emphasize the motivation of our work in Section II, where we analyze the problems of concurrent applications in VE and the problems of hybrid co-scheduling. Some background materials including Xen and its default Credit Scheduler are introduced in Section III. Our proposed optimizations PCS and BCS are introduced in Section IV and Section V. The experiments and measurement results are given in Section VI. We discuss the related work in Section VII. Finally, our conclusions and future work are given in Section VIII.

II. Motivation

We first analyze the problems of concurrent applications in current mainline schedulers in VMMs, and present a scenario of the problem to verify our analysis. Then we introduce the hybrid co-scheduling and point out the problems in existing hybrid co-scheduling schemes, which motivate us to propose our optimizations.

A. Problems in Concurrent

Current schedulers in VMMs have been focusing on throughput and fairness. They operate on each CPU independently, and may neglect the cooperation among the VCPUs. Thus, concurrent applications such as MPI applications which contain a large quantities of cooperative operations, such as synchronizing signal, suffer from performance degradation because of such lack of cooperation concern in scheduler. Further more, some prerequisites are no longer tenable while shifting from native environment to VE, e.g. VCPUs in a domain are not always online at the same time. Thus, a scheduler might schedule a VCPU which is waiting for synchronizing signal from or a lock held by a preempted VCPU.

Hybrid co-scheduling is an efficient strategy for concurrent domains in VMM. We summarize the advantages of hybrid co-scheduling as follow.

1) *Keeping the prerequisite in a native environment.* The co-scheduling claims that the VCPUs in concurrent domain are scheduled concurrently, which maintains the prerequisite in native environments for concurrent applications that all concurrent CPU cores (they become VCPUs in VE) run at the same time.
2) *No semantic detection requirement.* Because hybrid co-scheduling keeps the prerequisite in native environments, the schemes in native environments remain valid with hybrid co-scheduling, and thus it is unaware of guest OS semantics.
3) *Orthogonal to underlying scheduler.* The hybrid co-scheduling does not require special approaches to guarantee the fairness or isolation, so the hybrid co-scheduling can be deployed on most of underlying schedulers.
4) *Transparent to guest OS.* For similar reason, we need not modify guest OS to integrate hybrid co-scheduling.

Additionally, hybrid co-scheduling allows non-concurrent VCPUs in a system co-exist with concurrent VCPUs. When a co-scheduling is taken, the CPUs which run queues do not contain a co-scheduled VCPU will pick a non-concurrent VCPU to fill the time slice. This approach improves CPU

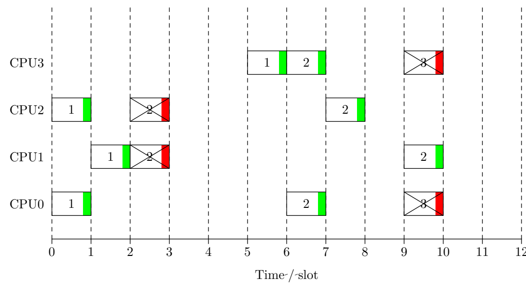


Fig. 1. The scenario of non-co-scheduling. The wait is occurred on CPU1 and CPU2 at 2nd time slot, and on CPU 0 and CPU3 at 9th time slot.

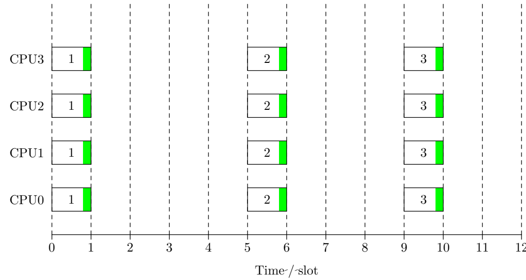


Fig. 2. The scenario of co-scheduling. No idle slots.

utilization, and achieves higher throughput than traditional co-scheduling.

To further understand the problem, we consider the following scenario. Assuming that there is a concurrent domain with 4 VCPUs running on a physical machine with 4 CPU cores. The four VCPUs are evenly distributed on these CPUs, and each contains a concurrent thread. All threads synchronize with each other at the end of each time slot. If a VCPU has been scheduled, the time slice it gains is just one time slot. The scenario of the non-co-scheduling is shown in Fig. 1. It is obvious that CPU1 and CPU2 just waste time at Slot 3 because they are spinning to wait the never-arriving synchronizing signal from CPU3. In the first ten time slots, although each VCPU is scheduled three times, only two steps of task are finished. One third of the time is wasted on the worthless spinning. In contrast, in a co-scheduling scenario which is shown in Fig. 2, none of the time slots are wasted, because the VCPUs in concurrent domain are scheduled simultaneously.

B. Problems in Hybrid Co-scheduling

Hybrid co-scheduling is an effective scheme for concurrent application scheduling in VE, and achieves better performance than underlying scheduler when only one concurrent domain resides in a system. However there are still some problems caused by exclusiveness and contention in hybrid co-scheduling. Especially when there are multiple concurrent domains in a virtualized system, the problems are obvious, edge out the advantages, and may even cause a serious performance degradation. We summary them from several aspects below.

1) *Unsatisfactory CPU Utilization.* Although hybrid co-scheduling allows concurrent domains and non-concurrent domains to co-exist in a system simultaneously, which improves CPU utilization, it still inherits exclusiveness in some extent. Hybrid co-scheduling cannot co-schedule two concurrent domains at the same time. When a concurrent domain is co-scheduled, the CPUs which run-queue do not contain corresponding VCPUs have to schedule either a non-concurrent VCPU or remain idle. Thus, it causes an unsatisfactory CPU utilization and unfairness, especially in many-core processors, when multiple concurrent domains reside in a system.
2) *Scheduling Fragments.* Hybrid co-scheduling is mandatory. When a CPU launches co-scheduling, it raises the co-scheduling signals to all CPUs in the system, and forces other CPUs to reschedule regardless their own scheduling status. Thus, more scheduling fragments are produced than original underlying scheduler, and additionally more CPU time is consumed on the scheduling operation.
3) *Scheduling Contention.* Due to the exclusiveness in hybrid co-scheduling, only one concurrent domain can be co-scheduled at a time. When there are multiple concurrent domains in a system, the contention among them occurs that when one concurrent VCPU launches co-scheduling, the running concurrent VCPUs are pushed out from running state mandatorily regardless when they are picked. These phenomena raises so many scheduling signals and causes so many scheduling operations, and thus seriously affects the performance when there are multiple concurrent domains.

