SRVC: An Efficient Scheduler for Concurrent Virtual Machines over the Xen Hypervisor

Po-Jen Chuang* and Chih-Yuan Chou

Department of Electrical and Computer Engineering, Tamkang University, Tamsui, Taiwan 25137, R.O.C.

Abstract

A CPU scheduler can support virtualization practices by distributing physical resources to form virtual machines (VMs). To facilitate concurrent applications in virtualization platforms, this paper presents an SRVC scheduler which is a Scheduler involving the R-B Tree structure, the Virtual runtime of the CFS mechanism and a Concurrent waiting queue. The SRVC scheduler can practically cut down the overall runtime by using the R-B Tree structure to shorten the virtual CPU (VCPU) lookup time and attain efficient task scheduling. It meanwhile applies the CFS mechanism to reduce the gap between concurrent and non-concurrent VMs and obtain balanced CPU time distribution and resource utilization. To promote synchronization on concurrent VCPUs and scheduling, SRVC takes in a concurrent waiting queue to handily pick up marked VCPUs for arranged execution. As simulation results demonstrate, SRVC needs constantly less runtime than existing Credit and Turbo schedulers due to its ability to reduce the runtime in both concurrent and non-concurrent applications.

Key Words: Virtualization, The Xen Hypervisor, Concurrent Virtual Machines, Scheduling, Performance Evaluation

1. Introduction

Virtualization techniques [1,2] have significantly improved the hardware availability during planned or unplanned shutdown and have hence brought revolutionary changes to modern IT industries. By distributing computing resources to a wide scope of applications, virtualization techniques can practically magnify the utilization of hardware resources to save the required numbers of physical machines. The derived benefits are enormous and unprecedented. Current virtualization platforms, including Xen [3], VMware [4] and KVM [5], use such physical resources as CPU, memory, network interface cards or storage to establish fully functional virtual machines (VMs) which can execute operating systems and applications like real or physical computers. In practicing virtualization, we must consider how to allocate these physical resources efficiently to form the desired VMs. Among virtualization platforms, the Xen hypervisor is a powerful open-source industry standard. Its configuration settings, including the virtual CPU (VCPU), virtual memory and CPU scheduler, appear in a good number of VM architectures. Our focus in this investigation is to develop a new CPU scheduler for Xen to facilitate virtualization practices in a more efficient way.

Credit [6] and Turbo [7,8] are two existing CPU schedulers for Xen. In virtualization practices, if we use the asynchronous Credit scheduler to handle a concurrent application whose tasks must be executed simultaneously, we may explicitly waste the CPU runtime. This is because Credit does not consider that all tasks in a concurrent application need communication and synchronization to perform the concurrent execution and
therefore needs extra runtime to handle it. The Turbo scheduler – based on priority settings – is proposed to improve the concurrent execution problem, especially the time-consuming issue in Credit. Its basic idea is to mark the Turbo priority to each concurrent task and move the marked tasks to the top of the run queue during scheduling. Such a design helps Turbo outperform Credit in runtime but only partially – because, for task lookup, Turbo adopts the same task execution run queue structure as Credit which contains other time-consuming operations, such as insertion, deletion and searching. To further reduce the required runtime, i.e., to handle concurrent tasks more efficiently, we come up with a new CPU scheduler in this paper. We build the new Scheduler on the Red-Black (R-B) Tree structure [9, 10], the Virtual runtime concept of the completely fair scheduler (CFS) [11] and a handy Concurrent waiting queue. The new scheduler is hence briefed as SRVC. In practice, SRVC adopts the R-B Tree to replace the original run queue in both Credit and Turbo and succeeds in cutting down the task lookup time. It also uses the CFS mechanism to reduce the gap between concurrent and non-concurrent VMs and achieve fairer CPU time distribution. By balancing CPU time distribution between concurrent and non-concurrent applications, SRVC improves the concurrent problem in Credit and Turbo and, as a result, attain higher efficiency than both schedulers. SRVC meanwhile sets up a handy concurrent waiting queue to mark VCPUs awaiting execution and follows the queue – which is simple but effective in synchronizing concurrent VCPUs and scheduling – to sort them out for execution.

