A Simulation Framework to Evaluate Virtual CPU Scheduling Algorithms

Cuong Pham, Qingkun Li, Zachary Estrada, Zbigniew Kalbarczyk, Ravishankar K. Iyer
University of Illinois at Urbana-Champaign
{pham9, qli19, zestrad2, kalbarcz, rkiyer}@illinois.edu

***Abstract* — Virtual CPU (VCPU) scheduling algorithms that efficiently manage processing-resource at the machine virtualization layer are key to facilitate resource sharing and workload consolidation in Clouds. Such algorithms are mostly inherited from pre-virtualization designs, thus need to be revamped and re-evaluated. This paper presents a simulation framework based on SAN model to rapidly evaluate Virtual CPU scheduling algorithms. The paper also demonstrates an evaluation of three VCPU scheduling algorithms using this framework.**

Keywords: Cloud Computing, VCPU, Scheduling, Simulation

I. Introduction

Virtualization is one of the enabling technologies for Cloud Computing. It facilitates resource sharing and workload consolidation, allowing for better resource utilization, as well as energy and cost savings. Achieving this goal requires implementing highly efficient resource scheduling algorithms. This paper focuses on VCPU scheduling algorithms. These algorithms deal with the problem of assigning physical CPUs (PCPUs) to Virtual Machines (VMs) in a balanced way.

Empirical evaluation of VCPU scheduling algorithms is difficult and requires significant effort. As Virtualization technology is becoming more mature, the code base of the hypervisors is growing (e.g. the 3.3 version of the XEN hypervisor has 300K line of code). In addition, scheduling is often implemented at the system level, which requires deep understanding of the hardware architecture, as well as proficiency in low-level programming and debugging. Furthermore, the proposed algorithms are sometimes affected by the existing hypervisor architecture. For example, when evaluating the balance scheduling algorithm for KVM, Sukwong el at. [1] had to tweak the algorithm to make it work above the Linux kernel's process scheduling.

In order to address this problem, this paper presents a model of the virtualization environment, with an open interface for users to define their VCPU scheduling algorithms. The model then can be simulated to evaluate the performance of the defined algorithm. The model is built on top of Stochastic Activity Network (SAN) model [5], using Mobius simulation tool [9]. It allows:

- assembling a complete virtualization system with flexible configurations, e.g., an arbitrary number of VMs with an arbitrary number of VCPUs;
- plugging in any VCPU scheduling algorithm in the form of C functions;
- automating and accelerating simulation experiments to evaluate the plugged in algorithms.

Although the framework's components are constructed using the SAN model, the users of the framework are not required to know SAN or its underlining concepts. All they need is to use Mobius's GUI interface to drag and drop components, draw connections (e.g. from a VCPU component to a VM component), type in parameters (e.g. workload distribution), and write a C function to express the scheduling function. We believe that the ease of use and fast evaluation process of this approach significantly boosts the preliminary evaluations of VCPU scheduling algorithms.

II. Background and Related Work

A. SAN Model and Mobius Overview

SAN model [5] is a higher abstraction of Petri Nets. SAN has been demonstrated to be a convenient and effective model to evaluate system's characteristics related to performance and dependability [6].

Informally, the basic constructs of SANs are the following:

Place ●: a place contains a natural number of tokens and can represent a possible state of the modeled system.

Activity ⊢: an activity indicates transitions between places. It expresses how long a transition takes to complete, and it can be described as a random variable. This construct can have a set of cases, which are used to model the possible outcomes of a transition.

Input gate ◀: input gates enforce a condition for an activity to be enabled.

Output gate ▶: output gates allow the execution of a function after the completion of an activity. Output gates can be used to update the state of the model.

Composed model: Multiple SAN models can be combined into a composed model using Replicate and Join operations. Composed models are often used to simplify the model and mitigate the state explosion problem.

