The Hybrid Scheduling Framework for Virtual Machine Systems

Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu

Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
{clweng, felixwang, mlli, xdlu}@sjtu.edu.cn

Abstract

The virtualization technology makes it feasible that multiple guest operating systems run on a single physical machine. It is the virtual machine monitor that dynamically maps the virtual CPU of virtual machines to physical CPUs according to the scheduling strategy. The scheduling strategy in Xen schedules virtual CPUs of a virtual machines asynchronously while guarantees the proportion of the CPU time corresponding to its weight, maximizing the throughput of the system. However, this scheduling strategy may deteriorate the performance when the virtual machine is used to execute the concurrent applications such as parallel programs or multithreaded programs. In this paper, we analyze the CPU scheduling problem in the virtual machine monitor theoretically, and the result is that the asynchronous CPU scheduling strategy will waste considerable physical CPU time when the system workload is the concurrent application. Then, we present a hybrid scheduling framework for the CPU scheduling in the virtual machine monitor. There are two types of virtual machines in the system: the high-throughput type and the concurrent type. The virtual machine can be set as the concurrent type when the majority of its workload is concurrent applications in order to reduce the cost of synchronization. Otherwise, it is set as the high-throughput type as the default. Moreover, we implement the hybrid scheduling framework based on Xen, and we will give a description of our implementation in details. At last, we test the performance of the presented scheduling framework and strategy based on the multi-core platform, and the experiment result indicates that the scheduling framework and strategy is feasible to improve the performance of the virtual machine system.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management

General Terms Algorithms, Experimentation, Performance

Keywords Virtualization, Scheduling Strategy, Hybrid Scheduling

1. Introduction

With the development of the computer technology, the processing power of computer system is increasing quickly, and the multi-core (many-core) CPU gradually becomes popular in the computer system. As a result, it is feasible to aggregate the functionality of multiple standalone computer systems into a single hardware computer, in order to promote the usage of the hardware while decrease the cost of the power. Virtualization technology [1][2] is a good way to achieve these benefits. Differing from the traditional system software stack, a virtual machine monitor (VMM) is inserted between the operating system level and the hardware level in the virtualized system. In the virtual machine system, multiple virtual machines (VM) with a specified individual instance of the operating system are running simultaneously on the top of the VMM. Currently, examples of system virtulization include VMWare [3], Xen [4], Virtual server [5], Denali [6][7], User Mode Linux [8], etc.

However, the complexity of the virtualization technology introduces additional management challenges. In this paper, we focus our attention on CPU scheduling in the virtual machine system. It is the VMM that virtualizes the physical CPU to the virtual CPU, on which the guest operating system is running. Usually the total number of virtual CPUs in the virtualized system is larger than the number of physical CPUs, and the schedule module in the VMM maps virtual CPUs of virtual machines into physical CPUs in a time-share manner. The fair share of CPU should be guaranteed by the VMM, That is to say, each VM should get the number of CPU time in a proportion according to the strategy. For example, a VM with a weight of 10 will get twice as much CPU as a VM with a weight of 5 on a contended host.

In the virtual machine system with the multi-core CPU, usually treated as a virtual symmetric multiprocessing (SMP) system, a specifical characteristic is that all virtual CPUs in a SMP virtual machine are usually not online all the time, and possibly not online at the same time, which is different from the non-virtualization scenario. This is because the number of virtual CPUs of all virtual machines is usually larger than the number of the physical CPUs, and these virtual CPUs have to share the limited number of the physical CPUs in turn. When the workload in a VM is the non-concurrent application, the VMM schedules virtual CPUs of the VM asynchronously while just has to guarantee a certain proportion of the CPU time. This method is beneficial to simplify the implementation of the CPU scheduling in the VMM, and will not distinctly reduce the performance of the system, and it is widely adopted in the implementation of the VMM. However, when the workload in the VM is the concurrent application such as multithreaded programs or parallel programs with the synchronization operation, these existing CPU scheduling methods may deteriorate the performance.

The motivation of this paper is to improve the performance of the CPU scheduling when the workload of the VM is the concurrent application. The main contribution of this paper includes as follows. With theoretical analysis, the conclusion is that the asynchronous virtual CPU scheduling strategy will waste considerable physical CPU time when the system workload is the concurrent application. A hybrid scheduling framework is presented to efficiently deal with the variety of applications in the VM, where VMs are di-

VEE'09, March 11–13, 2009, Washington, DC, USA.

vided into the high-throughput VM and the concurrent VM. And the scheduling strategy is presented for mapping virtual CPUs in the VMM, which adopts different algorithms correspondingly for the two different types of VMs, while guarantees the fairness of CPU sharing among VMs.

The rest of this paper is organized as follows. The next section presents the CPU scheduling problem and its analysis in the VMM. Section 3 presents a hybrid scheduling framework and algorithm for the VMM. Section 4 discusses the implementation of the scheduling framework and algorithm. Section 5 discusses the experimental results. Section 6 provides a brief overview to the related works, and Section 7 concludes the paper.

2. CPU Scheduling Problem and Analysis in the VMM

In this section, we firstly describe the general virtual machine architecture, and then give a typical CPU scheduling scenario to describe the CPU scheduling problem in the VMM. Finally, we theoretically analyze the CPU scheduling problem in the VMM.

2.1 System virtualization

As depicted in Figure 1, virtualization provides an additional layer (VMM) between the running operating system and the underlying hardware. The VMM manages the hardware resources and exports them to the operating systems running on them. As a result, the VMM is in full control of allocating the physical CPUs to the guest operating system.

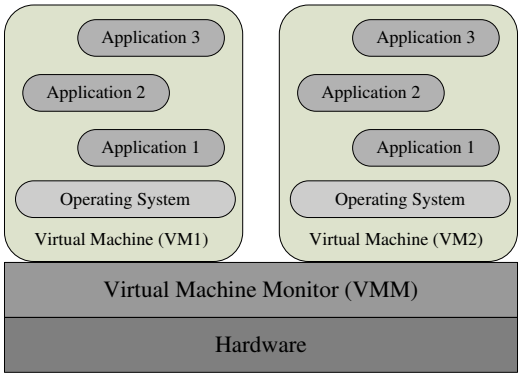

Figure 1. General virtual machine architecture

Currently, the VM with multiple processors is treated as a virtual SMP system, where all processors behave identically and any process can execute on any processor. There are two problems with CPU scheduling in the VMM, on which multiple virtual SMP systems run. One problem is how to guarantee the CPU fairness among different VMs, that is, the CPU time obtained by a VM should be in a certain proportion. And the other is how to deal with the synchronization problem such as the lock-holder preemption [9] for the virtual SMP system. For example, the VMM can preempt a virtual CPU with a thread holding a lock, which will result in an extension of the lock holding time. Before modeling and analysis, we firstly give a scenario to illustrate these two problems.