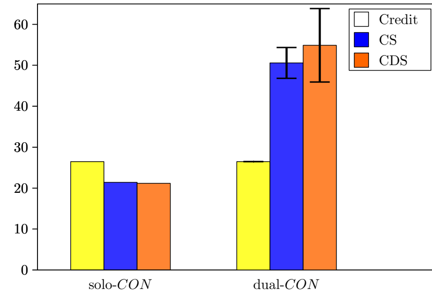


Fig. 3. experimental results of multiple domains running under co-scheduling (CS) and co-descheduling (CDS) schemes. In solo-CON configuration, both CS and CDS achieves about 25% higher performance, while in dual-CON configuration, they degrade the performance about 50%. The approximately 27% execution time difference in CS and CDS display their unfairness.

To visualize the effect of the problem, we conduct an experiment and the results are presented in Fig. 3. We run LU kernel in SPLASH2 benchmark [17], [18] on one and two concurrent domains (as solo- and dual-CON configurations in figure) using hybrid co-scheduling (CS) and co-descheduling (CDS) implemented in Credit Scheduler in Xen 4.0.1, and measure their execution time. If one concurrent domain resides in the system (i.e. solo-CON configuration), the CS and CDS speedup is about 26% on average. However, when two concurrent domains are in the system (i.e. dual-CON configuration),

the performance degrades seriously. The execution time is about 50s in CS and CDS, which is about twice the Credit Scheduler on average. The fairness is also not satisfactory, the execution time difference between the two concurrent domains is about 27% on average in CS and CDS.

In hybrid co-scheduling, each co-scheduling operation is global, affects all online CPUs, regardless whether the run queues of CPUs contain the corresponding concurrent VCPUs. We observe that the current hybrid co-scheduling is coarse space granularity. Therefore we consider hybrid co-scheduling optimization schemes in finer space granularity. Based on our above analysis, we propose two hybrid co-scheduling optimizations for finer space granularity to alleviate the exclusiveness and contention: PCS and BCS, which are discussed in detail Sections IV and V.

III. Xen and Credit Scheduler

The implementation of our proposed optimizations and experiments are conducted in Xen virtualization platform and Credit Scheduler. Here we briefly review Xen and Credit Scheduler.

A. Xen

Xen [16] is an open source virtualization platform supporting paravirtualization, which uses a modified guest OS kernel to coordinate with VMM. Xen adopts the hypervisor-domain architecture. The hypervisor runs on bare hardware, and administrators can use a privileged domain named Domain-0 to manage the virtualization system. Xen employs the frontend-backend model to handle the I/O operation in domains. The native hardware drivers are installed in the driver domain, which also keeps the backend driver. When a domain requests I/O operation, its frontend driver notifies the backend driver, which performs the actual I/O operations. Due to the paravirtualization and the hypervisor-domain architecture, the guest OS in Xen can achieve high performance close to a native OS.

B. Credit Scheduler

There are several schedulers in Xen's history. Credit Scheduler [14] is implementedas the default scheduler in current Xen. Credit Scheduler depicts the domainwith *weight* and *cap*. The credit value of each VCPU is used to ensure the fairness among the domains. Every 30ms, Credit Scheduler adjusts the credit for each VCPU according to the weight of its domain. The running VCPU uses a constant credit every 10ms. There are two basic priorities in Credit Scheduler: *UNDER* if a VCPU keeps a positive credit, and *OVER* if a VCPUs credit becomes negative. The VCPUs are sorted in each run queue of CPU according to their priorities (not credit). Credit Scheduler always picks the first VCPU from the queue to schedule I/O. The cap value is used to limit the execution time of VCPUs in Non-Work-Conserving (NWC) Mode. To quickly respond latency sensitive domains, Credit Scheduler introduces the highest *BOOST* priority. When an UNDER VCPU is woken from blocking, the schedule will temporarily BOOST it for running as soon as possible.

IV. Partial Co-Scheduling

To address the problems mentioned above, we proposed two optimizations for hybrid co-scheduling named Partial Co-scheduling (PCS) and Boost Co-scheduling (BCS). We introduce PCS in this section and BCS is discussed in next section.

A. Design of PCS

In our previous analysis, the main problems in hybrid co-scheduling are exclusiveness and contention. If we find out a way of finer space granularity that when one concurrent domain is co-scheduled, only the involved CPUs are distrubed to co-schedule the corresponding VCPUs, the remaining online CPUs still perform their ongoing tasks, we can achieve finer space granularity, and thus exclusiveness and contention can be alleviated or even removed. Our proposed PCS just follows this idea. PCS scheme features in three main aspects. First, we use a per-CPU array instead of a global variable to record the current co-scheduled domains. Second, we introduce a CPU mask variable in each concurrent domain to record the VCPU distribution on CPUs for precise *partial* co-scheduling. Third, we introduce global CPU mask variables to record the system wide co-scheduling states. Thus, the co-scheduling operations are migrated from a global system operation to an operation in *partial* systems.

PCS has the following advantages. First, this fine-grainularity co-scheduling scheme only co-schedules the necessary online CPUs precisely, and the rest CPUs can launch another co-scheduling at the same time. Thus, not only the advantages of hybrid co-scheduling are maintained, but also contention between multiple concurrent domains are alleviated. Second, two or more concurrent domains can be co-scheduled simultaneously in PCS, which achieves higher CPU utilization and better fairness than hybrid co-scheduling. Third, the scheduling fragments will be decreased, because less scheduling signals are propagated around the whole system in such finer apace granularity. Fourth, the tolerance is improved since the configuration mistakes by administrators for co-scheduling are limited in a subset of online CPUs in PCS instead of the whole system. These benefits are supported in our experiments as shown in Section VI.

