2016 IEEE 9th International Conference on Cloud Computing

Towards Application-centric Fairness in Multi-Tenant Clouds with Adaptive CPU Sharing Model

Anthony O. Ayodele, Jia Rao, Terrance E. Boult,
Department of Computer Science
University of Colorado, Colorado Springs, USA
{aayodele, jrao, tboult} @uccs.edu

Abstract - **The performance of cloud application is often quite disappointing due to unmanaged consolidation. Therefore, efforts are required to reduce co-tenants interference and provide predictable application performance in multi-tenant cloud environments. In this paper, we examined the complex interplay among cloud tenants as they compete for CPU time, and shared hardware resources. We propose Adaptive CPU Sharing (ACS) approach that reduces co-tenants interference and provides predictable application performance. Our approach is to monitor the progress of submitted applications at runtime, tracks the slowdown of individual application and applies adjustment until convergence. Thus, when an application suffered more slowdown, we allocate more CPU to reduce unfairness. In establishing system support for fine-grained profiling, we report system level activities at sub-second granularity. We predicted application performance degradation by creating a mathematical relationship between high-level application performance and low-level machine events (i.e., CPU steal time and L2 caches miss rate). We validate the added value of our approach by comparing application performance slowdowns (average) with various datasets. Based on our experimental results, our approach helps mitigate co-tenant interference and reduces unfairness by minimizing the overall application slowdowns.**

Index Terms – Multi-tenants Cloud Computing, Co-tenant Interference, Performance Measurement, Performance Variation, CPU Steal Time, Performance Degradation

I. INTRODUCTION

This paper is an extension of our previous work [1] with the main contribution that involve the implementation of mitigation techniques to reduce the impact of co-tenants' interference and improve application performance. Application performance degradation due to resource sharing and contention among co-tenants is well study in literature. However, most of the existing work focuses on a particular aspect of the resource sharing in isolation. There lacks a comprehensive understanding of the complex interplay between individual hardware components under resource contention. In this research work, we focused on predicting application performance by establishing a mathematical relationship between the high-level application performance and the low-level CPU multiplexing from resource sharing perspective. We measure application performance in term its execution runtime "mean" (average) in an interference free environment as a baseline. We define application performance degradation as unexpected slowdowns incurred by an application due to contention and co-tenant interference. Thus, we define multi-tenant cloud environments to be *fair* if all running applications experience equal slowdowns. This assumption is based on application performance in term of its execution runtime rather than on resource related metrics. Prior work [2, 3,4] supports this assumption. Therefore, we can say *unfairness* occurs in multi-tenant cloud environments when application of equal weight experience disparity slowdowns. The over-arching goal of this work is to reduce overall application performance degradation by considering CPU allocation and contentions on shared resources. We propose a novel approach that helps mitigate co-tenant interference, reduce unfairness in the allocation of shared resource and enhance application performance in a multi-tenant cloud environment. To this end, our overall approach is to:

- Quantify the impacts of CPU time multiplexing and hardware resource sharing on application performance and applies adaptive resource control (i.e., CPU allocation) to achieve equitable services among cloud users.
- Measure application performance in term of unfairness by using the relative execution runtime slowdown compared to the runtime in an interference-free environment.
- Design an Adaptive CPU sharing approach that helps mitigate co-tenant interference, and reduce unfairness in the allocation of shared resource in multi-tenant cloud environments. The goal is to minimize the "average" slowdown of co-located applications runtime.

The basic approach is to monitor the progress of submitted applications at runtime, tracks the slowdown of individual application and applies adjustments until convergence.

