CHAPTER 5

RESEARCH METHODOLOGY

5.1 Research Methodology

From the existing literatures in the research topic, various factors were found to be responsible for the performance of the paravirtualized guests in an paravirtualized environment. Various experiments were carried out in order to find out the impact of these factors and results were obtained. From the experiments conducted, it was noticed that out of the various possible improvements in the functioning of xen hypervisor, an enhancement in the CPU scheduling algorithm for the xen hypervisor could lead to bettter results. Based upon the results from the initial experiment, a modified credit scheduling algorithm, which accounts for the dynamic workload of the guest domains in runtime was developed, which was then tested with the original credit scheduler algorithm, and results were obtained with improvements in performance than the original scheduling algorithm. The experimental setup and the detailed methodology for the research are given in the following subtopics.

5.2 Experimental System Setup:

Figure 5.1: Physical Server for the experiment conducted

The whole experiment was carried on a Dell PowerEdge T310 physical server machine with the following Configuration:

CPU:
Intel Xeon Processor with 8 Physical CPUs (0-7)
Model: Intel(R) Xeon(R) CPU X3440 @ 2.53GHz
Speed: 1197 MHz

Memory:
8GB Physical Memory, 1X8 GB, 1333MHz

NIC interface:
1Gbps Physical NIC port * 2

Paravirtualized Environment:
The paravirtualized environment was setup in the xen server with xen hypervisor 4.6 installed on Centos kernel, as a distribution bundled in Citrix Xenserver 7.0. The technical details of the experimental setup is given below

Operating System: Citrix Xenserver 7.0, with Xen hypervisor 4.4 stable version.

Dom0 Kernel: CentOS 6 64 Bit Kernel, used 4 VCPUs, 2GB RAM

DomU Guest VMS: Three VMs were created, with each VM being CentOS 6.4 64 bit Version, memory as 2 GB Ram, with 3 VCPUs each

Afer the experimental setup was set, the whole research methodology was carried out with the completion of following major tasks:

1. Experiment and observations on the impact of various kinds of workloads in various parameters in existing CPU scheduling algorithms.
2. Analysis of the outputs from the experiment and chosing the parameter for performance improvement.
3. Development of new CPU scheduling algorithm based on improvisation of chosen parameter.
4. Experimental Observation and verification of improvement in performance of the new proposed algorithm.

5.3 Experiment with existing Xen Schedulers

In this research, the last two schedulers, SEDF and Credit Scheduler were compared with the following mentioned workloads, in both work conserving and non-work conserving modes.

• **web server**: The web server throughput was measured. In this workload, fixed size (10 KB) files using httperf was used.

• **iperf**: Maximum achievable network throughput using iperf was measured with this standard benchmark tool.

• **disk read**: Finally, benchmark disk read throughput was measured with the `dd` utility for reading 1000 1-KB blocks.

The experiment was set to find out to mainly observer and find out the answer to the following queries:

- How sensitive are I/O intensive applications to the amount of CPU allocated to Dom_0?
- Does allocation of a higher CPU share to Dom_0 mean a better performance for I/O intensive applications?
- How does the weight parameter for the guest domains impact on various types of workloads for a guest domain?

- How significant is the impact of scheduler parameters on application performance?

5.4 Modified Credit Scheduler with Workload based Dynamic Weight

After the experiments and observations, it was found out, as is described in the following chapter, that the weight of the guest domains plays an important role in allocation of share of physical cpus to the virtual cpus. According to the nature of different types of workload running in the guest domains, the requirement of physical cpu share to the virtual cpus is increased. However, in the default credit scheduler, the weight is defined initially, and not changed during the run time dynamically. One can manually assign the weights to the guest domains by issuing xen command. However, it is very difficult and impractical for manually checking the workload of cpu of virtual machines and change their weight values.

To resolve this issue, I formulated a way to dynamically calculate the workload of each running virtual machines, and based upon the workload of each virtual machines, and based on this data, formulate a simple algorithm that updates the weight of virtual machines running in xen dynamically. This algorithm runs every T ms time interval, in my case I set it to 10ms, and updates the weight of each guest domains dynamically in that interval. Thus, the credit will be accumulated for the guest domain based on this weight.

The algorithm simply calculates the workload of each virtual machines, except the host domain, whose weight is simply kept constant throughout the test. The weight computed for each guest domain is then used by the existing default credit scheduler, for the scheduling. So this algorithm is an improvement in existing credit scheduler, in that it dynamically updates the weight of guest domains, whereas the original credit scheduler does not update the weights, even if the workload at each domain is different.