B. Implementation

PCS can be implemented with underlying credit scheduler in a Xen system. First, we use a per-CPU array $CoDom$ to record the current co-scheduled domain instead of a global variable. Thus, PCS changes the co-scheduling operation is a *partial* operation, and each online CPU may co-schedule a distinct concurrent VCPU at the same time. Second, to precisely co-schedule the necessary CPUs when a co-scheduling operation is launched, we introduce a per-domain CPU mask variable $CoCPUs$ to record the VCPU distribution on online CPUs. Each $CoCPUs$ is configured as soon as a domain becomes to concurrent. Moreover, in order to make PCS working continuously, We use two global CPU mask variables, $CoWorkers$ and $CoPartners$, to record the state of the

Algorithm 1 Partial Co-scheduling Kernel

Put the current VCPU back to $RUNQ(cur_cpu)$;
Let $Next = NIL$;
if this_cpu($CoDom$) **then**
  for all $Iter \in RUNQ(cur_cpu)$ **do**
    if $Iter.Dom = CoDom$ **then**
      Let $Next = Iter$;
      break;
    end if
  end for
  if $Next = NIL$ **then**
    Pick $Next$ the first HIT VCPU in $RUNQ(cur_cpu)$;
  end if
  cpu_clear($CoWorkers$, cur_cpu);
  Let $CoCPUs$ =this_cpu($CoDom$).$CoCPUs$;
  if cpus_empty(cpus_and($CoWorkers$, $CoCPUs$)) **then**
    cpus_andnot($CoPartners$, $CoCPUs$);
  end if
  Let this_cpu($CoDom$)= NIL;
else
  Pick $Next$ using underlying scheduler rules;
  if $Next.Dom.Type = CON$ **then**
    Let $CoCPUs = Next.Dom.CoCPUs$;
    cpu_clear($CoCPUs$, cur_cpu);
    if cpus_empty(cpus_and($CoPartners$,$CoCPUs$)) **then**
      for all $cpu \in CoCPUs$ **do**
        Let per_cpu($CoDom$, cpu)= $Next.Dom$;
      end for
      cpus_or($CoWorkers$,$CoCPUs$);
      cpus_or($CoPartners$,$CoCPUs$);
      cpu_raise($CoCPUs$, $SCHED_SIG$);
    else
      Pick $Next$ the first HIT domain's VCPU in $RUNQ(cur_cpu)$;
    end if
  end if
end if
return $Next$;

current co-scheduling participant CPUs. $CoWorkers$ records the CPUs which are pending a request to co-schedule. When any CPU is scheduled, the corresponding bit in $CoWorkers$ will be cleared. $CoPartners$ records the CPUs which are in VCPU distribution of the current co-scheduled domains. When a co-scheduling operation for a domain is finished, i.e. no bits corresponding to the domain's $CoCPUs$ is set in $CoWorkers$. All these $CoCPUs$'s bits will be cleared from $CoPartners$. When a domain's co-scheduling is launched, the bits recorded in the domain's $CoCPUs$ are set in both $CoWorkers$ and $CoPartners$. Thus, a concurrent domain can be co-scheduled, only if no bits corresponding to the domain's $CoCPUs$ are set in $CoPartners$.

The pseudo-code of PCS scheme is show as Alg. 1. We use $Type$ to distinguish whether a domain is concurrent with two alternative values CON and HIT representing *Con*current and *Hi*gh *T*hroughput respectively. The outer "if" condition statement is used to judge whether there is a pending co-scheduling operation for this CPU. The "if" branch describes the situation when there is a pending co-scheduling operation. PCS first lookup and schedule the co-scheduled VCPU according to the domain recorded in $CoDom$. Then, it clears the corresponding bit in $CoWorkers$ for current CPU, and checks whether all the VCPUs in this domain has been co-scheduled. If so, the bits corresponding to $CoCPUs$ in $CoPartners$ will be clear to declare that this co-scheduling operation is finished. For "else" branch, the scheduler picks the next VCPU as underlying scheduler rules in advance. If the selected VCPU belongs to a concurrent domain, the algorithm check whether a co-scheduling operation can be lauched for this domain. If the co-scheduling is not allowed, just pick a VCPU from HIT domains as $Next$ instead; else, set the corresponding bits in $CoWorkers$ and $CoParters$ according to the concurrent domain's $CoCPUs$. Then co-scheduling signal is sent to CPUs in the $CoCPUs$ to launch a co-scheduling operation. It is obvious that PCS is orthogonal to the underlying scheduler, which means that we can plant PCS into any scheduler such as the Xen's default Credit Scheduler.

Furthermore, the VCPUs in concurrent domains are scattered among the online CPUs for effective co-scheduling operation. In our implementation, the VCPU scatter is done as soon as a domain becomes concurrent. The algorithm iterates all VCPUs in the new configured concurrent domain, and puts each VCPUs into the run queue of a CPU, such that there are least VCPUs of this domain in the run queue of the CPU. If there are more than one CPUs which satisfy the above condition, the algorithm selects the one which contains least concurrent VCPUs.

V. BOOST CO-SCHEDULING

A. Design of BCS

PCS complexes the hybrid co-scheduling to guarantee the space and time preciseness, but sometimes the system does not need such precise time edge alignment among the co-scheduled VCPUs. To reduce the algorithm complexity, we propose BCS using a voluntary approach which trades time preciseness for lower complexity. Different from PCS, BCS induce the scheduler to pick the appropriate concurrent VCPUs, through *boosting* the priorities for corresponding VCPUs, instead of raising a mandatory re-scheduling signal. When BCS finds that the next scheduled VCPU belongs to a concurrent domain, it iterates all the VCPUs of this domain, and *boost* their priorities to the highest level. Then the scheduler picks and schedules these boosted concurrent VCPUs on each CPU respectively, and achieves the similar effect as PCS.