First, we define multi-tenant cloud environments to be *fair* if all running applications experienced equal slowdowns. We denote by;

- *(i), application*
- *(t_alone_i),* as runtime when *(i)* runs alone
- *(t_share_i),* as runtime when *(i)* runs concurrently with other application in multi-tenant environments

Thus, the slowdown of an *application (i)* is calculated as:

$$\text{slowdown\_i} = \frac{\text{t\_share\_i}}{\text{t\_alone\_i}} \tag{1}$$

We measure *unfairness* to *application (i)* among other application (*n*) as the ratio between its peak (maximum) and lowest (minimum) slowdown.

$$\text{Unfairness} = \frac{MAX(slowdown\_0 ,slowdown\_1,\dots slowdown\_n\,)}{MIN(slowdown\_0,slowdown\_1,\dots slowdown\_n)} \tag{2}$$

The ultimate goal of our approach is to minimize the unfairness and ensure that applications with different runtime experience consistent slowdowns.

II. BACKGROUND AND MOTIVATION

In a multi-tenant cloud environment; there are significant contributing factors that impact overall application performance and throughput. Thus, we discuss below three prevailing challenges facing the Multi-tenants' clouds environment that continues to impact negatively users experience and adoption of the cloud.

Attempt to understand these challenges and the need to address them motivates the basis for our research work.

A. Multi-tenant CPU Sharing

As a standard practice by cloud providers, a multi-tenant public cloud platform such as Amazon Elastic Compute Cloud EC2 [5] instance (virtual machine [VM]) share resources with other instances on a single host in a virtualized environment. Predominantly shared resource is the CPU cycles; however, unmanaged sharing of CPU usage has significant consequences on tenants' performance and throughput. The sharing of CPU time causes undue CPU steal time [6].

B. Contention for Shared Hardware Resource

Mekkat et al. [7] mentioned that one of the key challenges in designing heterogeneous multicore systems is the sharing of on-chip resources such as the last-level cache (LLC), which may have significant impact on system and application performance. Zhuravlev et al. [8] stated that the challenges of shared resource contention existed because chip multicore processor cores are not independent processors but rather share common resources among cores such as the last level cache (LLC).

C. Lack of Fairness in Resource Management

Cloud application performance largely depends on proper utilization of multiple reconfigurable shared resources such as the CPU, memory, and disk I/O bandwidth. It is imperative to ensure that any reconfiguration, allocation, and utilization of shared resources are fair to all tenants in such a way that mitigate undue interference and reduce performance degradation.

III. RELATED WORK

In this section, we review existing research work and discuss how this research work complements existing work. First, in our prior research work [1], we profiled and analyzed the impact of co-tenants interference and established the root causes of application performance variation in multi-tenant cloud environment. Xu et al. [4] in their research propose a fair-progress process scheduling (FPS) policy to improve system fairness. Their strategy is to force equally weighted applications to have the same amount of slowdown when they run concurrently by allocating more CPU time. In this paper, we leverage similar approach by allocating more CPU based on the high-level application performance and low-level contention for shared resources. Menon et al. [9] in their work presented Xenoprof, a system-wide statistical profiling toolkit implemented for the Xen virtual machine environment with focused on performance overheads for network I/O device. Blagodurov et al. [10] investigated thread scheduling can help mitigate contention for shared resource. Nathuji et al. [11] presented Q-Clouds, a QoS-aware control framework that tunes resource allocations to mitigate performance interference effects by using online feedback to build a multi-input multi-output (MIMO) model that captures performance interference interactions and uses it to perform closed loop resource management. Also, the concept of proportional share–based algorithm was proposed [12, 13, 14], which allocates CPU resources based on resource specifications in other to help mitigate interference among co-tenants and reduce performance degradation.

IV. DESIGN

In this section, we provide the design of our fine-grained measurement framework, experimental platform setup, experiment test cases, and implementation approach. Our implementation approach includes configuration of testbed platform, software installation, and codes design. The support system design includes in-hypervisor reporting of CPU allocation and cache misses. First, a multi-tenant private cloud platform is configured with a quad-core processor (Xen environment) with two guests VM; (VM-INTR) and VM-SPEC). (VM-SPEC), and (VM-INTR) are assigned CPU weight with the ratio of 1:1. The guest OS and Xen are both instrumented to report *CPU steal time* and hardware performance statistics of a VM in real time. We design two system level fine-grained profilers; *CPU steal time profiler*, and *Hardware performance counters profiler* to record and measure the interplay as tenants compete for CPU time and shared hardware resources. Measurements from the two profiler help derive meaningful models to predict the performance of various applications in the cloud. Finally, a system level Adaptive CPU Sharing (ACS) algorithm was design and implemented. ACS monitors the runtime statistics of co-running applications and predicts the overall slowdown of each application. If unfairness exists, ACS throttles the application that has lower slowdown than the other by setting a lower CPU cap. Table 1 and Figure 1 show our experimental system setup and reference architecture.