2.2 Scenario

There are four processors in a SMP virtual machine, on which a thread of a multithreaded program is running respectively, and the unit time of CPU scheduling is a slot. The weight of the VM is $3/10$, and there is a synchronization operation between threads at the end of each step, and the length of the step is equal to the length

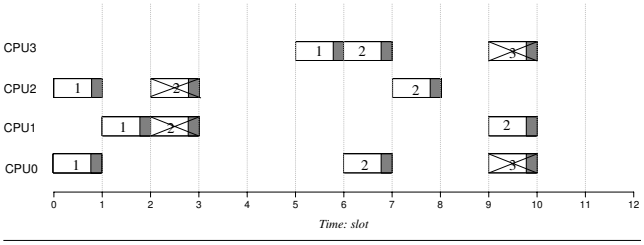

Figure 2. The scenario of Noncoscheduling

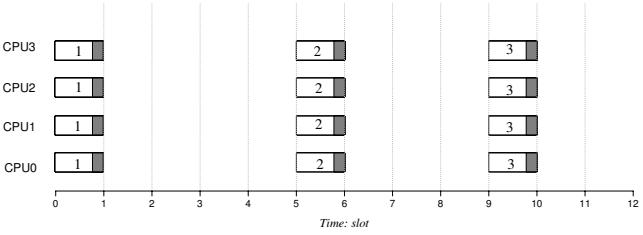

Figure 3. The scenario of coscheduling

of the slot. This scenario can be abstracted from the multithreaded program or the parallel program.

There exist two kinds of scheduling strategies. One is that each virtual CPU is asynchronously assigned to the physical CPU in order to maximize the throughput, while guaranteeing the CPU fairness according to the weight. However, this strategy will deteriorate the performance when the workload is a concurrent application. A possible scheduling sequence of the multithreaded program by this scheduling strategy is shown as Figure 2, the multithreaded application only completes the 2 steps in the length of the 10 slots, while there are 4 slots of CPU time to be wasted for the synchronization.

The other scheduling strategy is that each virtual CPU is synchronously assigned to the physical CPU in order to avoid the additional cost of the synchronization, while guaranteeing the CPU fairness according to the weight. One possible scheduling sequence of the multithreaded program by the scheduling strategy is shown as Figure 3, and it completes the 3 steps in the length of the 10 slots.

This scenario with the two strategies gives a straightforward comparison, and the coscheduling strategy outperforms the noncoscheduling strategy for the concurrent workload, under the condition that the CPU fairness is guaranteed, and each virtual CPU runs 3 slots in the length of 10 slots. The limitation of the coscheduling strategy is that the number of the virtual CPU should be no more than the number of the physical CPU, and the scheduling of one application may be delayed because lack of enough idle physical CPUs. In the follows, we will theoretically analyze the assignment of the concurrent application on the SMP virtual machine.

2.3 Modeling

This section formalizes the job model, defines the scheduling model, and presents the objective function of the schedule.

Job model. $J = \{J_1, J_2, ..., J_{|J|}\}$ denotes a concurrent job, which comprises $|J|$ tasks. Each task of the job is running in sequence, and has to synchronize with other tasks in a fixed interval. Task J_i can be decomposed by the synchronization operation into a sequence of $|J_i|$ phases, that is, $\{J_i^1, J_i^2, ..., J_i^{|J_i|}\}$. Each phase of a task comprises one part of computation and one subsequent part of synchronization. For the kth phase of task J_i, $J_i^k.e$ denotes the part of the computation, and $J_i^k.s$ denotes the part of the synchronization. Once one phase of a task begins to run on a physical

CPU, we assume that it will keep running until the completion of the phase.

Scheduling model. In the virtual machine system, the assignment of a concurrent job in the SMP virtual machine includes: the assignment of the job on virtual CPUs by the guest operating system, and the assignment of the virtual CPUs on the physical CPUs by the VMM.

The physical computer includes a group of homogenous CPUs, denoted by $P = \{P_1, P_2, ..., P_{|P|}\}$, and the number of the physical CPUs is $|P|$. The VMs running on the physical computer is denoted by $V = \{V_1, V_2, ..., V_{|V|}\}$, and $|V|$ denotes the number of VMs in the system. The weight proportion of VM V_i is denoted by $\omega(V_i)$, which represents the proportion of physical CPU time consumed by the VM, and then $\sum_i \omega(V_i) = 1$. The set of virtual CPUs in the ith VM is denoted by $C(V_i) = \{v_{i1}, v_{i2}, ..., v_{i|C(V_i)|}\}$, and $|C(V_i)|$ denotes the number of virtual CPUs in VM V_i. For avoiding overmuch switching cost of virtual CPUs mapping on physical CPUs, the following relation exists, that is, $\forall i, |C(V_i)| \leq |P|$. For the same reason, we assume that $|J| \leq |C(V_i)|$.

The scheduling problem of a concurrent job J in VM V_i is formalized by $\chi = (\tau, \pi)$, where $\tau : J \rightarrow \{0, 1, 2, ..., \infty\}$, and $\pi : J \rightarrow P$. τ maps phases of tasks to the set of time slots, and π maps phases of tasks to the set of physical CPUs in the system.

As each task of the concurrent job is running in sequence and synchronizes with each other at the end of phases in interval, we have $\tau(J_k^m) + J_k^m.e + J_k^m.s < \tau(J_l^n)$ if $m < n$. $\pi(J_k^m)$ denotes a physical CPU, on which task J_k runs during the course of its phase J_k^m, and it is determined by the mapping realtions: $J_k^m \rightarrow C(V_i)$ and $C(V_i) \rightarrow P$. The makespan of job J executed on VM V_i by the schedule χ is $T_\chi(J) = \max_k\{\tau(J_k^{|J_k|}) + J_k^{|J_k|}.e + J_k^{|J_k|}.s\}$.

Objective function. We now formulate the scheduling issue of the concurrent job in the virtual machine system as an optimization problem as follows.

Let χ be a schedule of a concurrent job J of VM V_i on the physical machine with P. The objective function is $\min T_\chi(J)$, while subject to:

The proportion of the CPU time obtained by VM V_i is $\omega(V_i)$;
$\tau(J_k^m) + J_k^m.e + J_k^m.s < \tau(J_l^n)$ if $m < n$;
$|J| \leq |C(V_i)| \leq |P|$.