Obviously, BCS shares the most important advantages with PCS. They are both fine space granularity, only co-schedule the necessary CPUs and leave the rest CPUs untouched. Furthermore, BCS achieves some more benefits. The most significant feature of voluntary approach is to utilize the original

Algorithm 2 Boost Co-scheduling Kernel

Select $Next$ using the underlying scheduler rules.
if $Next.Pri = COS$ **then**
  Let $Next.Pri = Default_Pri$;
else if $Next.Dom.Type = CON$ **then**
  for all $IterVCPU$ in $Next.Dom.ActiveVCPUs$ **do**
    Let $Next.Dom.Pri = COS$;
  end for
end if
return $Next$;

underlying functions and mechanisms as much as possible. Therefore, BCS is Simple to implement. We just add several lines of codes to update the priorities of co-scheduled VCPUs, reserving all the mechanisms of the underlying scheduler such as fairness and isolation guarantee. Less codes bring safer, more stable and less cost for development and maintenance. Hence BCS finds a balance between co-scheduling and the underlying scheduler. Moreover, due to no more scheduling signals are raised by BCS, the scheduling fragments are less than PCS, as evidenced in our experimental results in Section VI.

While, BCS does not align the co-scheduling time precisely, because no scheduling signal is raised. For some strict concurrent applications such as frequent synchronizing and cross domain concurrent applications, BCS does not perform better than PCS. The experiments results demonstrate that the co-scheduling time edge difference is much smaller than the scheduling period of scheduler, so we assert that in most practical situations, we can ignore the time edge difference.

B. Implementation

We implement BCS into the Xen's default Credit Scheduler. We add a new priority level named COS for co-scheduling. When the scheduler picks a concurrent VCPU as the next VCPU and the priority of this VCPU is not COS, a co-scheduling operation need to be launched. Then, the scheduler iterates and *boosts* all the VCPUs in this concurrent domain to COS. While the VCPU with COS priority is picked by the scheduler, its priority will be fallen back to default priority ($UNDER$ in Credit Scheduler). The pseudo-codes are displayed in Alg. 2.

Similar to PCS, BCS also need scatter the VCPUs of each concurrent domain among online CPUs. We use the same strategy to ensure the VCPU scatter among online CPUs as in PCS.

Besides, a user interface is necessary to configure the type of domains running in system. As we have mentioned in Section IV-B, we add a field to the domain data structure in scheduler named *type*, and provide two alternative constant values HIT and CON for *type*. For HIT domains, the scheduler treats them with original underlying scheduler (Credit Scheduler in our current implementation); and for CON domains, the scheduler treat them with hybrid co-scheduling schemes. We add a parameter *type(-t)* into the *credit-sched* command in *xm*, so the user can configure the type of domains on the fly. When a domain is configured to CON from HIT, the scheduler will call the VCPU scatter module to scatter the VCPUs in the new concurrent domain for the future co-scheduling. We implement this user interface into both BCS and PCS.

VI. Experiment Measurement and Analysis

We conduct experiments and collect scheduler data to verify the effectiveness of our proposed hybrid co-scheduling optimizations, PCS and BCS. In this section, we describe our experiments and analyze the results.

A. Testbed

Our testbed system has a quad-core CPU (Intel Core i5 760, 2.8 GHz) and 4GB RAM (two 2GB DDR3 RAM, 1333Mhz). We use Xen 4.0.1 as VMM, and the operating system installed in Domain-0 is Ubuntu 10.04 Server x64 with Xen patched Linux kernel 2.6.31. We implement our proposed PCS and BCS into the default Credit Scheduler respectively. We also implement the hybrid co-scheduling (CS) and the co-descheduling (CDS) schemes into Credit Scheduler for comparison. Four guest domains run in experiments. Each domain has two VCPU cores and 394MB RAM. The operating system installed in each domain is CentOS 5.5 x64 with Xen patched kernel. Most of our experiments only need one or two guest domains. Thus the rest domains in such experiments just remain idle with HIT type, in order to observe the effect by domains with different types. We use our modified *xm* (described in Section V-B) to configure the type of domains.

B. Benchmarks

We select two parallel performance benchmarks, LU kernel in SPLASH2 [17], [18] and NPB [19], to evaluate PCS and BCS in practice.

1) *LU*: The LU kernel in SPLASH2 is a computational intensive benchmark. The LU kernel split a dense matrix with $N \times N$ array of $B \times B$ blocks into a upper and lower triangle matrix. In LU experiments, we measure the execution time, co-scheduling frequency, as well as time differences among VCPUs in concurrent domain when co-scheduling for BCS. We run LU kernel with 2 processors, and let $N = 4096$ and $B = 16$.
2) *NPB*: NAS Parallel Benchmarks (NPB) are used to evaluate the performance of parallel computing systems, which are mainly derived from computation fluid dynamics (CFD) programs. NPB benchmarks are recognized as the standard indicator of computer performance. We select six benchmarks in NPB for our experiments, which are BT, CG, EP, FT, LU, and MG. We compiled BT in Class A with 1 and 4 processors for single- and dual-CON configurations respectively, FT and LU in Class A with 2 processors, CG, EP and MG in Class B with 2 processors. We measure the execution time for each benchmark under different configurations.