5.4.1 Current Xen Credit SchedulerAlgorithm

The current algorithm of the xen credit scheduler is given below:

Algorthm ***csched_schedule*** (now);
Input: now (the current time).
Output: ret (task to run next).
BEGIN {Check the VCPU that is about to end its time slice}
IF current running VCPU is runnable **THEN**

Insert it to local run queue;
ELSE
Current VCPU is idle or local run queue is empty;
END IF
{Select next runnable VCPU from local run queue}
Get the task from the top of local run queue;
IF the VCPU's priority is TURBO, OR no domain is set to be the turbo domain and the VCPU hasn't eaten through its credits
THEN
Remove the task from local run queue;
ELSE {See if there is more urgent task on other PCPU}
IF there is more urgent work on other PCPUs **THEN**
Get this more urgent work;
ELSE
Get the task from the top of local run queue;
END IF
END IF
{Update idlers mask if necessary}
IF the VCPU is IDLE **THEN**
IF idlers mask hasn't been updated **THEN**
Update idlers mask;
END IF
ELSE
IF the PCPU was idling **THEN**
Clear it from idlers mask;
END IF
END IF
{SET the time slice}
IF it's an IDLE VCPU **THEN**
Set illegal time slice (-1);
ELSE
Set the time slice to be CSCHED_MSECS_PER_TSLICE;
END IF
Set task to run next;
{Clear turbo domain setting if necessary}
IF TURBO DOMAIN is set and task to run next is the VBSP of TURBO DOMAIN
THEN CLEAR the TURBO DOMAIN setting;
END IF
{Set a turbo domain if necessary}
Get the next member from local run queue;
IF it is not the end of local run queue **THEN**
Get VCPU from the top of local run queue;
IF it's a non-idle VBSP and no domain has been set to be the TRUBO
THEN Set TRUBO DOMAIN;
Inform each PCPU that its run queue needs to be sorted;
END IF

END IF
RETURN task to run next;
END

Algorithm 1: Credit Scheduler Algorithm for xen Hypervisor

5.4.2 Modified Credit Scheduler Algorithm with Dynamic Weight Update

After the above algorithm finishes, it recomputes the credit of each VCPU, based upon the weight of its domain. Before the recomuputation, the weights of each domain will be modified by my currently proposed algorithm, so that the domain with higher workload will more credits than the domain with less workload.

A function called workload_calculate() is first of all created, that using the various data in xen's predefined structures, calculates the workload of all guest domains, and the total workload, which will be declared as global variables, so that the workload data will be availabe to the dynamically weight adjusting function that will be used for dynamic weight calculation.

In steps 8-9, the workload *WLi* of each user domain *domi* is calculated for the time interval *[curExcTime −T, curExecTime]* (i.e., the last *T* seconds), where *curExecTime* is the current time. In Xen's source code, a pointer variable, called *prv*, has been declared in common/sched credit.c (prv is declared as *struct csched private *prv*). The pointer *prv* is pointing to a structure (i.e., *struct csched private*) which holds the information about the states of the Credit scheduler (such as the list of active user domains, the total weight, *etc.*).

As it can be seen in step 8, I have used one of *csched private*'s member, called *active sdom*, to get the list of all active user domains. For each active user domain, one of its member is used, called *active vcpu*, to access each active VCPU of the user domain, as shown in step 8-c. The loop variables *sdom* and *svc* (declared in steps 6 and 7) are pointers of the *csched dom* and *csched vcpu* structures which are also declared in common/sched credit.c.

In step 8-c-i, a function vcpu runstate get() is called so that the current states of the active VCPU can be obtained. The states are stored in the structure *runstate* declared in step 5. The corresponding IDs of the active user domain *sdom* and the active VCPU *svc* are obtained in steps 8-a and 8-c-ii, respectively. In step 8-c-iii, the value of curExecTime(i,j) is set as the current

accumulated execution time of the active VCPU since it starts (it is obtained by the member *time* of the structure runstate, where RUNSTATE running indicates the accumulated execution time).

The difference between the current accumulated execution time curExecTime(i,j) and the last accumulated execution time lastExecTime(i,j) of the VCPU(i, j) is denoted as diffExecTime(i,j), and it is calculated in step 8-c-iv. The diffExecTime(i,j) can be treated as the workload of VCPU(i, j) in the monitoring window, i.e., the time interval *[curExcTime −T, curExecTime]*. In step 8-c-v, the workload WLi of the active user domain dom(i) for the last T seconds is calculated by summing its VCPU's workloads. In step 8-c-vi, the value of lastExecTime(i,j) is updated as the curExecTime(i,j) for the future iterations of the loop. Finally, the total workload WL(total) of all user domains is calculated in step 8-e, where WL(Total) = $\sum_{i=1}^{n} WL(i)$.

The algorithms for the workload computation function is given below.