2.4 Strategy analysis

The first-come-first-service (FCFS) strategy is a simple and efficient method to schedule the computation-intensive concurrent application on the SMP system. So we assume that a SMP virtual machine is dedicated to execute tasks of a concurrent job in a certain period. In this paper, we focus on the CPU scheduling in the VMM, and assume that each task of a concurrent job will be assigned to a fixed virtual CPU by the guest operating system for reducing the cost of the context switch. Correspondingly, the job scheduling problem reduces to the scheduling of the virtual CPU to the physical CPU in the system. Then in the schedule χ, τ and π also represent the mapping relation of the corresponding virtual CPU to the set of time slots and the set of physical CPUs.

Formally, the time is subdivided into a sequence of fixed-length *slots*, which is the basic time unit, and slot i corresponds to the time interval $[i, i + 1)$. Within each slot, each physical CPU is allocated to a corresponding virtual CPU. To simplify the presentation, we assume that each phase of a task is equal to a slot, then $J_i^k.e + J_i^k.s = 1$.

$Received(v_{ij}, t_1, t_2)$ denotes the CPU time obtained by the virtual CPU v_{ij} in the interval $[t_1, t_2)$, and the CPU time obtained by the VM is equally distributed among its virtual CPUs. Then the deviation is defined as follows:

$$Lag(t, v_{ij}) = t \times |P| \times \omega(V_i)/|V_i| - Received(v_{ij}, 0, t) \quad (1)$$

The $|Lag(t, v_{ij})|$ is used to evaluate the fairness of the scheduling strategy. For an ideal fair scheduling strategy, which can guarantee that the CPU time consumed by a VM is strictly in proportion to its weight, we have $|Lag(t, v_{ij})| \leq 1$, where the time unit is a slot.

Proportional share (PS) scheduling strategy. This strategy allocates CPU in proportion to the number of shares (weights) that VMs have been assigned, which displays as Figure 2. This strategy is widely adopted by VMMs such as Xen. And the correlation of the two virtual CPUs is not considered during the assignment of the physical CPU to the virtual CPU. The key part of this strategy is the concept of weight, while the CPU time obtained by a VM is equally distributed among its virtual CPUs.

Lemma 1. For the concurrent job J in VM V_i, the supremum of the completion time T_m of phase m by the ideal PS scheduling strategy is:

$$\sup T_m = \max_k \tau(J_k^{m-1}) + \left\lfloor \frac{2 \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor + 1$$

Proof. At the time $t_0 = \max_k \tau(J_k^{m-1})$, the virtual CPU v_{ij^*} with the maximal next scheduling interval has the following property. It is scheduled nearly overfull, and $Lag(t_0, v_{ij^*}) = -1$, while it will be scheduled at the time t_1 when $Lag(t_0, v_{ij^*}) = 1$. Because the virtual CPU v_{ij^*} is not actually scheduled, we have $Received(v_{ij^*}, 0, t_0) = Received(v_{ij^*}, 0, t_1)$. According to Equation (1), $t_1 \times |P| \times \omega(V_i)/|V_i| - t_0 \times |P| \times \omega(V_i)/|V_i| = 2$ holds. Then, $t_1 - t_0 = \left\lfloor \frac{2 \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor$. As the virtual CPU v_{ij^*} has the maximal next scheduling interval, we have $T_m = \max_k \tau(J_k^{m-1}) + (t_1 - t_0) + 1$, that is, $\sup T_m = \max_k \tau(J_k^{m-1}) + \left\lfloor \frac{2 \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor + 1$. ∎

Theorem 1. For the concurrent job J in VM V_i, the supremum of the makespan of J by the PS scheduling strategy with the maximal deviation $|Lag|$ is:

$$(\max_k |J_k|) \times \left\lfloor \frac{2 \times |Lag| \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor.$$

Proof. In Lemma 1, after replacing the deviation value of 1 with the value of the PS scheduling strategy's maximal deviation $|Lag|$, the maximal scheduling interval is: $t_1 - t_0 = \left\lfloor \frac{2 \times |Lag| \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor$. And task J_i has $|J_i|$ phases, and the maximum of phases of Job J is $\max_k |J_k|$, then the supremum of the makespan of J is $(\max_k |J_k|) \times \left\lfloor \frac{2 \times |Lag| \times |V_i|}{|P| \times \omega(V_i)} \right\rfloor$. ∎

Co-proportional share (CPS) scheduling strategy. Besides guaranteeing that the CPU time is allocated to VMs in proportion to the number of their weights, this strategy co-assigns the set of virtual CPUs in a VM to the physical CPUs, that is, virtual VPUs are coscheduled to the physical CPUs in the system.

Theorem 2. For the concurrent job J in VM V_i, the makespan of job J by the CPS scheduling strategy is:

$$\left\lceil \frac{(\max_k |J_k|) \times |V_i|}{|P| \times \omega(V_i)} \right\rceil.$$

Proof. As all virtual CPUs are coscheduled to the physical CPUs, the makespan is determined by the time length of the task with the maximal number of phases. The weight proportion of a virtual CPU in VM V_i is $\frac{\omega(V_i)}{|V_i|}$, and the number of physical CPUs is $|P|$, and the execution time of each phase is 1, then the makespan of $(\max_k |J_k|)$ phases is $\left\lceil \frac{(\max_k |J_k|) \times |V_i|}{|P| \times \omega(V_i)} \right\rceil$. ∎

Corollary 1. For a concurrent job J in VM V_i, the makespan of job J by the PS strategy with maximal deviation $|Lag|$ is denoted

by $makespan_{ps}$, and the makespan of job J by the CPS strategy is denoted by $makespan_{cps}$, then, $\frac{makespan_{ps}}{makespan_{cps}} = 2 \times |Lag|$.

Corollary 1 is easily deduced by the above two theorems. For a practical PS scheduling strategy, the value of $|Lag|$ is usually larger than 1. Therefore, the makespan of the concurrent job J by the PS strategy will be not less than the twice of the makespan by the CPS strategy at worst. As a result, the scheduling issue in the VMM should be studied further to reduce this kind of the performance loss.

In this section, we theoretically analyze the two kinds of scheduling strategies for the CPU scheduling in the VMM. Theoretical result indicates that the coscheduling strategy is a potential efficient method for allocating physical CPUs to virtual CPUs when the workload in the VM is the concurrent application.

3. Scheduling Framework and Algorithm

This section presents a hybrid scheduling framework, specifics regarding its implementation are detailed in the following section. We begin with the high-level design issues that have driven our work, followed by the scheduling framework in the VMM. The final part of this section gives a description of the CPU scheduling algorithm in the VMM.

3.1 Design issues

To maximize the performance in the virtual machine system, the following design issues should be considered for the scheduler in the VMM.