In experiments, we run each benchmark in each scheduling scheme with two configurations respectively, solo-CON

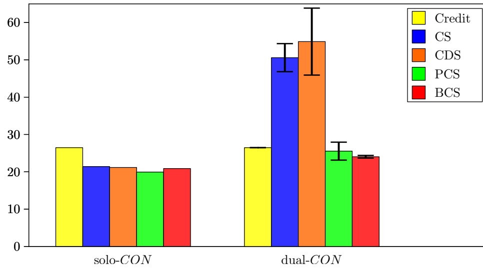


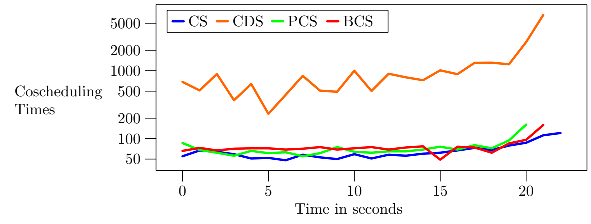
Fig. 4. experimental results of LU benchmark in Execution time. For solo-CON configuration, PCS and BCS keep the higher performance. For dual-CON configuration, PCS and BCS avoids the performance degradation, achieve higher performance than Credit and better fairness than CS and CDS.

configuration and dual-CON configuration. For solo-CON configuration, the benchmarks run on only one CON domain with the rest three idle HIT domains. While, in dual-CON configuration, the benchmarks run on two CON domains concurrently with the rest two idle HIT domains.

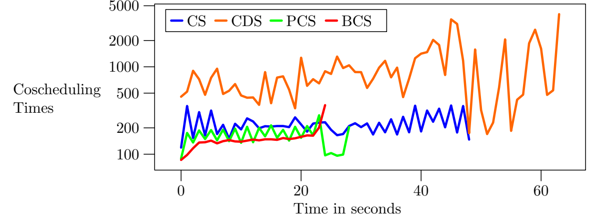
C. Experiment Results

First, we conduct the LU benchmark experiment. We run LU benchmark on the Credit Scheduler, CS, CDS, PCS and BCS schemes respectively. For each scheme, we run LU benchmark with two configurations respectively. Fig. 4 shows the experimental results in execution time for solo- and dual-CON configurations.

LU benchmark is a concurrent CPU-intensive application, and does not have many I/O accesses. From the experimental results, for solo-CON configuration, all the co-scheduling schemes (including CS, CDS, PCS, BCS) achieve better performance than Credit Scheduler. The execution time in CS, CDS, PCS, and BCS reduces from 26.47s to 21.41s, 21.17s, 19.93s, and 20.88s respectively, which is about 26% faster than underlying Credit Scheduler on average. It testifies that the hybrid co-scheduling is an effective method for concurrent applications, and PCS and BCS maintain the advantages of hybrid co-scheduling in performance aspect. In contrast, for dual-CON configuration, neither CS or CDS performs well, the execution times on CS and CDS are 54.34s and 46.82s, 45.91s and 63.85s, almost 1X longer than Credit Scheduler. Moreover, the fairness of CS and CDS is not good enough for dual-CON configuration, especially on CDS. The execution time differences between the two concurrent domains are 7.52s and 17.94s (16.06% and 39.08%) in CS and CDS respectively. Compared to CS and CDS, our optimizations PCS and BCS perform better for dual-CON configuration. Their execution times achieve a reasonable level, which are 23.13s and 25.94s, 24.39s and 23.68s, only about 14% slower than solo-CON configuration because more virtualization cost are needed. Both PCS and BCS are about 8% faster than Credit Scheduler, and about 1X faster than CS and CDS on average. Additionally, PCS and BCS also achieve good fairness between the two concurrent domains, especially in BCS. The execution time differences are 2.81s and 0.71s (12.14% and 2.99%) in PCS and BCS respectively.



(a) Solo-CON configuration



(b) Dual-CON configuration

Fig. 5. Co-scheduling frequency for solo- and dual-CON configurations. Higher co-scheduling frequency show more scheduling fragments and contention exists in system. For solo-CON configuration, the co-scheduling frequency of PCS and BCS are almost the same as CS, while for dual-CON configuration, the co-scheduling frequency of PCS and BCS are lower than CS and CDS, that shows less scheduling fragments and lower contention with PCS and BCS.

We also use LU benchmark to measure the co-scheduling frequency on different co-scheduling strategies. Fig. 5 shows the measurement results for solo- and dual-CON configurations. Fig. 5(a) shows that for solo-CON configuration, the co-scheduling frequencies are nearly the same among CS, PCS, and BCS, approximately $60-70$ /second. It illustrates that PCS and BCS maintain the advantages of hybrid co-scheduling. The co-scheduling frequency of CDS is much higher than other strategies, about $300-1000$ /second on average because of the stricter rule in CDS. Fig. 5(b) shows the results for dual-CON configuration, where CS and CDS display their unstableness because of the contention between concurrent domains. The co-scheduling frequency of CS is about $150-400$ /second, and frequnecy of CDS is about $200-3000$ /second. This increased co-scheduling frequency causes more processor execution time consumed on the scheduling critical path, increasing the hypervisor occupied time, and finally causes the performance degradation in guest system. The two CON domains in CS and CDS must share running time serially, and serious performance degradation occurs for these two domains. The fluctuation makes the system more unstable and unpredictable, especially in CDS. In constrast, the co-scheduling frequency of PCS is about $150-200$ /second on average and the co-scheduling frequency of BCS is about $100-180$ /second on average. Not only the co-scheduling frequency is lower than CS and CDS, but also they are more stable. The lower co-scheduling frequency means less contention and better performance. The stableness means that PCS and BCS are predictable. Furthermore, the two CON domains in PCS and BCS can run in parallel, that brings extra performance benefit. The execution time results in Fig. 4 also support our analysis

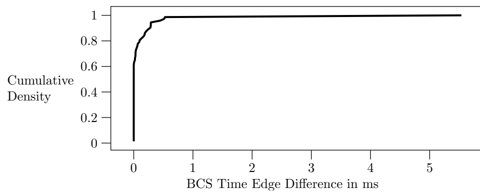


Fig. 6. Cumulative density of BCS time edge difference between two virtual CPUs in concurrent domains. More than 99% of the time edge difference are less than 1ms, so that we can ignore such time edge difference.

from co-scheduling frequency, that PCS and BCS perform better than CS and CDS.