TABLE 1. OVERVIEW OF SYSTEM SETUP

Component name	**Xen 4.0.2 (domain- 0)**	**VM-SPEC (tenant #1)**	**VM-INTR (tenant #2)**
System information	CentOS 5.3 Linux, Intel(R) Core (TM) Quad CPU Q9550 2.83GHZ	CentOS 5.3 Linux, Intel(R) Core (TM) Quad CPU Q9550 2.83GHZ	CentOS 5.3 Linux, Intel(R) Core (TM) Quad CPU Q9550 2.83GHZ
Memory	3.48GB	2.8GB	2.8GB
Hard disk size	103GB	50GB	50GB
CPU Core	4	4 vCPU	4 vCPU

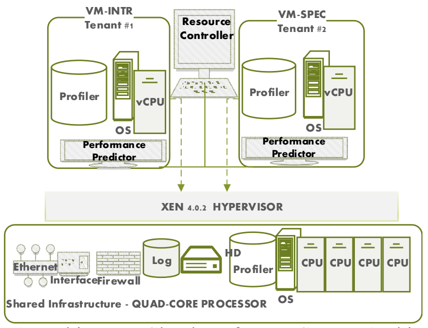

Figure 1. Multi-tenant Clouds Reference System Architecture

A. Experiment Test Cases

We conducted our research with three (3) experimental test cases:

- Test Case 1(Dataset 1): experiment in an interference free environment with standard resource management.
- Test Case 2 (Dataset 2): experiment in a co-located interference prone environment with standard resource management.
- Test Case 3 (Dataset 3): experiment in a co-located interference prone environment with controlled resource management.

Our experiments benchmark SPEC CPU2006 simulate certain aspects of real-world application workloads and comprises of both CPU and memory bound benchmarks. Table 2 shows the Datasets for Test Case 1, Test Case 2, and Test Case 3 respectively.

TABLE 2: EXPERIMENTAL TEST CASES 1-3

TEST CASE 1		TEST CASE 2		TEST CASE 3	
VM-SPEC	VM-INTR (w/milc)	VM-SPEC	VM-INTR (w/milc)	VM-SPEC	VM-INTR (w/milc)
Mcf	No Load	Mcf	Milc	Mcf	Milc
Milc	No Load	Milc	Milc	Milc	Milc
Gobmk	No Load	Gobmk	Milc	Gobmk	Milc
Bzip	No Load	Bzip	Milc	Bzip	Milc
Soplex	No Load	Soplex	Milc	Soplex	Milc
Libquantum	No Load	Libquantum	Milc	Libquantum	Milc

B. Fine-grained CPU Steal Time Profiler

To measure the CPU steal time, the profiler makes periodic system call via the Xen hypervisor to get the system runstate with HYPERVISOR_vcpu_op (VCPUOP_get_runstate_info) operation. VCPUOP_get_runstate_info triggers an hypercall [15] which allows the guest OS to perform privileged operation through Xen hypervisor, similar to the use of system calls in a conventional operating system. The VCPUOP_get_runstate_info was implemented on each guest OS to report the vCPU steal time for given period. The profiler takes two arguments; *profiling duration* (length of the sampling period) and the *vCPU identification*. We define *vCPU runnable time* as the time the vCPU is willing to run but do not have the chance to run because other VMs are using its time. Also, we define *vCPU offline time* as the capped time for the vCPU i.e. there is a static cap on the CPU allocation. We denote by;

- (o), *vCPU offline time*
- (r), *vCPU runnable time*
- (S_x), *CPU steal time*

Thus, *CPU steal time (S_x)* for a given vCPU (y) is calculated as;

$$S_x(y) = r_y + o_y \qquad (3)$$

Algorithm 1 below supports the CPU steal profiler and helps in the measurement of the CPU steal time. The profiler achieves fine grained profiling at the milliseconds level.