- **Fairness**. When multiple VMs share a single physical system, it is not a good appearance that a VM with the heavy workload can impropriate the CPU resource of other VMs without any limitation. The scheduler in the VMM should keep fairness in resource sharing among VMs. That is, the amount of the CPU time obtained by a VM is controlled. An effective method is proportional share fairness, by which the CPU usage of a VM is in proportion to the number of weights that the VM has been assigned.
- **Workload balancing**. For multiple virtual CPUs in a VM, their assigned physical CPU time should be nearly equal. So the scheduler should balance the CPU time among virtual CPUs in a SMP VM, and make virtual CPUs perform similarly as the physical CPUs in the real SMP machine, and avoid the long delay of a virtual CPU compared to the other virtual CPUs in the VM.
- **Wasted CPU time**. The additional cost is unavoidably introduced when the system is virtualized. Currently, the additional cost of virtualization is relatively small. However, the inappropriate semantic rules may bring an unneglectable cost of the wasted CPU. For example, a considerable CPU time may be wasted during the period of synchronization between virtual CPUs in a VM.
- **Adaptiveness**. There are a variety of workloads in VMs, and VMs are scheduled by the VMM to the physical CPUs in the system. So it is a large challenge for the scheduler that can achieve the goal of good fairness, efficient workload balancing, and minimal wasted CPU time, when allocating the physical CPU time to VMs. The scheduler with a good adaptiveness can make a better trade-off among these factors, and can change its strategy for VMs with the different workload properties.

3.2 Scheduling framework

Applications running on the VM can be classified as the high-throughput kind of applications and the concurrent kind of applications. For a high-throughput application, the scheduling goal is to maximize the throughput. Taking a web server application as an example, the performance goal is that the server can process access requests as many as possible, and the processing threads of requests are independent from each other. Obviously, the PS scheduling strategy is suitable for scheduling physical CPU time over virtual CPUs when the workload of the VM is the high-throughput application, as adopted in Xen.

For the concurrent application, the CPS scheduling strategy is a potential efficient method as analyzed above (see Section 2.4). On the other hand, The constraint is that the number of the virtual CPU should be no more than the number of the physical CPU, and the scheduling of an application may be delayed because lack of enough idle physical CPUs. Moreover, the implementation of coscheduling also introduces an additional cost. It is a trade-off issue between the increased performance and additional cost. However, the performance profit is larger than the performance loss when a SMP VM is dedicated to execute the concurrent application.

Therefore, we present a hybrid scheduling framework for the VMM. In the scheduling framework shown as Figure 4, virtual machines on the VMM can be divided into two kinds, one kind is the high-throughput VM, and the other is the concurrent VM. The two kinds of VMs will be created to run the corresponding kinds of applications. Moreover, a VM can be changed from the high-throughput type to the concurrent type, or from the concurrent type to the high-throughput type, in response to the change of its workload's characteristic. When the majority of the workload in a VM is the high concurrent application, the FCFS job assignment is adopted.

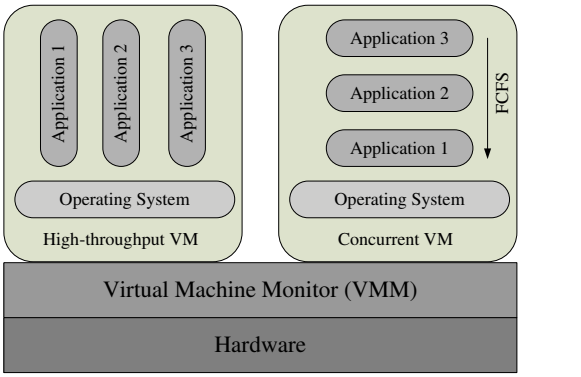

Figure 4. The hybrid structure of virtualization

Besides the number of the virtual CPU, the capacity of the memory, and the location of the file system, etc., the two properties should be set when a VM is created, that is, the weight of the VM and the type of the VM. The configure file of a VM includes all above information. The setup work flow of the CPU scheduling for a VM is shown as Figure 5. The initial scheduling parameters are set in the configure file of a VM, which is created by the system administrator. According to the configure file, the VM is created with a certain weight. After the VM is created, the VMM scheduling module has to recalculate the allocation proportion of physical CPU time among VMs, and the new weight proportion of CPU time of a VM is the value of its weight divided by the new total weight. Then the scheduling module assigns the physical CPU time to VMs according to the new proportion, while the scheduling fashion of virtual CPUs in a VM is determined by the VM's type. When it is a concurrent VM, its virtual CPUs are mapped into physical CPUs synchronously, otherwise its virtual CPUs are mapped asynchronously. During the lifetime of a VM,

its type can be modified by the system administrator, according to the characteristic of its workload.

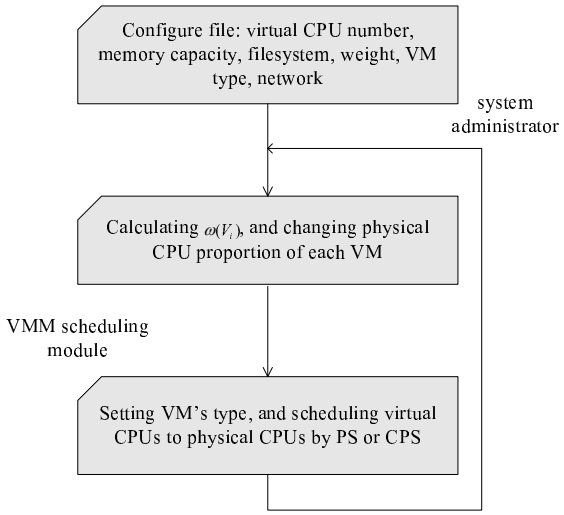

Figure 5. The setup work flow of scheduling

3.3 Scheduling algorithm

After the weight of VMs and their types are determined, then the physical CPUs are allocated to the virtual CPUs by the VMM scheduling algorithm.

For representing the type of VMs, we define an enumerate structure as follows.

```
typedef enum {
  HIT, /*the high-throughput type*/
  CON, /*the concurrent type*/
}VMType;
```

For VM V_i, $VT(V_i)$ denotes the type of the VM, and there are two types of VMs, $VT(V_i) = HIT$ if V_i is a high-throughput VM, and $VT(V_i) = CON$ if V_i is a concurrent VM. We also define: $VT(v_{ij}) = HIT$ if V_i is a high-throughput VM, and $VT(v_{ij}) = CON$ if V_i is a concurrent VM.

The scheduling algorithm for the VMM supporting the SMP VM includes the following four parts.