Once constructed, a model can be solved either analytically/numerically or by simulation, as provided by the Mobius tool. Our framework utilizes the simulation infrastructure of Mobius.

B. VCPU Scheduling Algorithms

VCPU scheduling remains a challenge for Virtualization technologies, especially with hypervisors starting to host symmetric multiprocessing (SMP) VMs.

A naïve, yet popular, implementation is to use a simple Round-Robin algorithm when assigning processor resources to each VCPU. This option is available in most hypervisors. Sometimes it is the only option, e.g. in KVM or Virtual Box hypervisors. This approach can cause additional synchronization latency for guest VMs due to VCPU preemption. For example, most critical sections in an OS kernel are non-preemptible as they are designed to finish quickly and reduce the wait time of other threads. However, VCPU scheduling is usually unaware of guest preemptions, due to a problem called semantic gap, it may preempt a VCPU, which is in a middle of executing a critical section. This causes other threads, which is waiting on the same lock in other VCPUs, wait additional time.

In order to eliminate this synchronization latency, VMWare applies a co-scheduling algorithm [3], which uses a concept similar to gang scheduling [4]. The idea of co-scheduling is as follows: the scheduler forces all the VCPUs of a VM to start (co-start) and stop (co-stop) at the same time. Such an algorithm helps to avoid the synchronization latency, as both the waiting VCPUs and the lock-holding VCPU are preempted and resumed at the same time. This "strict" co-scheduling approach, however, introduces a fragmentation problem. A VCPU can only be scheduled after the hypervisor gathers enough resources to execute all other VCPUs in the same VM.

VMWare later implemented a 'relaxed' co-scheduling algorithm [2] in their ESX 3 and 4 versions. This algorithm makes its best effort to perform co-starts and co-stops when resources are available. In case there are not enough resources to perform a co-start, it allows a single VCPU to be scheduled. The scheduler maintains a cumulative skew for each VCPU, compared to the rest of VCPUs in the same VM. When the skew of a VCPU grows above a certain threshold, it is forced to schedule in the "co-start" manner only (until the skew drops below a pre-defined threshold).

This relaxed co-scheduling mitigates the CPU fragmentation problem, but it introduces synchronization latency as a trade-off. Sukwong el at. argued in [1] that synchronization latency significantly increases when the sibling VCPUs (the VCPUs in the same VM) are scheduled in the run-queue of the same physical CPU. They called this scenario the VCPU-stacking problem. Following that observation, they introduced the balance-scheduling algorithm, which attempts to avoid the VCPU-stacking problem.

C. *Related Work*

Different kinds of VCPU scheduling algorithms [2, 3, 6] have been discussed in Section I. For the scheduler modeling, [7] applies a mathematical model to analyze and compare proportional share, co-proportional share scheduling strategies, and then they proposes a scheduling framework implemented on Xen. Study [7] empirically compares XEN's three built-in VCPU scheduling algorithms. Our simulation framework can be used to compare the algorithms. More importantly, it provides a convenient infrastructure to quickly examine new (e.g., idea-based) algorithms.

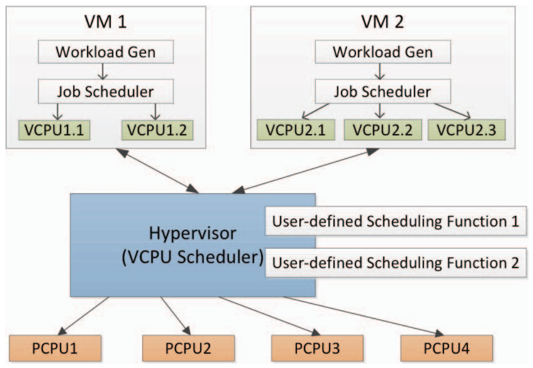

Figure 1: Structural model of a virtualization system

III. VM SIMULATION FRAMEWORK

This section describes the design and implementation of a complete virtualization model, which allows users to conveniently construct virtualization systems and exercise any VCPU scheduling algorithm.