To confirm our conclusion which we have mentioned in Section V-A that the co-scheduling time edge difference among VCPUs in concurrent domain does not affect the co-scheduling features in BCS, we measure the scheduling time edge difference between the two VCPUs in concurrent domain in LU benchmark experiment. Fig. 6 shows the measurement results in cumulative density. It is clear that almost all time differences are less than 1ms. The scheduling period in Xen's Credit Scheduler is 30ms, all the time edge differences in our experiment on BCS are much shorter than the scheduling period. Hence, although the co-schedulings are not precisely aligned, the time difference in BCS is so small that it can be negligible. BCS remains the features of co-scheduling that VCPUs in concurrent domain are scheduled simultaneously in most situations.

Besides SPLASH2 LU benchmark experiment, we conduct the experiments using NPB benchmarks. We select 6 benchmarks in NPB to run on Credit Scheduler and the four co-scheduling schemes, CS, CDS, PCS, and BCS, respectively. We measure their execution time with solo- and dual-CON configurations. The 6 benchmarks selected are BT, CG, EP, FT, LU, and MG. Fig 7 shows the execution time results of the NPB benchmarks.

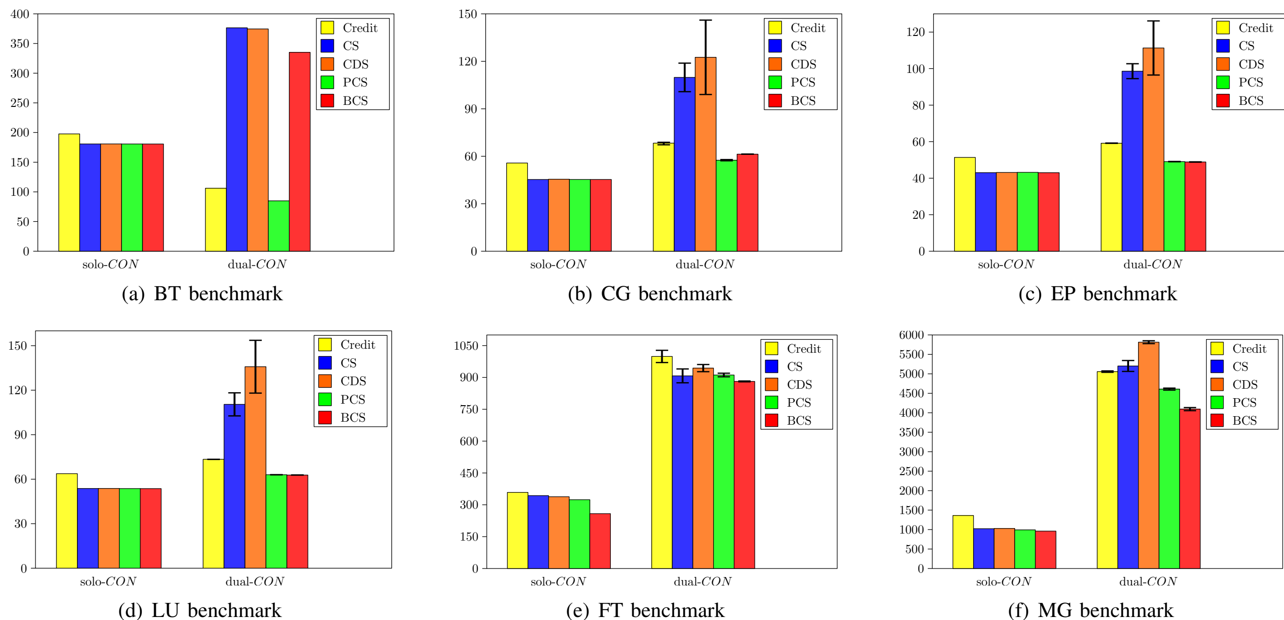
BT [19] is a simulated CFD application to solve 3D compressible Navier-Stokes equations. Fig. 7(a) shows the execution time results of BT and SP respectively. For solo-CON configuration, all the co-scheduling schemes speed up the application. The execution time of BT reduces from 197.67s in Credit Scheduler to about 180s in hybrid co-scheduling schemes which speeds up about 9.4%. However, for dual-CON configuration, CS and CDS performs terrible. The execution time of BT in CS and CDS increases to about 375s, 3.5X to Credit Scheduler 106.13s. In contrast to CS and CDS, PCS achieves a high performance. The execution time of BT in PCS is 84.90s, 25% faster than Credit Scheduler, 3.7X faster than CS and CDS on average. BCS performs worse than PCS because of its imprecise scheduling time alignment. In such cross domain concurrency, the impreciseness is enlarged to the extent that cannot be negligible. The execution time in BCS are 335.11s, which is still better than CS and CDS.

CG [19] is a kernel test in NPB benchmarks, which computes an approximation to the smallest Eigen value of a large sparse unstructured matrix with Conjugate Gradient method. EP [19] generates pairs of Gaussian random deviates as specified by certain scheme. LU [19] is also a simulated CFD application with symmetric successive over-relaxation method. The execution time results of these three benchmarks are displayed in Fig. 7(b), 7(c) and 7(d). These three benchmarks are all CPU-intensive application. For solo-CON configuration, similar to the results of BT, co-scheduling schemes speed up these benchmarks. The execution time of CG, EP, and LU in co-scheduling schemes are about 45s, 43s, and 53s respectively, speed up 23.8%, 19.4%, and 20.2% to Credit Scheduler respectively. However, for dual-CON configuration, the results are different. Performance in CS and CDS degrades seriously. The execution time of CG, EP, LU in CS are about 1.6X, 1.7X and 1.5X to the Credit Scheduler respectively; and the execution time in CDS are about 1.9X, 2.0X, 1.9X to Credit Scheduler respectively. In contrast, PCS and BCS perform better, the execution time of PCS are about 17.5%, 20.4%, and 15.8% faster than Credit Scheduler, about 1.0X, 1.2X, and 0.87X faster than CS and CDS on average, and the execution time of BCS are about 11.4%, 22.9%, and 17.7% faster than Credit Scheduler, about 0.96X, 1.2X, and 0.88X faster than CS and CDS on average respectively. Because contention confuses the steps of scheduling in CS and CDS, the fairness are sacrificed additionally. All these three figures show bad fairness in CS and CDS with dual-CON configuration, the execution time differences between two concurrent domains are 17.8%, 8.6%, 15.1% in CS, and 47.4%, 30.7%, 30.2% in CDS respectively. However, PCS and BCS perform good fairness in these three benchmarks. The time differences are 1.3%, 0.4%, 0.17% in PCS, and 0.03%, 0.3%, 0.12% in BCS respectively, which are much less than in CS and CDS.