Virtual CPU initial mapping. When a VM is created, each virtual CPU of the VM will be inserted to the run queue of a physical CPU, respectively. All available physical CPUs for the virtual CPU v_{ij} makeups a set of physical CPUs, which is denoted by $AP(v_{ij})$, and we have $AP(v_{ij}) \subseteq P$. For the two types of VMs, we have:

- if $VT(V_i) = CON$, for virtual CPU v_{ij}, there doesn't exist any virtual CPU v_{ik} ($k = 1, ..., |C(V_i)|, and\ k \neq j$) in the run queue of any physical CPU $P_l \in AP(v_{ij})$.
- if $VT(V_i) = HIT$, for virtual CPU v_{ij}, $AP(v_{ij}) = P$.

Then, a physical CPU P_{l^0} with the minimal workload in $AP(v_{ij})$ is chosen, and the virtual CPU v_{ij} is inserted into the run queue of the physical CPU P_{l^0}.

Physical CPU time allocation. The physical CPU time of the system is allocated to VMs in proportion to the number of the weights that VMs have been assigned. The concept of potential energy (PE) is borrowed from the mechanics. In a certain interval, the PE value of each virtual CPU is reset according to the weight.

The PE value of virtual CPU v_{ij} is $PE(v_{ij})$. The dissipation of PE per slot is denoted as PE_{unit}, the time length of the allocation interval is denoted as L slots, and the time of allocating physical CPU time is denoted as an allocation event. Then in the interval with L slots, the total PE of the system is $PE_{total} = |P| \times PE_{unit} \times L$.

The procedure of PE allocation is as the following pseudocode.

```
For each VM V_i:
  The PE increment PE_inc = PE_total × ω(V_i);
  For each virtual CPU v_ij in VM V_i
    PE(v_ij) = PE(v_ij) + PE_inc ÷ |C(V_i)|;
```

After updating the PE value of virtual CPUs, virtual CPUs in the run queue of a physical CPU will be sorted in the decreasing order of their PE values.

PE dissipation. We denote the virtual CPU currently running on the physical CPU P_i as $PV(P_i)$, and it is the head element in the run queue of the physical CPU, and the run queue of P_i is denoted by $runq(P_i)$. After the time of a slot, the PE value of the virtual CPU running on a corresponding physical CPU is decreased by PE_{unit}, and the processing procedure is as the following pseudocode, where the boot strap processor is abbreviated as BSP.

```
For each physical CPU P_k:
  PE(PV(P_k)) = PE(PV(P_k)) − PE_unit;
  If P_k is BSP, and an allocation event time achieves
    Performing Physical CPU time allocation;
  Else
    Sorting virtual CPUs in the decreasing order of PE;
```

Scheduling and balancing. At the beginning of each slot, a virtual CPU should be selected for each physical CPU in its run queue. The scheduling and balancing algorithm is described as the following pseudocode.

```
For each physical CPU P_k:
  1) If PE(PV(P_k)) < 0, then:
    Lookuping v_{i*j*} in the head of the run queue of
    P_{k'}(k' ≠ k), with PE(v_{i*j*}) = max_{k'} PE(PV(P_{k'})),
    and runq(P_k) ∩ C(V_{i*}) = φ if VT(V_{i*}) = CON, and
    then migrating v_{i*j*} to P_k and executing it.
  2) If PE(PV(P_k)) > 0 and VT(PV(P_k)) = CON,
  then:
    Lookuping all other virtual CPUs that belong to the
    same VM as PV(P_k), and their physical CPUs. Then P_k
    sending Inter-Percossor Interrupt (IPI) to these physical
    CPUs, and all virtual CPUs in this VM being scheduled.
  3) If PE(PV(P_k)) > 0 and VT(PV(P_k)) = HIT,
  then:
    The virtual CPU PV(P_k) being scheduled to the
    physical CPU P_k.
```

Then the work flow of the VMM scheduling algorithm with the above four parts is as Figure 6. It is the part of ***Physical CPU time allocation*** that guarantees the CPU fairness according to the weight that VMs are assigned, and the coscheduling operation for virtual CPUs in a concurrent VM is implemented in the part of ***Scheduling and balancing***.

4. Detailed Design and Implementation

We have implemented a working prototype of the hybrid scheduling framework in the virtual machine monitor. It is based on open-source Xen-3.0.4, and the guest operating system is Linux. We

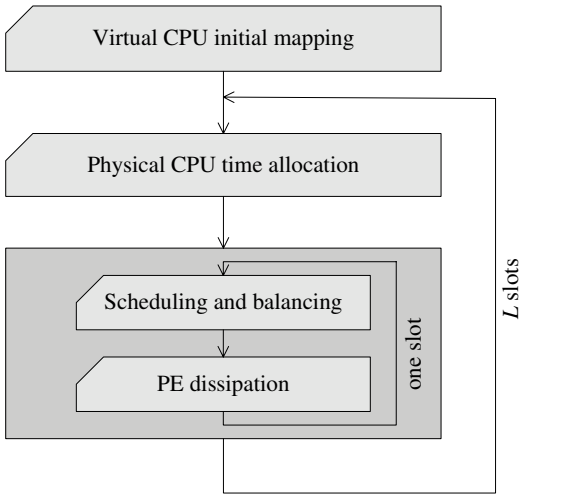

Figure 6. The work flow of CPU scheduling

choose Linux and Xen because of their broad acceptance and the availability of their open-source codes.

In the following subsection, the implementation of the hybrid scheduling framework and the algorithm is described. At the first, we will give a overview and discussion of the CPU scheduling strategies in Xen.

4.1 CPU scheduling algorithms in Xen

Xen allows users to choose the CPU scheduling algorithm among different algorithms. Three different CPU scheduling algorithms are introduced until now, all allowing users specify CPU allocation by the CPU weight. As the implemented prototype in this paper is based on Xen, we briefly discuss the main features of the three scheduling algorithms in Xen.

Borrowed Virtual Time (BVT). It is a fair-share scheduling algorithm based on the concept of virtual time, dispathing the runnable VM with the smallest virtual time firtstly [10]. The low-latency support is provided in BVT for realtime and interactive applications by allowing latency-sensitive client to warp back in virtual time to gain scheduling priority. And the client can effectively borrow virtual time from its future CPU allocation.

The scheduling algorithm is configured with a context switch allowance, and it is the real time by which the current VM is allowed to advance beyond another runnable VM with equal claim on the CPU. Each runnable VM receives a share of CPU in proportion to its weight. To achieve this goal, the virtual time of the currently running VM is incremented by its running time divided by the weight.

The lack of the non work-conserving (NWC) mode in BVT severely limited its usage, and led to the introduction of the other scheduling algorithm.

Simple Earliest Deadline First (SEDF). In this algorithm, the real time property is used to ensure time guarantees [11]. Each VM specifies the slice and period together to represent the CPU share requested by this VM, that is, a VM will receive at least the slice specified by the VM in each period of the length specified by the VM.