A. *Framework Overview*

A virtualization system consists of hardware resources (e.g. CPU cores, memories, hard-disks), a hypervisor, and one or more VMs. Each VM has at least one VCPU, and at most the same number of VCPUs as the number of physical cores. Each VM is responsible for executing one workload, which is distributed evenly on its VCPUs. The hypervisor is responsible for assigning physical CPU resource to VCPUs.

The proposed framework mimics this architecture by allowing users to construct VM models (e.g. defining number of VCPUs and workload characterizations), define VCPU scheduling functions (in form of a C function), and configure the number of physical CPUs, and put them all together to run simulations. Figure 1 illustrates this structure.

A VM model is composed of a workload generator sub-model, several VCPU sub-models, and a job scheduler sub-model to evenly distribute the workload to the VCPUs. The number of VCPUs in each VM is configurable: users can plug in as many VCPU sub-models to the composed VM model as they need to. The workload sub-model randomly generates workloads (represented by load duration – the amount of time it requires a VCPU to process) and synchronization points (the barrier that stops the workload generation until all the preceding jobs are completed). A VCPU sub-model has an interface to connect to the job scheduler (inside the VM) and another interface to connect to the VCPU scheduler (inside the hypervisor).

In order to support user-defined scheduling functions, the defined VCPU scheduler model exports a C function call interface, which passes the states of the VCPUs and PCPUs, to an outside library. Users implement their C/C++ VCPU scheduling function in this interface.

For the purpose of evaluating VCPU scheduling algorithms, our framework considers physical CPU cores as the only hardware resource.

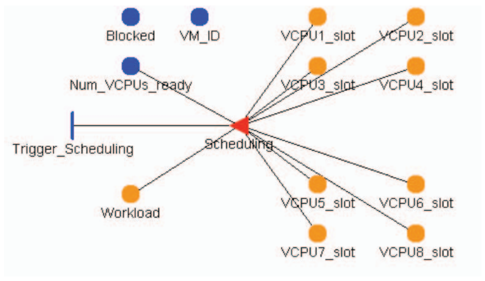

Figure 3: SAN model of Job Scheduler

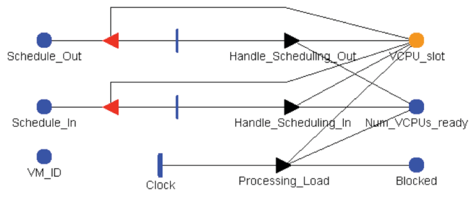

Figure 4: SAN model of VCPU

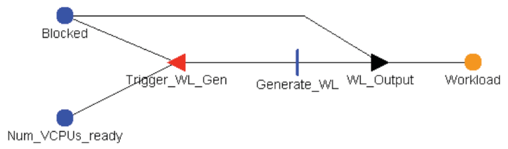

Figure 5: SAN model of Workload Generator

B. Framework Components

This section describes the sub-models of this framework. These are pluggable components with pre-defined joining places between models to guide the construction of virtualization systems.

1) Job Scheduler of a VM

This model is the hub of each VM. It takes inputs from a workload generator model via the `Workload` place. Based on the state of all the VCPUs, represented by `VCPUx_slot` places (`x=1..8`), the function of input gate `Scheduling` decides which VCPUs to pass the workloads to. Each `VCPUx_slot` is later joined with one VCPU model. The `Scheduling` event is fired when (i) there is a pending workload and (ii) there is at least one `READY` VCPU. The `Blocked` place is enabled when a synchronization point is blocking the VM from processing requests. This place is shared across all sub-models in a VM.

Figure 3 shows the SAN model of the Job Scheduler. In this figure, eight VCPU slots are statically defined to allow at most eight VCPU models to be plugged. In order to support bigger VMs, more VCPU slots can easily be added.