Extended simulation runs have been conducted to evaluate the performance of the three target schedulers – Credit, Turbo and our SRVC – in terms of runtime. Note that, during concurrent VM execution, we gradually adjust the VM weights to decrease the virtual runtime in the R-B Tree so that a physical CPU (PCPU) can select the most proper VCPUs for execution. The obtained simulation result shows that, in contrast to Credit and Turbo, our SRVC yields significantly shortened global runtime. That is, the proposed SRVC generates substantially better performance than the other two schedulers: it reduces the required runtime for both concurrent and non-concurrent applications.

2. Background Study

2.1 The Xen Architecture [3]

The Xen architecture contains the Xen hypervisor and a number of VMs (Domains) and physical machines. Xen modifies the Linux kernel to load the Xen hypervisor at boot and to start the Domain-0 operating system on top as the host OS. Domain-0 mainly takes control of other Domain-U (U > 0) which act as the guest OS, and will be loaded first at boot as it contains the control commands to start the other VMs. It is the main system to control and manage the Xen hypervisor, and also the only VM that has native device drivers via which to communicate with physical devices. The Xen hypervisor will install a front-end device driver in each Domain-U so that it can send a hypercall to Domain-0 to request a needed device service.

In a Xen system, each Domain has its own VCPU structure. Tasks over the guest OS will be first executed in VCPUs and then passed to PCPUs. For Xen, VCPUs are similar to the threads in a traditional OS which are scheduled and executed in PCPUs. When the kernel scheduler in the guest OS schedules the threads in each Domain to VCPUs, the scheduler in the hypervisor will schedule all VCPUs to the run queues of PCPUs. That is, the hypervisor will determine which VCPU is to be executed next in a PCPU. The VCPU scheduling is important in lifting the performance of the whole VM system: it can prevent a certain VCPU from occupying too much PCPU execution time and also prevent too many VCPUs from being executed in a few PCPUs only. Proper VCPU scheduling can produce fairer Domain scheduling as well as a more balanced system load, and substantially enhance the performance of the entire VM system [12].

2.2 The Credit Scheduler [6]

Credit is a proportional fair share CPU scheduler adopted by Xen since Version 3.0. Designed to be light weight and efficient, it sets a parameter tuple (weight, cap) for each VM and allows the guest OS in each VM to decide how much of a PCPU the VM can consume – according to its parameter weight relative to the weights of other VMs. For example, if VM1 and VM2 are respectively with weights 256 and 512, VM2 will have a doubled chance (in contrast to VM1) to get its VCPUs sche-
The parameter cap, on the other hand, indicates the maximum amount of PCPU a VM may consume. For instance, if the cap of a VM is 50 (or 100), indicating the VM can consume up to 50% (or 100%) of a PCPU. Each PCPU in the scheduler manages a local run queue of runnable VCPUs. It sorts the queue by the VCPU priority, picks the next VCPU to run from the head of the queue and assigns the selected VCPU a 30-ms time slice (the time slice by default is fixed at 30 ms). Each running VCPU will receive a "timer interrupt" every 10 ms to calculate the credit consumption over the 30-ms time slice. If a VCPU does not complete its execution in the pre-set 30 ms, it will be removed and put after all other VCPUs of the same priority. The priority of a VCPU can be over – having exceeded its fair share of the PCPU resource, or under – not yet exceeded. All VCPUs initially carry the under priority.

When a VCPU runs, it starts to consume the credit value. It will keep the under priority before consuming all allocated credit value, and get the over priority after using up all of the value. The Credit scheduler calculates the initial credit value of each VM by its weight and number of VCPUs, and will decrease the credit value of a VM each time when one of its VCPUs gets scheduled. When the credit value of a VM turns negative, the scheduler will adjust the priority of its VCPUs to over. At this point, a VCPU not yet finishing execution will be moved to the end of all other VCPUs which have the same priority. The priority of a VCPU can be over – having exceeded its fair share of the PCPU resource, or under – not yet exceeded. All VCPUs initially carry the under priority.

When a VCPU runs, it starts to consume the credit value. It will keep the under priority before consuming all allocated credit value, and get the over priority after using up all of the value. The Credit scheduler calculates the initial credit value of each VM by its weight and number of VCPUs, and will decrease the credit value of a VM each time when one of its VCPUs gets scheduled. When the credit value of a VM turns negative, the scheduler will adjust the priority of its VCPUs to over. At this point, a VCPU not yet finishing execution will be moved to the end of all other VCPUs which have the same priority. The priority of a VCPU can be over – having exceeded its fair share of the PCPU resource, or under – not yet exceeded. All VCPUs initially carry the under priority.