One boolean flag indicates whether the VM is eligible to receive extra CPU time (work-conserving mode). This slack time is distributed in a fair manner after all the runnable VMs received their CPU share. A VM obtains 30% when the slice is equal to 3 while the period is equal to 10, or when the slice is equal to 30 while the period is equal to 100. The time granularity in the definition of the period impacts the scheduler fairness.

One main shortage is the lack of global workload balancing on multiprocessors, and the CPU fairness depends on the value of the period.

Credit Scheduling. This algorithm is a kind of proportional share (PS) strategy, featuring automatic workload balancing of virtual CPUs across physical CPUs on a SMP host [12]. Before a CPU goes idle, it will find any runnable virtual CPU in the run queue of the other physical CPUs. This approach guarantees that no physical CPU idles when there exists a runnable virtual CPU in the system.

Each VM is associated with a *weight* and a *cap*. When the cap is 0, VM can receive extra physical CPU (WC-mode), while a non-zero cap (expressed as a percentage) limits the amount of physical CPU time obtained by a VM (NWC-mode). The algorithm uses 30 ms time slices for the physical CPU allocation. The priorities (credits) of all runnable VMs are recalculated in the interval of 30ms, which is mainly in proportion to *weight* that VMs are assigned by the user. The basic unit time of scheduling is 10 ms, and the credit of the running virtual CPU is decreased every 10 ms.

This algorithm can efficiently achieve a global workload balancing on a SMP system when the majority of the workload is not the high concurrent application. It is the latest scheduling algorithm and the default scheduling algorithm in Xen.

4.2 Scheduling module

In this subsection, we will describe the scheduling module in the VMM based on Xen.

Similarly as the scheduling module in Linux and Xen, this module is invoked every clock tick. At each clock tick, the mission of the scheduling module running on each physical CPU is to determine the next virtual CPU, which will execute on the physical CPU. The key codes of the scheduling module are as follows.

```
static void schedule(void) {
  ......
  next_slice = ops.do_scheduler(now);
  ......
}
```

In this paper, a new scheduler $sched\_hybrid\_def$ is implemented based on Xen as follows.

```
static struct scheduler *schedulers[] = {
    &sched_sedf_def,
    &sched_credit_def,
    &sched_hybrid_def,
    NULL
};
```

Specificially, the $sched\_hybrid\_def$ scheduler is defined as follows, which is similar as the structure of the credit scheduler in Xen.

```
struct scheduler sched_hybrid_def = {
    .name           = "Multi-core PE Scheduler",
    .opt_name       = "pe",
    .sched_id       = XEN_SCHEDULER_PE,
    .init_domain    = hsched_dom_init,
    .destroy_domain = hsched_dom_destroy,
    .init_vcpu      = hsched_vcpu_init,
    .destroy_vcpu   = hsched_vcpu_destroy,
    .sleep          = hsched_vcpu_sleep,
    .wake           = hsched_vcpu_wake,
    .adjust         = hsched_dom_cntl,
    .pick_cpu       = hsched_cpu_pick,
```

```
    .tick           = hsched_tick,
    .do_schedule    = hsched_schedule,
    .dump_cpu_state = hsched_dump_pcpu,
    .dump_settings  = hsched_dump,
    .init           = hsched_init,
};
```

4.3 Coscheduling implementation

For avoiding that the two groups of the virtual CPUs from the two individual concurrent VMs perform coscheduling at the same time, so we define a global variable syn_domain_id, which should be modified in the mutually exclusive means. And a global variable syn_map is defined to record the ID of physical CPUs needed to be coscheduled.

The skeleton of the coscheduling implementation is as follows.

```
/* The coscheduling launched by others*/
/* participating in the coscheduling */
if((syn_domain_id > 0)&&(cpu_isset(cpu,syn_map))){
 find the virtual CPU in the runq to be
 coscheduled, and return it;
}
/* Not participating in the coscheduling */
else if
((syn_domain_id > 0)&&(!cpu_isset(cpu,syn_map))){
 take the head virtual CPU in the runq,
 and return it;
}
/* there is no coscheduling currently */
else{
 take the head virtual CPU in the runq,
  if (its PE > 0 ){
   if (its VM is CON, and spin_trylock(lock)){
    Add the ID of physical CPUs into the
    workers, which is associated with virtual
    CPUs in the VM;
    cpumask_raise_softirq \\
    (workers, SCHEDULE_SOFTIRQ);
    spin_unlock(lock);
   }
   else{
    return it;
   }
  }
  else{
 /* get the virtual CPU with positive
    PE from others */
   return hsched_workload_balance();
  }
}
```

The virtual CPUs of a VM distributed on a group of physical CPUs are coscheduling through the `SCHEDULE_SOFTIRQ` raised by a physical CPU, which firstly schedules one of the virtual CPUs. And the `hsched_workload_balance` selects the runnable virtual CPU from other physical CPUs, while avoiding that the two virtual CPUs in a concurrent VM are in the run queue of the same physical CPU.

4.4 User interface

When a VM is created, it is a high-throughput VM by default, and the user interface should be provided to the system administrator, so that the type of a VM can be changed by the system administrator according to the characteristic of the workload in the VM.

Therefore, we added new option "`xm sched-pe`" into the Xen management tool, that is `xm`. The user interface of the hybrid scheduler is implemented by C and Phyton languages. Through this user interface, a VM's type and its weight can be set directly by the system administrator.

When a VM changing from the high-throughput type to the concurrent type by the system administrator, additional checking and adjustment should be considered in the scheduling module in the VMM. That is because any two virtual CPUs of a VM should be not in the run queue of the same physical CPU when the VM is a concurrent type.

5. Performance Evaluation

In this section,we perform some experiments on the virtual machine system with benchmarks.

5.1 Experimental methodology

Workload. To evaluate the system performance, we used two benchmarks suites from Stanford Parallel Applications for Shared Memory (SPLASH-2) [13][14]. Additionally, a throughput benchmark of web server is adopted. Specifically, the three benchmarks are as follows.

- **LU**. The LU kernel in SPLASH-2 factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = N \times B$) to exploit temporal locality on submatrix elements.
- **Barnes**. The Barnes application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method.
- **Web server**. We measure the throughput of the web server, which is implemented by httfperf [15]. Httperf is a tool for measuring the web server performance. It provides a flexible facility for generating various HTTP workloads and for measuring the server performance.

Experimental system. All experiments were executed on a Dell PowerEdge 1900 server with dual quad-core Xeon X5310 CPUs, and 2GB of RAM. The virtualized system ran Xen 3.0.4, and all VMs ran the Fedora Core 6 Linux distribution with the Linux 2.6.16 kernel.