2) VCPU

Figure 4 shows the SAN model of this component. The VCPU model connects with the Job Scheduler model via the `VCPU_slot` place, which consists of the following fields:

- `remaining_load`: the remaining time to complete the current load.

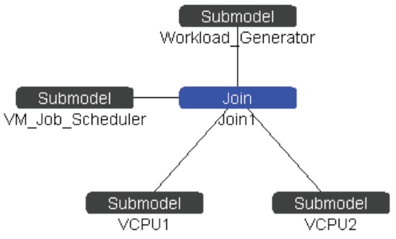

Figure 2: Composed model of a two VCPUs VM

TABLE 1: JOIN PLACES* IN VIRTUAL MACHINE MODEL

State Name	Sub-model Variables
`Blocked`	`Workload_Generator->Blocked`
	`VM_Job_Scheduler->Blocked`
	`VCPU1->Blocked`
	`VCPU2->Blocked`
`Num_VCPUs_ready`	`Workload_Generator->Num_VCPUs_ready`
	`VM_Job_Scheduler->Num_VCPUs_ready`
	`VCPU1->Num_VCPUs_ready`
	`VCPU2->Num_VCPUs_ready`
`VCPU1_slot`	`VM_Job_Scheduler->VCPU1_slot`
	`VCPU1->VCPU_slot`
`VCPU2_slot`	`VM_Job_Scheduler->VCPU2_slot`
	`VCPU2->VCPU_slot`
`Workload`	`Workload_Generator->Workload`
	`VM_Job_Scheduler->Workload`

*Note: Common names are join places

- `sync_point`: if the value is 1, it represents a synchronization point. Otherwise (if the value is 0), this workload does not require synchronization.
- `status`: the status of the VCPU. A VCPU can be:
  - `READY`: assigned a PCPU, but no workload assigned.
  - `BUSY`: assigned a PCPU and processing a workload.
  - `INACTIVE`: not assigned to any PCPU. Note that this VCPU can be in the middle of processing a workload (reflected by the `remaining_load` field), or even holding a lock (reflected by the `sync_point` field).

When a VCPU is in the `BUSY` state, at each time unit (triggered by the `Clock` activity described later) the `Processing_load` output gate reduces the `remaining_load` by 1. When the `remaining_load` reaches 0, the status of the VCPU is changed to `READY`, and `Num_VCPUs_ready` gets increased by 1. Both the `BUSY` and `READY` states can be implicitly considered as `ACTIVE` states.

The VCPU model also connects with the Virtual CPU scheduler via the `Schedule_Out` and `Schedule_In` places. The `Schedule_In` place notifies the VCPU that it has been assigned a PCPU. Meanwhile the `Schedule_Out` place notifies the VCPU that it has to relinquish the assigned PCPU, thus transit to `INACTIVE` state.

3) Workload Generator

Figure 5 shows the SAN model of this component. This sub-model generates a workload when two conditions are met: (i) there is at least one `READY` VCPU, and (ii) the VM is not blocked (due to synchronization points). Each generated workload consists of two fields:

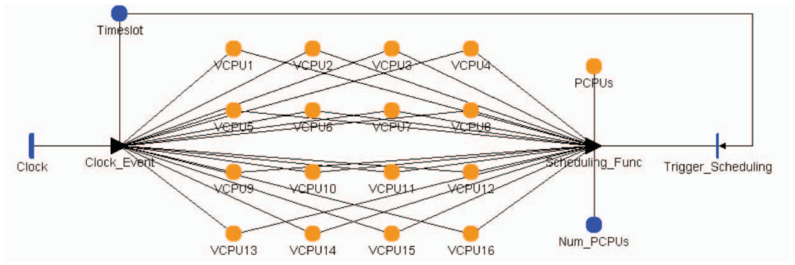

Figure 6: SAN model of VCPU Scheduler

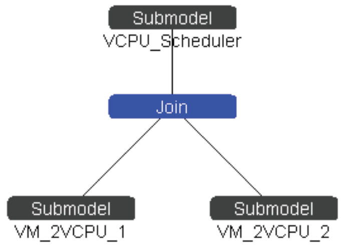

Figure 7: Composed model of a virtualization system with two VMs

TABLE 2: JOIN PLACES IN VIRTUAL SYSTEM MODEL

State Variable Name	Sub-model Variables
`Schedule_In1_1`	`VM_2VCPU_1->Schedule_In1`
	`VCPU_Scheduler->VCPU1->Schedule_In`
`Schedule_In1_2`	`VM_2VCPU_1->Schedule_In2`
	`VCPU_Scheduler->VCPU2->Schedule_In`
`Schedule_Out1_1`	`VM_2VCPU_1->Schedule_Out1`
	`VCPU_Scheduler->VCPU1->Schedule_Out`
`Schedule_Out1_2`	`VM_2VCPU_1->Schedule_Out2`
	`VCPU_Scheduler->VCPU2->Schedule_Out`

* Shown join places are between the first VM_2VCPU_1 model and VCPU_Scheduler model. Join places of the second VM_2VCPU_2 are omitted due to space limit.

- `load`: the time it takes for a VCPU (with an assigned PCPU) to process the workload
- `sync_point`: represents synchronization primitives. For this project, we only consider barrier synchronization. This means that synchronization points require all the preceding jobs to be completed before the next job can be assigned. The content of this field is passed to the `sync_point` field in `VCPU_slot` of the VCPU that this workload is assigned.

The generation of `load` and `sync_point` is configurable to any distribution and rate. The generations happen in output gate `WL_Output`. One important parameter is the ratio of number of synchronization points to the number of workloads. For example, the 1:5 ratio means that for five workloads there is one synchronization point. This ratio affects the efficiency of synchronization latency solutions, such as strict co-scheduling and relaxed co-scheduling.

4) Virtual Machine

The Virtual Machine model is a composed model that consists of a Workload Generator, a Job Scheduler, and several VCPU sub-models. Users can adjust the number of VCPU sub-models. Figure 2 illustrates a Virtual Machine model with two VCPUs. The join places of this model are presented in **TABLE 1**.

5) Virtual CPU Scheduler

The VCPU Scheduler model consists of the following components:

- `Clock`: fires at every time unit to regulate the operation of the scheduling function (inside `Scheduling_Func` output gate) and computes the remaining timeslice of each `ACTIVE` VCPU.
- `VCPU` places: each `VCPU` place represents one possible VCPU in the system. A place is enabled only when it is connected with a VCPU model via the `Schedule_In` and `Schedule_Out` fields. An place consists of the following fields:
    - `Schedule_In` and `Schedule_Out`: used to connect with a VCPU model. As explained in Section III.B.2), the scheduling function uses `Schedule_In` and `Schedule_Out` to notify the VCPU when a PCPU is assigned and unassigned, respectively.
    - `Last_Scheduled_In`: this field stores the time stamp when the VCPU was last assigned a PCPU. This information is needed by scheduling algorithms to determine the next VCPUs that get PCPUs.
    - `Timeslice`: when a PCPU is assigned to a VCPU, a timeslice is also assigned to the VCPU to specify how long the VCPU can keep the PCPU. The timeslice decreases as `Clock` fires until it reaches 0 and the VCPU must relinquish the PCPU. A `Schedule_Out` event will be sent to the VCPU model.
- `Num_PCPUs`: users use this place to configure the number of PCPUs in the system.
- `PCPUs` array: each element of this array contains the state of a PCPU (`IDLE` or `ASSIGNED`).
- Scheduling function (defined as the function of `Scheduling_Func` output gate): this function is essentially a connector to user-defined scheduling functions. We defined a standard function call interface that can be used to call virtually any scheduling function:

```
bool  schedule(VCPU_host_external*  vcpus,  int
num_vcpu, PCPU_external* pcpus, int num_pcpu,
long timestamp)
```