If a PCPU cannot locate a VCPU with the under priority in its local run queue, it will look up other PCPUs for one. This load balancing practice guarantees that each VM will receive the fair share of the PCPU resource. Besides, before a PCPU goes idle, it will look up other PCPUs for any runnable VCPU to assure that no PCPU idles when there are runnable tasks awaiting execution in the system. As we can see, the Credit scheduler schedules all VCPUs to PCPUs in an assumed non-synchronized fashion to achieve the maximum PCPU utilization. However, employing weights to ensure available PCPUs for VCPUs may significantly waste the PCPU time. In concurrent execution, as mentioned, VCPUs must communicate and synchronize with each other and tasks in respective VCPUs must be completed concurrently. If some tasks are not completed in the current time slice, no tasks in any VCPU can be brought in for execution in the next time slice: they need to wait until all previous concurrent tasks are executed. Figure 1 gives an example to illustrate the time-consuming problem incurred by the PCPU scheduling of the Credit scheduler. Assume that the credit value of a VM allows each VCPU 3 time slots for 3 subsequent tasks (numbered as 1, 2 and 3) in the next 10 time slots. Synchronization must be performed among VCPUs at the end of each task (indicated by the dark box). Before all current tasks (with the current numbers) complete their execution, the next task (indicated by the next number) in any VCPU cannot be executed. As Figure 1 reveals, the PCPU scheduling keeps task 1 of VCPU2 from being executed until the 5th time slot. The act, in turn, forces task 2 of VCPU3 (scheduled in the 3rd time slot) and task 2 of VCPU1 (scheduled in the 5th time slot) to be cancelled (indicated by the dotted crosses) and re-scheduled at latter time slots, obviously wasting the PCPU time and prolonging the total runtime.

### 2.3 The Turbo Scheduler [7,8]

Turbo aims to reduce the PCPU time consumed by the required concurrent execution of VCPUs in Credit. It uses the Turbo priority to mark VCPUs awaiting concurrent execution. All marked VCPUs will be selected to run at the next time slice, to avoid the overall runtime waste due to synchronization. The basic practice is to set the Turbo priority to a VCPU whose ID is 0 (VCPU0), search the rest of the VCPUs in the same concurrent VCPU, and tag them all with the Turbo priority. Figure 2 illustrates the practice of Turbo. It depicts each VCPU as a square with a tuple (Domain, VCPU ID). Each PCPU

---

Figure 1. The PCPU time waste of the Credit scheduler: an example.
has its local run queue and the next VCPU to run is picked off the head of the queue. Figure 2(a) uses the bold red tuples (1,0)–(1,3) to indicate VCPUs marked with the Turbo priority. Figure 2(b) puts the marked VCPUs at the heads of corresponding PCPU run queues, to avoid the PCPU runtime waste due to concurrent VCPU execution.

By marking the Turbo priority to VCPUs awaiting concurrent execution, Turbo can reduce the overall runtime waste due to synchronization. But, as it adopts the task execution run queue structure of Credit for task lookup, it still needs extra runtime to perform certain time-consuming operations like insertion, deletion and searching. For instance, it needs extra $O(n)$ runtime when $n$ VCPUs are waiting in the concurrent run queue. Besides, assigning each selected VCPU a 30-ms time slice will make Turbo consume another 30 ms waiting time if unable to locate a concurrent VCPU with ID = 0 in searching the run queue and setting the priority for VCPUs of one concurrent VM.

3. The Proposed Scheduler

To improve the performance of previous schedulers, i.e., to remove existing limitations and weaknesses, we devise an advanced SRVC scheduler in this investigation. Our main purpose is to handle VCPUs in one concurrent VM and look them up in the PCPU queues in a more efficient way. Our SRVC involves the completely fair scheduler (CFS, used by Linux kernel since Version 2.6) – especially its concept of preemptive multitasking [13] – to reduce the gap between concurrent and non-concurrent VMs. Employing the CFS mechanism and the concept of virtual runtime to perform task scheduling can secure more balanced (fairer) PCPU time distribution and better efficiency, to improve the concurrent problem in Credit. SRVC meanwhile dismisses the use of the original run queue structure in Credit and Turbo, and instead adopts the RB-Tree structure to shorten the VCPU lookup time. To assist synchronization on concurrent VCPUs, it also sets up an efficient handy concurrent waiting queue to mark awaiting VCPUs and arrange their execution.