Scheduler. In the experiment, we will compare the credit scheduler in Xen with the hybrid scheduling scheduler. The performance of three types of VMs is tested, that is, the VM in the default Xen, and the concurrent VM and the high-throughput VM.

5.2 Experimental result

For testing the performance of the CPU scheduler in the VMM, a VM as *Dom1* is configured with 4 virtual CPUs and 512MB memory. The VM *Dom0* is configured with 8 virtual CPUs and 1024MB memory, and its weight is 256. We test the performance with the above three workloads, that is, LU, Barnes, and web server with httperf.

For each situation, the performance of *Dom1* in the default Xen is denoted by `Credit`. In the prototype of the hybrid scheduling framework, the performance of *Dom1* is denoted by `CON` when it is set as a concurrent type, while the performance of *Dom1* is denoted by `HIT` when it is set as a high-throughput type.

When the workload in *Dom1* is LU, $n = 4096$, and $B = 16$, and the number of processors is 4, and the result is shown as Figure 7. When the workload in *Dom1* is Barnes, the number of bodies is 262144, and the number of processors is 4, and the other parameters are the default values, and the result is shown as Figure 8. When the workload is the web server, the web server is

built by Tomcat, and is measured by httperf with the default value, and the result is shown as Figure 9.

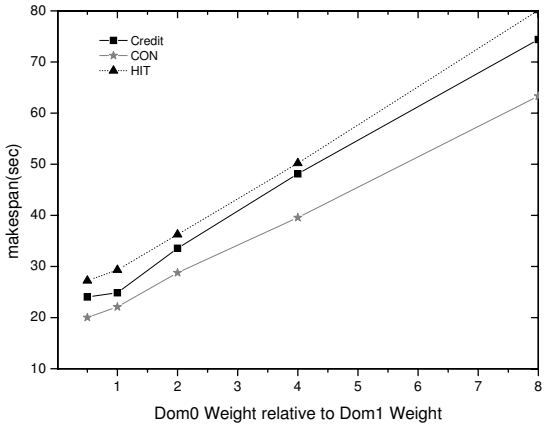

Figure 7. LU workload

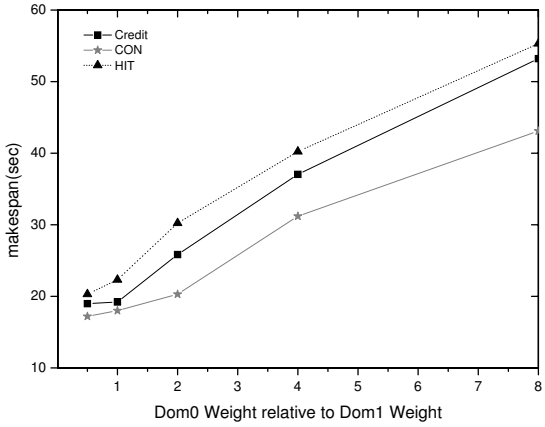

Figure 8. Barnes workload

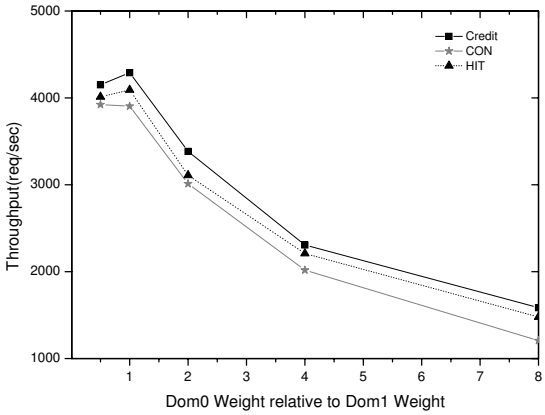

Figure 9. Web server workload

Experimental results indicate that the coscheduling strategy performs well when the majority of workload is the concurrent application, otherwise may deteriorate the performance. And the credit scheduler in Xen can achieve a good performance when there are high-throughput workloads in the VM. However, the performance of the credit scheduler is lower than the performance of the coscheduling strategy, when the workload in VM is the high concurrent application.

Another result can be concluded from the group of experiments. That is, the working prototype of the hybrid scheduling framework will achieve a better performance when the appropriate strategy is selected for the VM.

6. Related Works

The scheduling issue in the VMM had close relationship with the scheduling of operating system, as the VMM could be considered as a specific kind of operating system.

A simple notion of priority for process scheduling is usually used in conventional operating systems. A task with a higher priority is scheduled prior to a task with a lower priority, and priorities may be static or be dynamically recalculated. While there are many sophisticated priority schemas such as decay usage scheduling [16]. It is a priority and usage based mechanism for CPU scheduling employed by BSD [17] Unix. Decay usage scheduling is motivated by two concerns: fairness and performance. For achieving fairness, CPU slice is allocated to processes that have received fewer slices in the recent past. The CPU time will be allocated to I/O-intensive processes before the computation-intensive processes, in order to improve throughput by overlapping CPU and I/O activities. For ensuring that a particular application receives a certain percent of the CPU usage, the fair-share schedulers is introduced [18][19], and it can provide proportional sharing among processes and users in a way compatible with a UNIX-style time sharing framework. However, experiments indicates that the fair-share scheduler provides reasonable proportional fairness over relatively large time intervals [20]. Lottery scheduling provides a more disciplined proportional sharing approach than fair-share schedulers [21]. Firstly each process receives a number of tickets proportional to its share. Then the scheduler randomly picks a ticket and schedules the process that owns this ticket to receive a CPU slice.

There are much research for effectively scheduling parallel application on general multi-processors or distributed system. Backfilling and gang scheduling [22] are two major strategies for scheduling parallel jobs. Backfilling scheduling such as [23] [24] tries to balance the goals of utilization, which attempts to schedule jobs that are behind in the priority queue of waiting jobs to unutilized nodes, rather than keep them idle. To prevent starvation of larger jobs, it requires that the execution of a job selected out of order will not delay the start of jobs that are ahead of it in the priority queue. This method is based on the estimation of job execution time. And coscheduling (gang-scheduling) [25][26][27], on the other hand, tries to schedule related threads or processes to run simultaneously on different processors. When scheduling any of the processes in the related group, all of them are scheduled for execution so that they can communicate efficiently. Otherwise, one could wait to send or receive a message to another while it is sleeping, and vice-versa. Some works [28][29] propose combining these two strategies together for better performance.

Applications can be typically divided into two types: I/O-intensive applications and computation-intensive applications. Correspondingly, current research efforts are focused on the scheduling issue in the VMM with the computation-intensive workloads and I/O-intensive workloads, respectively.