FT [19] is a 3D fast Fourier Transform-based spectral method. MG [19] finds the solution of the 3D scalar Poisson equation with V-cycle Multi-Grid method. The execution time results show in Fig. 7(e) and 7(f). Different from the above four benchmarks, FT and MG have more disk accesses, so they are I/O-intensive. The limited I/O access channel is a bottleneck on the performance for dual-CON configuration. The performance for dual-CON configuration is about 2.5X and 4X lower of FT and MG respectively than for solo-CON configuration in any scheduling scheme. Even though in I/O channel contention environment, PCS and BCS benefit the concurrent domain scheduling to some extent. In both FT and MG experiments, PCS and BCS achieve better performance than Credit Scheduler, CS and CDS. The execution time of FT and MG is about 9.1% and 26.1% faster in PCS and BCS than in other scheduling schemes respectively on average. PCS and BCS achieve better fairness than CS and CDS. The execution time difference of FT and MG is about 1.8%, 1.1% in PCS, and 0.4%, 1.9% in BCS respectively.

VII. Related Work

The co-scheduling scheme is proposed by Ousterhout [7] to address the scheduling problems in concurrent system. The traditional co-scheduling suffered from some drawbacks, such



(a) BT benchmark (b) CG benchmark (c) EP benchmark

(d) LU benchmark (e) FT benchmark (f) MG benchmark

Fig. 7. Execution time in seconds of NPB benchmarks. For all these benchmarks, PCS and BCS keep the benefits brought by CS for solo-CON configuration. For dual-CON configuration, PCS speeds up the system and avoids the performance degradation as CS and CDS for all these benchmarks. BCS achieves the same results expect BT which is a cross domain parallel application. Both PCS and BCS achieves better fairness because of their small execution time difference.

as the low CPU utilization, and the exclusion to other schedulers. VMWare Communities [11] introduced co-scheduling into VE and implemented it in VMWare products using stricter co-scheduling strategy called co-descheduling, where the concurrent VCPUs need to not only be scheduled concurrently but also be descheduled concurrently. However, it causes many scheduling fragments and introduces serious contention among domains. Weng et al [8] proposed a hybrid scheduling framework in VE, which combines an underlying high-throughput guarantee scheduler and co-scheduling scheme together. The hybrid scheduling framework allows non-concurrent domains in a system, alleviates exclusiveness of the co-scheduling. But if there are multiple concurrent domains, the contention still exists, which motivates us to propose PCS and BCS. Jiang et al [6] proposed some optimization of scheduler in KVM virtualization environment. In their work, they introduce the idea of voluntary approach, and proposed Approximate Co-scheduling, which motivates us to proposed BCS. While in our BCS the benefits of PCS are achieved. In their work, co-descheduling scheme is also mentioned. Xu and Bai et al [12], [13] proposed an automatic concurrent domain detection method for CPU intensive domains, based on their research on performance evaluation of parallel programming in VE. They implemented hybrid co-scheduling scheme as the underlying scheduling support. Although their work releases the administrator from system configuration, there is no optimization for co-scheduling, and their detection method only works for CPU intensive domains.

There are some work on the LHP problem besides co-scheduling. Uhlig et al [4] proposed a lock-aware scheduler, which schedules VCPUs according to spin lock detection in guest OS. The lock-aware scheduler delay to preempt the determined lock-holder till the lock released. But it introduces a complex computation in lock detection. Additionally, some fault detection will cause a long CPU preemption. Friebel et al [5] based on lock-aware idea proposed the method that prevents unnecessary active waiting instead of the delay preemption. Jiang et al [6] brought the similar idea which called waiting yield method. Neither their work avoids the complex lock detection which is not needed in co-scheduling scheme.

Another strategy for LHP problem is gang-scheduling proposed by Feitelson et al [9], whose idea is very similar to co-scheduling. It schedules related threads simultaneously on different processors based on Ousterhout Matrix. Hence, gang-scheduling is stricter than co-scheduling. Wang et al [20] implemented gang-scheduling into IBM parallel system. Feitelson [10] proposed an optimization using multiple packing sets of processors instead of single packing set. Jette et al [21] analyzed the performance of gang-scheduling systematically. Compared to co-scheduling, gang-scheduling employs Ousterhout Matrix to align the scheduling steps, so it imposes restrictions in practice, and introduces exclusiveness to other scheduling schemes.

There are some work on schedulers in VE for parallel or concurrent applications in multi-processor or cluster system recently. Here we list some of them that we are interested. Somani et al [22] proposed isolation analysis method for schedulers. Govindan et al [23], [24] proposed a communication-aware scheduler which brings more precise CPU time account for I/O operation. Liu et al [25] proposed a method that use dedicated CPU cores to handle the I/O events to alleviate

the frequency of the scheduler tickle and context switching. Chen et al [26] proposed dynamic switching-frequency scheduling scheme using various scheduling time slice to address the performance degradation for the over-commit domain. Merkel et al [27] introduced the pattern recognition method to detection the resource competition and tried to reduce the energy consumption through contention avoidance. Kazempour et al [28] proposed an asymmetry-aware scheduler for VE on Asymmetric Multi-Processors (AMP) hardware platform, which gives guest OS in VE an effective sight of AMP platform.