The following demonstrates the operations of SRVC.

3.1 Virtual Runtime in CFS [11]

Designed to provide fair task scheduling, CFS has been officially used in Linux kernel since Version 2.6.23. SRVC brings it in to improve the run queue problem so as to shorten the overall runtime. Virtual runtime (Vruntime) is a critical concept in CFS. We can obtain the Vruntime of VCPUs by increasing or decreasing – based on the scale of their weights – a fixed proportion of the actual runtime. The value of Vruntime can lead to fairer scheduling as it reflects the overall amount of CPU time a current task has consumed and is hence a proper measure to help decide if the current task should be removed from the ongoing execution in order to start a new task. We can attain the Vruntime of a VCPU by the following formula:

$$\Delta \text{Vruntime} = \Delta \text{Time} \times \frac{\text{Default\_Weight}}{\text{Weight}}$$

$\Delta \text{Time}$ indicates the time between two consecutive Vruntime computations, and Default\_Weight is a predetermined fixed value. We can obtain and adjust the Vruntime of a VCPU from multiplying $\Delta \text{Time}$ by the ratio of (the default weight)/(the weight of the VCPU). Clearly, when the weight of a VCPU grows, the ratio will drop and so will the Vruntime, and when the weight of a VCPU goes down, both the ratio and Vruntime rise up. After obtaining the Vruntime, we incorporate it into the RB-Tree to work out task scheduling and select the next VCPU for execution. By practicing the RB-Tree algorithm, we can schedule all VCPUs that attain the Vruntime: the VCPU with the smallest Vruntime will be
placed to the leftmost side of the tree and be first picked up for PCPU execution in the next run.

3.2 The Red-black (R-B) Tree Algorithm [9,10]

After attaining the Vruntime of all waiting VCPUs, we perform the R-B Tree algorithm to sort out and decide their execution priority. The R-B Tree is a self-balancing binary search tree in data structures. Its self-balancing practice can guarantee searching in O(log n) time, n is the total number of elements in the tree. The O(log n) time covers the insertion/deletion and tree rearrangement/recoloring [10]. To work out task scheduling, we first input the obtained Vruntime of all waiting VCPUs into the R-B Tree. The tree will conduct the insertion operation according to the “key” values (the Vruntime values in our scheduler) and moves the smallest key value (the VCPU with the shortest Vruntime) to the leftmost side of the tree. Such a practice helps us attain the sequence of task scheduling efficiently. We can decide the scheduling priority of all VCPUs according to their Vruntime values and handily work out the execution sequence. That is, we can always pick up the VCPU at the leftmost node of the R-B Tree (which has the shortest Vruntime) for the next execution.

3.3 The Scenario of the R-B Tree and Vruntime

Figure 3 gives an example to illustrate the operation of Vruntime and the R-B Tree. Assuming there are three VCPUs – A, B and C – awaiting execution in the system, and they are respectively with ΔVruntime 5, 4 and 3, namely, ΔVA = 5, ΔVB = 4 and ΔVC = 3. The system maintains a number of variables, including Leftmost which caches the leftmost node of the R-B Tree, Curr which marks the VCPU currently under procession and Min Vruntime which indicates the shortest Vruntime in the tree.

In Figure 3(a), assume the Vruntime for A is 100 (VA = 100) and A is the first node to enter the R-B Tree, i.e., the root node of the tree. Variable Leftmost will cache the pointer pointing to A which is in the leftmost node of the tree and PCPU will pick A for execution. A is now marked as Curr because it is the node currently under execution. Min Vruntime then records 100 as the smallest Vruntime in the tree. The next node to enter the tree will carry this min Vruntime (100) to get a better chance of being the leftmost node.

A node will be removed from the R-B Tree after being selected for execution. As Figure 3(b) shows, after A is executed, its Vruntime VA becomes 100 + 5 = 105 (5 is ΔVruntime, the increment value of VA in this run). VA is then inserted back to the R-B Tree (with the new value 105). In Figure 3(c), a new node B is inserted to the R-B Tree with the new min Vruntime value 105. It gets to the leftmost position of the tree (Figure 3(d)) and will be taken in the next execution (Figure 3(e)).