Where:

- `VCPU_host_external`: a data structure that has the same layout as the `VCPU` place;
- The pointer `*vcpus`: points to an array of `VCPU_host_external` elements. This array is used as both input and output of the function;
- `num_vcpu`: the number of VCPUs in the system;
- `PCPU_external`: a data structure that contains the state of a PCPU in the system;
- The pointer `*pcpus`: points to an array of `PCPU_external` elements. Similar to the `vcpus` array, this array is used as both input and output to reflect the state of PCPUs before and after the execution of the scheduling function;

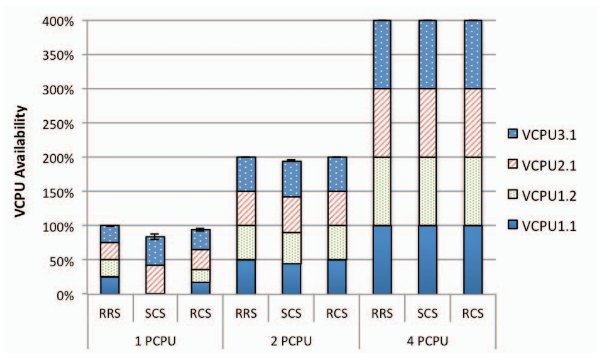

Figure 8: The availability of four VCPUs in three VMs (2VCPUs + 1 VCPU + 1 VCPU) (95% confidence level)

- o `num_pcpu`: the number of PCPUs in the system;
- o `timestamp` indicates the system time.

Figure 6 shows the SAN model of VCPU scheduler. Note that the model statically defines 16 VCPU slots, which allows for 16 VCPU sub-models. The slots that do not have plugged in VCPU sub-models are not enabled. In order to support bigger virtualization systems, more VCPU slots can be easily added to the model.

6) *Virtual System*

Virtual System model is a composed model consisting of a VCPU scheduler and several pluggable Virtual Machine sub-models. The number and settings of Virtual Machine sub-models are user-configurable. Figure 7 shows a Virtual System with two VMs, each VM having two VCPUs. The join places of this model are presented in **TABLE 2**.

IV. EVALUATION

We use the simulation framework to compare three VCPU scheduling algorithms: Round-Robin Scheduling (RRS), Strict Co-Scheduling (SCS) [3], and Relaxed Co-Scheduling (RCS) [2].

A. *Verifying the Fairness of Scheduling Algorithms*

In the first set of simulations, we compared the scheduling fairness of the three algorithms. Fairness reflects the ability of scheduling algorithms to guarantee that all VCPUs receive resources in a balanced way. In order to quantify fairness, we define a metric called *VCPU Availability*. This metric reflects the average portion of time that a VCPU is in the ACTIVE state during a simulation. A *100% VCPU Availability* means that the VCPU is always in ACTIVE state (e.g., there are more PCPUs than VCPUs). This metric is obtained by using a reward variable (in the SAN model) that monitors the state transition of each VCPU. The setup of the experiment is as follows:

- Three VMs: one 2-VCPU VM (VCPU1.1 and VCPU1.2) and two 1-VCPU VM (VCPU 2.1 and VCPU3.1);
- The synchronization rate is 1:5;
- The number of PCPUs is varied from 1 to 4.

Figure 8 presents the result of the simulation (with 95% confidence level and <0.1 confidence interval). The results

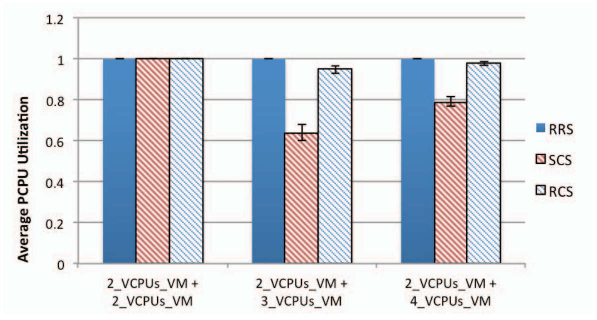

Figure 9: The averaged PCPU Utilization (of four PCPUs) in different VM setups (95% confidence level)