VIII. Conclusion and Future Works

To alleviate exclusiveness and contention among multiple concurrent domains in current versions of hybrid co-scheduling, we present two optimizations named PCS and BCS using finer space granularity. PCS raises the CPU scheduling signals only for the necessary CPUs in co-scheduling operation instead of for all online CPUs; while BCS uses voluntary approach that boosts the priorities for co-scheduled VCPUs to induce the scheduler to pick the appropriate VCPUs. According to the analyses and experimental results, PCS and BCS maintain the advantages of hybrid co-scheduling, and benefit virtual machines performance and fairness further, especially when there are multiple concurrent domains in a system. However, PCS exactly follows the hybrid co-scheduling criteria, while it introduces more complexity; BCS simplifies the implementation, while it looses the co-scheduling rules, that does not perform well in some situation such as cross-domain concurrency. The experimental results show that, when two domains run concurrent CPU-intensive applications simultaneously in a virtualized system, PCS and BCS improve the performance by approximately 25% compared to non-co-scheduling underlying scheduler, and double the performance compared to existing hybrid co-scheduling and co-descheduling. The execution time difference between the two concurrent domains is less than 3% on average.

Some related topics are worth our further investigation. First, the co-scheduling scheme limits the number of VCPUs in one domain. Therefore, how to overcome the over-commit restriction, especially with multiple concurrent domains, is one of our future effort. Second, AMP system gets more and more attention due to many-core processors. We plan to investigate the problems of co-scheduling and parallel computing in AMP virtualized environments in future.

References

[1] The Message Passing Interface (MPI) standard. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[2] PVM: Parallel Virtual Machine. [Online]. Available: http://www.csm.ornl.gov/pvm/

[3] OpenMP.org. [Online]. Available: http://openmp.org/wp/

[4] V. Uhlig, J. LeVasseur, E. Skoglund, et al, “Towards scalable multiprocessor virtual machines,” in *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004, pp. 1–14.

[5] T. Friebel and S. Biemueller. (2008) How to deal with lock holder preemption. [Online]. Available: http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf

[6] W. Jiang, Y. Zhou, Y. Cui, et al, “CFS optimizations to KVM threads on multi-core environment,” in *International Conference on Parallel and Distributed Systems*, 2009, pp. 348–354.

[7] J.K. Ousterhout, “Scheduling techniques for concurrent systems,” in *Proc. of the 3rd International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[8] C. Weng, Z. Wang, M. Li, et al, “The hybrid scheduling framework for virtual machine system,” in *Virtual Execution Environments*, 2009, pp. 111–120.

[9] D.G. Feitelson and L. Rudolph, “Gang scheduling performance benefits for fine-grain synchronization,” *Journal of Parallel and Distributed Computing*, vol. 16, no. 1, pp. 306–318, 1992.

[10] D.G. Feitelson, “Packing schemes for gang scheduling,” in *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1996, pp. 89–110.

[11] VMWare Communities. (2008) Co-scheduling smp vms in vmware esx server. [Online]. Available: http://communities.vmware.com/docs/DOC-4960

[12] C. Xu, Y. Bai and C. Luo, “Performance evaluation of parallel programming in virtual machine environment,” in *International Conference on Network and Parallel Computing*, 2009, pp. 140–147.

[13] Y. Bai, C. Xu and Z. Li, “Task-aware based co-scheduling for virtual machine system,” in *Symposium On Applied Computing*, 2010, pp. 181–188.

[14] Yaron. (2007) Credit Scheduler. [Online]. Available: http://wiki.xensource.com/xenwiki/CreditScheduler

[15] Xen hypervisor source. [Online]. Available: http://xen.org/download/index_4.0.1.html

[16] P. Barham, B. Dragovic, K. Fraser, et al, “Xen and the art of virtualization,” in *ACM Symposium on Operating Systems Principles*, 2003, pp. 164–177.

[17] Stanford Parallel Applications for Shared Memory (SPLASH). [Online]. Available: http://www-flash.stanford.edu/splash/

[18] S. Woo, M. Ohara, E. Torrie, et al, “The SPLASH-2 programs: characterization and methodological consideration,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995, pp. 24–36.

[19] H. Jin, M. Frumkin and J. Yan, “The OpenMP implementation of NAS Parallel Benchmarks and its performance,” NASA Ames Research Center, Tech. Rep. NAS-99-011, October 2003.

[20] F. Wang, H. Franke, M. Papaefthymiou, et al, “A gang scheduling design for multiprogrammed parallel computing environments,” in *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1996, pp. 111–125.

[21] M.A. Jette, “Performance characteristics of gang scheduling in multiprogrammed environments,” in *Supercomputing 97*, 1997, pp. 1–12.

[22] G. Somani and S. Chaudhary, “Application performance isolation in virtualization,” in *IEEE International Conference on Cloud Computing*, 2009, pp. 41–48.

[23] S. Govindan, A.R. Nath, A. Das, et al, “Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms,” in *Virtual Execution Environments*, 2007, pp. 126–136.

[24] S. Govindan, J. Choi, A.R. Nath, et al, “Xen and co.: communication-aware CPU management in consolidated Xen-based hosting platforms,” *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 1111–1125, 2009.

[25] J. Liu and B. Abali, “Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization,” in *International Conference on Supercomputing*, 2009, pp. 225–234.

[26] H. Chen, H. Jin, K. Hu, et al, “Dynamic switching-frequency scaling: scheduling overcommitted domains in xen vmm,” in *International Conference on Parallel Processing*, 2010, pp. 287–296.

[27] A. Merkel, J. Stoess and F. Bellosa, “Resource-conscious scheduling for energy efficiency on multicore processors,” in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 153–166.

[28] V. Kazempour, A. Kamali and A. Fedorova, “AASH: an asymmetry-aware scheduler for hypervisor,” in *Virtual Execution Environments*, 2010, pp. 85–96.