After B is executed, we will update its Vruntime by adding ΔVB (4) to VB (105) to get a new VB (109) and put B with the new value back to the right of the R-B Tree (Figure 3(f)). To insert another node C into the tree, we can get its Vruntime by the current Min Vruntime (105) and make it reach the leftmost side of the R-B Tree, to be selected for execution in the next run. After execution, C will go back to the tree with the new Vruntime 108 (from adding ΔVC (3) to the current Min Vruntime (105)). (Fig-

Figure 3. The scenario of the R-B Tree and Vruntime.
In the operations of Figures 3(a)–(h), the immediate results of an insertion or removal seem to violate the self-balancing property of the R-B Tree. To restore the property, we need a small number – $O(\log n)$ – of color changes which are quick in practice and take at most three tree rotations, as demonstrated in Figures 3(i)–(k).

After the restoration process ends in Figure 3(k), the VCPU currently located at the leftmost side of the R-B Tree is A which has the smallest Vruntime value ($V_A = 105$), in contrast to B ($V_B = 109$) and C ($V_C = 108$), and will be selected for the next execution. It shows that, incorporating Vruntime into the practice of the R-B Tree, we can always pick up a VCPU with the shortest Vruntime for execution.

### 3.4 The Concurrent Waiting Queue

As mentioned, both Credit and Turbo set up concurrent VCPU scheduling by the concurrent task run queue which requires considerable VCPU lookup time. To handle concurrent VCPU scheduling more efficiently, our SRVC uses a concurrent waiting queue (con-wait-queue) to set the run queue. We first use a flag to mark each concurrent VCPU (CON-VCPU) awaiting execution. Before inserting VCPUs into the R-B Tree to work out the running schedule, we will sort them into two groups: unmarked non-CON-VCPUs and marked CON-VCPUs. After the sorting, we insert non-CON-VCPUs into the R-B Tree and put CON-VCPUs into the con-wait-queue. A set of triple indexes ($PCPU$, $VCPU$, and $Domain$) is used to denote CON-VCPUs waiting to enter the con-wait-queue and to insert them to proper positions. Of the three indexes, $PCPU$ will take a CON-VCPU to the con-wait-queue of a proper PCPU and $Domain$ will take it to the proper position in the con-wait-queue. A one-dimensional array $con\_bit$ is built to monitor all CON-VCPUs and decide their bit positions according to $VCPU$.

Figure 4 gives an example to show the forming of our con-wait-queue. Assume the index set of a CON-VCPU about to be inserted into the waiting queue is ($PCPU1$, $VCPU1$, $Domain2$). $PCPU1$ indicates the CON-VCPU belongs to the con-wait-queue of PCPU1 and $Domain2$ specifies it shall go to the second position in the queue. $con\_bit$ [$Domain \ ID$] helps monitor and check if all CON-VCPUs in the same VM have been inserted into the con-wait-queue. In this case, ($PCPU1$, $VCPU1$, $Domain2$) will make us select $con\_bit$ [2] and set bit 1 from 0 to 1, to indicate this CON-VCPU has entered the right position of the right con-wait-queue.

After all CON-VCPUs enter the con-wait-queue, we pick them up and insert them, one by one, into the left side of the leftmost node in the R-B Tree. In this way, we help all CON-VCPUs get to the leftmost position of the tree and be picked up, one by one, for execution by the PCPU. Note that before CON-VCPUs are put into the tree, we will cache the pointer pointing to the previous leftmost node into $pre\_leftmost$ for future quick reference. We will remove a CON-VCPU from the R-B Tree after it is executed. When the last CON-VCPU gets executed and removed, the previous leftmost node becomes the leftmost node again and a new run of execution will begin with the VCPU at the leftmost position of the tree. Figure 4 exhibits clearly that selecting CON-VCPUs from our con-wait-queue is an efficient time-saving $O(1)$ operation. For better illustration, we use the algorithm in Figure 5 to summarize the proposed SRVC mechanism.

### 3.5 Scheduler Comparisons

Among the three schedulers, Credit is the only one without particular considerations on concurrent VCPU execution. It ignores the fact that all tasks in a concurrent application must communicate and synchronize to meet the concurrent execution requirement (which takes extra runtime). In its practice, Credit acquires available PCPUs for VCPUs by weights and uses the time-consuming run queue structure to trace tasks, thus generating $O(n)$ time complexity and explicitly wasting the PCPU runtime. To
solve the concurrent problem, Turbo uses priority tags to mark concurrent tasks and move the marked tasks to the top of the PCPU run queue during task scheduling. Turbo hence advances Credit in concurrent execution. It nevertheless wins no proper performance gain because of tracing tasks also by the time-consuming run queue structure of Credit.