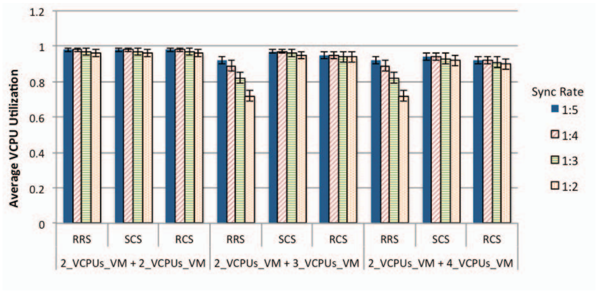

Figure 10: The averaged VCPU Utilization with four PCPUs in different VM setups (95% confidence level)

show that RRS always achieves scheduling fairness regardless of the resource. In our setup, the two co-scheduling algorithms achieve poorer fairness than RRS. For example, in the one PCPU setup, SCS cannot schedule the 2-VCPUs VM due to the strict requirement of VCPU co-start – the number of PCPUs is always smaller than the number of required VCPUs. In the same setup, RCS is able to schedule the 2-VCPU VM, thanks to the relaxed requirement of co-start. However, due to the skew-threshold constraint, the VCPUs of these 2-VCPU VM receive less PCPU resources than the VCPUs of 1-VCPU VMs (which are not constrained by the skew-threshold). The fairness of the two co-scheduling algorithms improves as the number of PCPUs increases. RCS generally achieves better fairness than SCS. They achieve balanced scheduling (i) for the VM that have the same configuration and (ii) when the number of VCPUs is not greater than the number of PCPUs.

B. *Physical CPU Utilization*

In this set of simulations, we evaluated the *PCPU utilization* of the three scheduling algorithms with different VM configurations. *PCPU utilization* measures the portion of time that a PCPU is assigned to VCPUs during a simulation. This metric also reveals the CPU fragmentation problem of the co-scheduling algorithms. In order to obtain the averaged utilization of all the PCPUs, we defined a reward variable (in the SAN model) that monitors the state transition of all the PCPUs. The setup of the experiment is as follows:

- Three sets of VMs: (set 1) each VM had two VCPUs; (set 2) the first VM has two VCPUs and the second VM had three VCPUs; (set 3) the first VM had two VCPUs and the second VM had four VCPUs;

- The synchronization rate is 1:5;
- The number of PCPUs is four across all simulations.

Figure 9 presents the simulation results (with 95% confidence level and <0.1 confidence interval). The results show that when the number of VCPUs is larger than the number of PCPUs, the two co-scheduling algorithms cannot fully utilize the PCPUs. This is caused by the CPU fragmentation problem mentioned in Section II.B. The results also show that the relaxed co-scheduling significantly mitigates this problem, as it can always achieve more than 90% PCPU utilization.

C. Virtual CPU Utilization

In this set of simulations, we evaluated the VCPU utilization resulting from the three scheduling algorithms with different VM setups. VCPU utilization measures the portion of time that a VCPU is used to process workloads during the simulation. This metric reveals the synchronization latency, as we configured the workload generation to be interrupted only when synchronization points block the VMs. In order to obtain the average utilization of all the VCPUs, we defined a reward variable (in the SAN model) that monitors the READY and BUSY states of all the VCPUs. The setup of the experiment is as follows:

- Three sets of VMs: (set 1) each VM had two VCPUs; (set 2) the first VM had two VCPUs and the second VM had three VCPUs; (set 3) the first VM had two VCPUs and the second VM had four VCPUs;
- The synchronization rate is varied from 1:5 to 1:2;
- The number of PCPUs is four across all simulations.