For improving the performance of computation-intensive workload in the virtual machine system, especially with the SMP VM, VMWare and Xen provide their scheduler in the VMM. VMware ESX Server is a popular commercial virtualization system for the x86 architecture that runs on the physical hardware with no lower level OS, and ESX server uses coscheduling for virtual CPUs in order to maintain correctness and high performance [30][31], which is an add-on software module. However, the number of vir-

tual CPUs is too strict, and the coscheduling strategy is fixed in the module. As an open-source product, the current default scheduler in Xen [4] is the credit scheduler [12], and this scheduler does not attempt to coschedule virtual CPUs. It is a proportional share scheduling strategy, and tries to maximize the throughput of the system while guaranteeing the fairness. Moreover, the three scheduling algorithms in Xen are compared in [32].

Differing from schedulers in VMWare and Xen, the hybrid scheduling framework in this paper can schedule virtual CPUs in a dynamic means according to the type of the VM, has a good adaptiveness aiming at the variety of the workload in the VM, while guaranteeing the fairness among VMs.

In the aspect of I/O-intensive workload, in order to improve the performance of I/O-intensive applications on the virtual machine system, a communication-aware CPU scheduler for Xen hypervisor is proposed [33]. The SEDF scheduler is modified so that it counts the numbers of received or sent packets by each domain (VM) and preferentially schedules I/O-intensive domains. SEDF and Credit schedulers are evaluated with different configurations [34]. Furthermore, they make some extensions such as fixing event channel notification and ordering the run queue within the CPU scheduler of the Xen hypervisor to improve I/O performance. VMM-bypass I/O [35] is also a method to improve I/O performance, which allows guest domains to carry out I/O operations without the involvement of the hypervisor and the privileged domain. Self-virtualized devices[36][37][38] also share the same idea. Another relevant work is concerned with monitoring the performance [39][40][41][42] (especially, the overhead for I/O processing) and the scheduler is configured with the gained insight from those information to achieve a high performance.

7. Conclusion

The CPU scheduling strategy and algorithm is crucial to promote the performance of the virtual machine system. Many aspects need be considered in the design of a scheduling strategy for the VMM, such as fairness, workload balance, adaptiveness. It is one of most challengeable problems in the virtualization technology.

In this paper, we analyze the CPU scheduling problem in the virtual machine monitor theoretically, and the result is that the asynchronous CPU scheduling strategy will waste considerable physical CPU time when the majority of the workload is the concurrent application. Therefore, we present a hybrid scheduling framework for the CPU scheduling in the VMM, which supports two types of virtual machines in the system: the high-throughput type and the concurrent type. The VM can be set as the concurrent type when the majority of its workload is the concurrent application, in order to reduce the cost of synchronization. Otherwise, it is set as the high-throughput type as the default. Moreover, we implement a prototype of the hybrid scheduling framework based on Xen. At last, we test the performance of the presented scheduling strategy based on the multi-core platform, and the experimental result indicates that the hybrid scheduling framework is feasible to improve the performance of the virtual machine system.

Although the proposed extensions to the VMM scheduling discussed in this paper may improve CPU performance to some degree, there is still much room for the further improvement. Specifically, the tradeoff of CPU scheduling between the concurrent type and the high-throughput type of VMs need to be further considered.

Acknowledgments

This work was supported partly by National Key Basic Research and Development Plan (973 Plan) of China (No. 2007CB310900), and Huawei Science and Technology Foundation, and National Natural Science Foundation of China (No. 90612018, 90715030 and 60503043). We would like to thank the anonymous reviewers for their thoughtful comments and suggestions.

References

[1] P. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[2] J. Smith and R. Nair. *Virtual Machines: Versatile platforms for systems and processes*. Elsevier, USA, 2005.

[3] C. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, December 2002.

[4] 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. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[5] Microsoft Corporation. Microsoft virtual server 2005. http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx.

[6] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, pages 195–209, October 2002.

[7] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, October 2002.

[8] J. Dike. User-mode linux. In *Proceedings of the 5th annual Linux Showcase & Conference*, 2001.

[9] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research & Technology Symposium (VM'04)*, San Jose, CA, may 2004.

[10] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a generalpurpose scheduler. In *Proceedings of the 17th ACM SOSP*, 1999.

[11] I. Leslie, D. Mcauley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[12] Credit Scheduler. http://wiki.xensource.com/xenwiki/credit scheduler.

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

[14] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.

[15] httperf. http://www.hpl.hp.com/research/linux/httperf/.

[16] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, 1993.

[17] S. Leffler, M. McKusick, and M. Karels. *The Design and Implementation of the 4.3* BSD *Unix Operating System.* Addison-Wesley, 1988.

[18] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[19] G. Henry. The fair share scheduler. AT&T *Bell Labs Technical Journal*, 63(8):1945–1957, 1984.

[20] R. Essick. An event based fair share scheduler. In *Proceedings of the Winter USENIX Conference*, pages 147–161, 1990.

[21] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, 1994.

[22] D. Feitelson, L. Rudolph and U. Schwiegelshohn. Parallel job scheduling - a status report. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16, 2004.

[23] B. Lawson, E. Smirni, and D. Puiu. Self-adapting backfilling scheduling for parallel systems. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP)*, pages 583–592, 2002.

[24] E. Shmueli and D. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 228–251, 2003.

[25] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Third International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[26] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[27] A. Batat and D. Feitelson. Gang scheduling with memory considerations. In *Proceedings in 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 109–114, 2000.

[28] Y. Wiseman and D. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, 2003.

[29] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 133–142, 2000.

[30] VMWARE. Performance tuning best practices for ESX server 3, 2007. http://www.vmware.com/pdf/vi_performance_tuning.pdf.

[31] VMWARE. Best practices using vmware virtual SMP, 2005. http://www.vmware.com/pdf/vsmp_best_practices.pdf.

[32] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[33] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and Co.: Communication-aware cpu scheduling for consolidated Xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE)*, pages 126–136, 2007.

[34] D. Ongaro, A. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the 4th international conference on Virtual execution environments (VEE)*, pages 1–10, 2008.

[35] J. Liu, W. Huang, B. Abali, and D. Panda. High performance vmm-bypass I/O in virtual machines. In *Proceedings of USENIX '06 Annual Technical Conference*, 2006.

[36] H. Raj and K. Schwan. Implementing a scalable self-virtualizing network interface on an embedded multicore platform. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2005.

[37] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 306–317, 2007.

[38] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 179–188, 2007.

[39] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, pages 387–390, 2005.

[40] A. Menon, J. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance: Overheads in the Xen virtual machine environment. In *Proceedings of the 1st international conference on Virtual execution environments (VEE)*, pages 13–23, 2005.

[41] D. Gupta, R. Gardner, and L. Cherkasovah. XenMon: QoSmonitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, 2005.

[42] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2006.