In SRVC, we employ the CFS mechanism and the concept of virtual runtime to reduce the gap between concurrent and non-concurrent VMs and by doing so we can distribute the CPU time more fairly to pursue desirable PCPU resource utilization. We also adopt the R-B Tree (instead of the original run queue) to help trace and select tasks, and ultimately cut task lookup time complexity from \( O(n) \) to \( O(\log n) \). In practice, we can further reduce time complexity. For instance, when updating the R-B Tree, we can meanwhile cache the candidate VCPU (which the PCPU will pick up for the next execution) to further save the lookup time. In handling concurrent applications, both SRVC and Turbo will collect all CON-VCPUs of the same VM and move them to the head of the PCPU to await execution. In this practice, Turbo needs \( O(n) \) time complexity to look up those marked VCPUs from the PCPU run queue. In SRVC, we can directly pick up the marked CON-VCPUs from the right position of our con-wait-queue and instantly locate them to the left child node of the leftmost node in the R-B Tree, to get them ready for PCPU selection and execution. Involving no additional lookup, we can complete the practice by only \( O(1) \) time complexity. When the concurrent runtime drops, the overall runtime drops as well. In other words, when the concurrent performance improves (due to reduced time complexity), the overall performance improves too.

For better reference, we list the key features of the three schedulers in Table 1. As we can see, SRVC uses Vruntime and the R-B Tree to establish task scheduling, hence its scheduling operation takes only \( O(\log n) \) – a notable reduction in contrast to the \( O(n) \) runtime of the linear queue operation in the other schedulers. In handling the concurrent application problem, Turbo uses flags to mark awaiting CON-VCPUs, looks up all marked CON-VCPUs in run queues and picks them up for execution. SRVC adopts a different approach. It puts each waiting CON-VCPU in the con-wait-queue and will start execution only when all of them are ready in the queue. SRVC hence takes only \( O(1) \) time complexity to get all CON-VCPUs in the con-wait-queue executed by the PCPU, largely reducing the time complexity.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PCPU’s queue operation</td>
<td>1. Linear queue: run queue 2. Time complexity: ( O(n) )</td>
<td>1. Linear queue: run queue 2. Time complexity: ( O(n) )</td>
<td>1. R-B Tree + Vruntime 2. Time complexity: ( O(\log n) )</td>
</tr>
<tr>
<td>Solving concurrent applications</td>
<td>NO</td>
<td>1. Tag priority: Turbo 2. Trace Turbo tags: ( O(n) )</td>
<td>1. Con-wait-queue 2. Time complexity: ( O(1) )</td>
</tr>
</tbody>
</table>

**Figure 5.** The algorithm for the proposed SRVC mechanism.

```plaintext
for each VCPU awaiting execution {
    attain its Vruntime;
    if it is a CON-VCPU
        insert into con-wait-queue according to triple index (PCPU, VCPU, Domain);
    else {
        insert into the RB-tree according to its Vruntime (shortest to the leftmost);
        cache the pointer pointing at the leftmost node in the RB-tree;
    }
    if all CON-VCPUs in the same VM have been inserted into the con-wait-queue according to array con_bit
        insert them, one by one, into the left side of the leftmost node in the R-B Tree;
    pick up the VCPU at the leftmost node of the R-B Tree (which has the shortest Vruntime) for execution in the next run;
```
4. Experimental Evaluation

4.1 The Experiment Model

To compare the performance of the three target schedulers, we carry out an experimental evaluation over the Intel Haswell Xeon E3-1230V3 platform with a 3.3 GHz processor. The system is installed with 16 GB DDR III RAM and 2 TB SATA II hard disk. The host OS in Dom0 is Ubuntu 12.10 with kernel 3.0.5 and Xen Hypervisor 4.1.4, and the guest OS in DomU is Ubuntu 12.10 with kernel 3.0.5. 1 GB memory, 4 VCPUs and 8 GB hard disk space are allocated for each VM. We adopt the NAS Parallel Benchmarks (NPBs) [14,15] and exhibit the two benchmarks NPB-LU.4.A and NPB-EP.4.A (briefed as LU and EP hereafter) in Table 2.