ALGORITHM: WORKLOAD_CALCULATE

Global Declarations:

1. diffExecTime(i,j)=0, lastExecTime(i,j)=0 for each vcpu(i,j); //accumulated execution time for jth VCPU of ith Domain
2. Set WL(i) = 0; for each active domain domain(i)
3. Set WL(total) = 0;

Local Declarations:

4. declare curExecTime(i,j) for each vcpu(i,j);
5. declare *struct vcpu_runstate_info runstate;*
6. declare *struct csched_dom *sdom;*
7. declare *struct csched_vcpu *svc;*

Start:

8. for each sdom in prv->active_sdom, do
    a. i <-- (sdom->dom->domain_id);
    b. WL(i)=0;

c. for each svc in sdom->active_vcpu, do

   i. vcpu_runstate_get(svc->vcpu, &runstate);

   ii. j <-- (svc->vcpu->vcpu_id);

   iii. curExecTime(i,j) <-- runstate.time[RUNSTATE_running];

   iv. diffExecTime(i,j) <-- CurExecTime(i,j) - lastExecTime(i,j);

   v. WL(i) <-- WL(i) + diffExecTime(i,j);

   vi. lastExecTime(i,j) <-- curExecTime(i,j);

d. end for

e. WL(total) = WL(total) + WL(i);

9. end for

End

Algorithm 2: Workload Calculation Function Algorithm for Active Domains

The function is performed for every *T* seconds and its time complexity is *O*(*nk*), where *n* and *k* are the number of user domains and the maximum number of VCPUs in the guest domains.

The modified credit scheduler with dynamic weight update is an enhanced version of the Credit scheduler for which a user domain with heavy workload can obtain more CPU share at the run-time. In order to allocate more CPU share to a heavy-workload user domain, the value of its weight will be increased. The weight of a user domain will also be decreased if its workload is lighter. However, a lighter-workload user domain might crash if the value of its weight is too small. To prevent a user domain crash, the weight of every user domain cannot lower than a specific value, called *minimum weight*, denoted by *Wmin*. In my implementation, I have set the value of *Wmin* as 8.

The modified credit scheduler with dynamic weight update will be performed for every *T* seconds to calculate the weight *W(i)* of each user domain *dom(i)* according to its workload *WL(i)* obtained by the Workload Calculation Function (WCF). The *WL(i)* (for each domain *dom(i)*, where $1 \leq i \leq n$) and *WL(total)* are declared as global variables in Algorithm 2 so that they can be used directly in the modified credit scheduler with dynamic weight update. The details of my proposed modified credit scheduler with dynamic weight update is given in Algorithm 3.

As shown in Algorithm 3, the modified credit scheduler with dynamic weight update first calls the WCF to get the workloads of user domains (i.e., *WL(i)* for each user domain *dom(i)* and *WL(total)*) as shown in step 2. The loop in steps 3-4 is used to calculate the weight of each active user domain *dom(i)*. It is to be noted that the way list of active domains and their domain IDs are obtained are the same as that of the WCF as shown in steps 3 and 3-a. In step 3-b, calculation of the weight *W(i)* for each active user domain *dom(i)* is done, where *W(i)* is calculated as the ratio of *WL(i)* to *WL(total)*.

It should be again noted that the calculated weight is normalized to an integer between 0 and 100 because the value of a weight must be integer. In order to avoid a user domain crash, in step 8-c, the value of *W(i)* is guaranteed to be larger than or equal to the minimum weight, i.e., *W(min)*.

The time complexity of the modified credit scheduler with dynamic weight update is $O(n)$, where n is the number of user domains in the system. After the modified credit scheduler with dynamic weight update calculates the workloads of user domains, it uses the original Credit scheduler to allocate proportional CPU share to each user domain. (see Section 4.3.3 for the original Credit scheduler). Since the workload of each user domain is updated dynamically for every T seconds, the weight value is adjusted according to it so that the CPU share of a user domain varies with its dynamic workload. As the result, the performance of each user domain can be improved.

ALGORITHM: DYNAMIC_WEIGHT_UPDATE

Local Declaration:

1. declare struct csched_dom *sdom;

Start:

2. workload_calculate(); //get workload of all domains
3. for each sdom in prv->active_sdom, do
    a. i <-- (sdom->dom->domain_id);
    b. W(i) <-- (integer(WL(i)/WL(total)))*100;
    c. if (W(i) < W(min)) then
        i. W(i) <-- W(min);

d. end if

4. end for

End

Algorithm 3: Modified Credit Scheduler with Dynamic Weight Update

5.5 Experimental observation and verification of the modified algorithm

The tests for the new algorithm was conducted for checking its performance with comparison with the original algorithm. For testing the new algorithm, rather than repeating the previously conducted experiments with the new algorithm, I tested it with the three guest domains with different amount of workloads in each and by checking the allocated PCPU share for the VCPUs of the guest domains. The test was conducted this way because, the new algorithm aims to allocate more PCPU share to the guest domains with higher amount of workload.

The experiment was set with the three user domain *dom*1, *dom*2 and *dom3,* all running simultaneously. Initially, the weight and the cap of *dom1*, *dom2* and *dom3* are set as 256 and 0, i.e., $W1 = W2 = W3 = 256$ and $Cap1 = Cap2 = Cap3 = 0$. Different amount of workloads were generated in each domain for conducting the experiment. **SysBench** benchmarking tool was used to create different processes (i.e., jobs) to the domains continually so that the expected workload can be generated. The experiment was performed to 200 seconds with sample taken at each 10 seconds and its actual allocated PCPU shares were observed. The experiment was conducted for both original credit scheduler algorithm and modified credit scheduler with dynamic weight update.