Figure 10 presents the results of the simulation (with 95% confidence level and <0.1 confidence interval). When the number of VCPUs is the same as the number of PCPUs (with the first set of VMs), the VCPU utilization is high and we do not see any difference among the scheduling algorithms. However, when the number of VCPUs is greater than the number of PCPUs (with the second and third set of VMs), the results show the co-scheduling algorithm reducing synchronization latency. The strict co-scheduling achieves the highest VCPU utilization, followed by relaxed co-scheduling. Due to the relaxed requirement of the co-start, the VCPU utilization of RCS is slightly lower than that of SCS. According to the PCPU utilization measurement presented in Section IV.B, SCS archives much better PCPU utilization when compared to SCS. These measurements demonstrate that RCS is better than SCS. Round-Robin scheduling, on the other hand, is significantly affected by the synchronization rate. As the synchronization rate increases, VCPU utilization quickly degrades.

V. DISCUSSION

This is a flexible simulation framework that aids in the evaluation of VCPU scheduling algorithms. However, the framework at this current state still has several limitations:

The Workload model is still primitive. It needs improvements in order to (i) include other resource requirements, such as memory, network bandwidth, and (ii) represent more synchronization mechanisms.

The model cannot be used to debug problems, which impact the correctness of guest operation. For example, some people suspect that the long synchronization latencies caused by VCPU scheduling could violate the assumptions of some locking mechanisms (e.g. spinlocks assuming that the critical sections are short).

Evaluating the fidelity of the model. At this state, we did our best to simulate the virtualization environment. But more thorough evaluation is needed to validate our model.

VI. CONCLUSION

We have presented the design and construction of a simulation framework for evaluating VCPU scheduling algorithms. The framework is built upon SAN models and the Mobius tool, making the framework easy to understand and configure for various virtualization setups. We demonstrate the usefulness of the framework by evaluating three VCPU scheduling algorithms: Round-Robin, Strict Co-Scheduling, and Relaxed Co-Scheduling.

ACKNOWLEDGEMENT

This work was supported in part by NSF grant CNS 10-18503 CISE the Department of Energy under Award Number DE-OE0000097, the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084, the Defense Threat Reduction Agency under award no. HDTRA1-11-1-0008, Boeing Corporation, and Infosys Ltd.

REFERENCES

[1] O. Sukwong, and H. Kim. "Is co-scheduling too expensive for SMP VMs?" In Proc. of the 6th conf. on Computer systems, ACM, 2011.

[2] VMWare Inc., "VMware® vSphere™: The CPU Scheduler in VMware ESX® 4.1", Techincal Report.

[3] VMWare Inc., "Co-scheduling SMP VMs in VMware ESX Server", http://communities.vmware.com/docs/DOC-4960

[4] U. Schwiegeishohn and R. Yahyapour. "Improving first-come-first-serve job scheduling by gang scheduling." In Job Scheduling Strategies for Parallel Processing, Springer Berlin/Heidelberg, 1998.

[5] W. Sanders, and J. F. Meyer. "Stochastic Activity Networks: Formal Definitions and Concepts." Lectures on Formal Methods and Performance Analysis (2001): 315-343.

[6] W. Sanders, and J. F. Meyer. "A unified approach for specifying measures of performance, dependability, and performability." Ann Arbor 1001 (1991): 48109.

[7] C. Weng, et al., "The hybrid scheduling framework for virtual machine systems," in *Proc. of the 2009 ACM SIGPLAN/SIGOPS intl' conf. on Virtual execution environments*, New York, NY, USA, 2009.

[8] L. Cherkasova, et al., "Comparison of the three CPU schedulers in Xen." Performance Evaluation Review 35, no. 2 (2007): 42.

[9] Clark, Graham, Tod Courtney, David Daly, Dan Deavours, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick Webster. "The Mobius modeling tool." In Proc. Petri Nets and Performance Models, 2001.