4.2 Runtime

To begin the simulation, we implement Benchmarks LU and EP on one VM and gradually increase the working VM from 1 to 8 VMs. Each VM has 4 VCPUs and performs a NPB program. When the number of VMs increases to 8, there will be a total of 32 VCPUs in the system waiting to get the resources of 4 PCPUs, and the 8 VMs will execute a total of 8 NPB programs. We then conduct the experiment respectively under LU and EP, to check and compare the performance of the three target schedulers in terms of runtime.

Figure 6 gives the results of runtime obtained under Benchmark LU. When VMs execute concurrent applications under Benchmark LU, the involved CPUs must communicate with each other to get the needed synchronization information first and then to perform the operation accordingly. As CPUs must spend time waiting for the synchronized acts, such concurrent applications may consume significant waiting time. The results in Figure 6 reveal that the Turbo scheduler requires less runtime than the Credit scheduler because, unlike Credit which has no special considerations on concurrent problems, Turbo uses priority tags to mark VCPUs awaiting concurrent execution and to sort them out for subsequent execution. That is, it saves the time consumption due to synchronization. Of the three schedulers, our SRVC takes the shortest overall runtime. We take less time to finish task execution and obtain better efficiency than the other two schedulers all because our efficient scheduling mechanism helps attain more balanced CPU time distribution, less VCPU lookup time and better synchronization on concurrent VCPUs.

Figure 7 gives the runtime results obtained under Benchmark EP. The results indicate very close performance for the three schedulers. In contrast to their performance under Benchmark LU, they now take closely shorter runtime to finish task execution. This is because, under Benchmark EP, they can complete task execution without engaging the time-consuming inter-CPU communications. That is, they can move directly to the next task, after finishing a previous one, to continue the execution process without delay.

4.3 Weights vs. Runtime

We assume the system has one concurrent VM (CON-VM) and seven non-CON-VMs, and carry out the

<table>
<thead>
<tr>
<th>Table 2. Benchmark parameters: LU and EP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Embarrassingly Parallel (EP)</td>
</tr>
<tr>
<td>Lower-Upper Gauss-Seidel solver (LU)</td>
</tr>
</tbody>
</table>

![Figure 6. Runtime comparisons under Benchmark LU.](image)
simulation respectively by CON-VM weight values 32 and 512 to check how CON-VM weight changes correlate to the weight changes of non-CON-VMs. We first set the CON-VM weight at 32 and gradually increase the non-CON-VM weights from 32, 64, 128, 256, to 512 – to obtain the runtime at each non-CON-VM weight value. This simulation involves the following three performance parameters:
1. The average runtime of non-CON-VMs (denoted as noncon_avg),
2. The runtime of the CON-VM (con), and
3. Speedup.

Figure 8 depicts the runtime results collected at CON-VM weight 32. At CON-VM weight 32, Turbo is shown with longer runtime than Credit in both CON-VM and non-CON-VMs, i.e., Turbo generates longer con and noncon_avg. CON-VM weight 32 is such a small value that gives Turbo only limited credits to schedule concurrent applications. Recall that Turbo needs to mark all VCPUs of one CON-VM with priority tags and will start the execution process only when all marked VCPUs are moved to the head of a PCPU. Hence, with such a small CON-VM weight, Turbo can hardly get all marked VCPUs of a CON-VM scheduled and executed – because those marked VCPUs may easily exceed the CON-VM weight and fail to reach the head of the PCPU. When it happens, Turbo must quit this concurrent application scheduling and yield the scheduling/execution chance to non-concurrent applications. When handling non-concurrent applications, Turbo actually takes more cumulative time than Credit because of adjusting the PCPU queue at each run. SRVC, by contrast, performs notably better as it yields much less runtime in either CON-VM (con) or non-CON-VMs (noncon_avg). The good performance results mainly from its efficient load-balancing design which reduces the gap between CON-VMs and non-CON-VMs, to generate fairer PCPU time distribution and less VCPU lookup time.

The runtime results in Figure 9 exhibit that, when the CON-VM weight is increased to 512, the system can execute tasks more effectively – at less runtime – in general. Task execution takes less runtime at CON-VM weight 512 mainly because VCPUs of the CON-VM now have higher probability to maintain the under priority and be scheduled for execution. It is clear that the runtime of the CON-VM will grow with the weights of non-CON-VMs. We can expect shorter CON-VM runtime when the CON-VM weight exceeds that of the non-CON-VMs – because VCPUs in the CON-VM will have a better chance to get the PCPU resource for scheduling and execution. However, when non-CON-VM weights rise, the CON-VM runtime will gradually grow to exceed that of non-CON-VMs. It happens to all schedulers because, when non-CON-VM weights rise, the CON-VM needs to compete with non-CON-VMs for the PCPU

Figure 8. Runtime comparisons at CON-VM weight 32.

Figure 9. Runtime comparisons at CON-VM weight 512.
resource until finally losing the competition. Figure 9 shows that, in all situations, our SRVC yields constantly shorter runtime than Credit and Turbo for both the CON-VM and non-CON-VMs, due to its favorable design.

4.4 Weights vs. Speedup

\[
\text{Speedup} = \frac{\text{Runtime before improvement}}{\text{Runtime after improvement}}
\]

We use the above defined speedup to check the performance gain of SRVC and Turbo (over Credit). More specifically, speedup is “the original runtime of Credit” over “the improved runtime of SRVC/Turbo”. The obtained speedup values will indicate how much SRVC and Turbo advance in reducing the runtime. We attain the runtime from running Benchmarks LU/EP at CON-VM weights 32/512 and non-CON-VM weights 32-512, and give the speedups of Credit/Turbo and Credit/SRVC in Figure 10.

Clearly, if Turbo and SRVC outperform Credit in runtime, the speedup value will exceed 1.00 and a bigger value indicates more improvement. As Figure 10 shows, at CON-VM weight 32, the speedup value of Credit/Turbo is constantly below 1.00 – exhibiting Turbo cannot exercise its design to advance Credit at small CON-VM weights. The situation changes at CON-VM weight 512 where the speedup of Credit/Turbo constantly exceeds 1.00 – revealing Turbo steadily outperforms Credit and the value rises to 1.50 when the non-CON-VM weight drops to 32. It displays that, with higher CON-VM weights, Turbo can practically improve Credit especially at low non-CON-VM weights.

SRVC performs even better. At CON-VM weight 32, speedups of Credit/SRVC stay above 2.50 for all situations (all non-CON-VM weights) and the value grows when the non-CON-VM weight shrinks. It grows over 3.00 when non-CON-VM weights drop to 64 and 32. At CON-VM weight 512, we see even more distinct improvement. The speedup of Credit/SRVC now rises between 3.00 and 3.50 at non-CON-VM weights 512, 256 and 128. At non-CON-VM weight 32, it reaches over 4.50 – in contrast to 1.50 of Credit/Turbo in the same environment.

4.5 Other Discussions

Besides SRVC, a number of schedulers are newly proposed to facilitate VM scheduling in Xen. For instance, [16] presents an SRT-Xen scheduler which brings in the soft real-time workloads in virtualization systems and implements a fair scheduling mechanism for both real-time and non-real-time tasks, to promote the performance of soft real-time domains. [17] attempts to dynamically schedule the CPU resources according to the resource statuses diagnosed by allocated and consumed credits, to make full use of the underlying physical resources of Xen VMs. There are also schedulers built over the concept of disk I/O fairness among DomUs or memory competition among VMs [18,19]. We believe that, if these schedulers are brought to work with our SRVC, the scheduling performance in Xen can be further enhanced.

5. Conclusions

Our research focus in this paper is to facilitate concurrent applications in virtualization practices. The main goal is to pursue more desirable hardware resource allocation and better task scheduling in order to shorten the runtime of both virtual and physical operations. To fulfill the goal, we build a new CPU Scheduler – SRVC – based on the R-B Tree structure, the Virtual runtime concept of the CFS mechanism and the proposed Concurrent waiting queue. The SRVC scheduler adopts the R-B Tree structure to shorten the VCPU lookup time and arrange task scheduling in a more efficient way. It uses the virtual runtime in the CFS mechanism to reduce the gap be-
tween concurrent and non-concurrent VMs and, as a result, to reach more balanced PCPU time distribution as well as resource utilization. In SRVC, we meanwhile develop a concurrent waiting queue to replace the original run queue structure of previous schedulers. The concurrent waiting queue helps us attain more desirable concurrent task scheduling so that we can handily pick up the marked VCPUs and arrange them for execution. The obtained simulation result has verified the advanced performance of our SRVC scheduler in contrast to that of other schedulers. It shows that our new scheduler can constantly reduce the overall runtime, including the runtime for both concurrent and non-concurrent applications, to enhance the overall system performance markedly.

References


Manuscript Received: Jan. 12, 2016
Accepted: Apr. 28, 2017