Algorithm 1: Algorithm for fine-grained CPU steal time profiler

```
Input: vCPU id, sampling length
Domain information ⟵ (v->domain = d ;)
Virtual CPU (vCPU) ⟵ (v->vcpu_id = vcpu_id;)
Domain state ⟵ (v->vcpu_info = ((vcpu_id < XEN_LEGACY_MAX_VCPUS)
steal ⟵param.total_stolen_time;
blocked ⟵param.total_blocked_time;
running ⟵ param.total_running_time;
offline ⟵ param.total_offline_time;
Foreach domain do
check current domain status, vCPU
  if
    vCPU allocation equal Null
    return NULL; domain = idle
    then (runstate.state = RUNSTATE_offline; blocked)
  end
  if
    vCPU allocation not Null
    return ACTIVE; domain= active
    then (runstate.state = RUNSTATE_running; running)
  end
printf("steal time: %llu running time: %llu blocked time: %llu \n", steal, running, blocked);
End
```

C. Fine-grained Hardware Performance Counters Profiler

Hardware performance counters statistics are reliable metrics for program characterization, system testing, and performance evaluation [16]. On modern CPU architectures, multiple cores share last-level cache (LLC), where LLC is the last cache available; beyond this cache, the access must go to memory. Therefore, cache activities are important metric to understand the memory usage of a VM in multi-tenant environment. To profile contention for shared hardware resource, we patched the Xen hypervisor with Perfctr-Xen [17] to allow measurement of low-level hardware performance counters Perfctr [18]. Our experimental system set-up is a Type-1 virtual machines [20] environment, whereby, the Xen hypervisor serves as the lowest layer with direct access to the supporting hardware infrastructure and the guest VMs run on top of the Xen hypervisor. The profiler captures L2 cache activities and help to measure the interplay between application performance and processor events. It's important to note that the Xen Quad-core processors used in this research work has two Level 2 (L2) caches, each shared by two processor cores [21, 22]. Also, the Xen Quad core support two hardware counters, and can report different events. The profiler was configured to count and report two predefined hardware performance statistics; *L2_rqsts.self.demand.mesi* [23], and *L2_rqsts.self.demand.i_state* [23]. Each core has a L2 cache of same size; the L2 cache miss rate is independent of the thread-to-core assignment [24]. Exiting research supports the use of metrics from last level cache miss rate as an effective data for quantifying shared hardware resource contention [9]. To establish the L2 cache miss_rate, we counted *L2_rqsts.self.demand.i_state* (all completed L2 cache demand requests from the core that miss the L2 cache) and *L2_rqsts.self.demand.mesi* (all completed L2 cache demand requests from the core) per thousand instructions. We denote by;

- *(W), L2 cache miss ratio (cache miss_rate)*
- *(R), L2 cache demand request missed*
- *(R'), L2 cache demand request*

Thus, we calculate the *L2 cache miss_ratio* as;

$$W = \frac{R}{R'} \qquad (4)$$

Algorithm for the fine-grained hardware performance counter helps in the measurement of *L2 cache miss_rate*. Algorithm is suppress for this paper.

D. Design of Adaptive CPU sharing model

As a background, a CPU scheduler determines which VM should run on the CPU at any given time. In general, majority of CPU schedulers are broadly categorized as either a proportional share (PS) or fair-share (FS) [24]. FS schedulers provide proportional-share among multi-tenant clients by adjusting the priorities of clients in the most suitable way. Experimental measurements had showed that most FS schedulers provide a reasonable, proportional fairness over relatively large time intervals [25]. On the other hand, PS techniques focus on allocating CPU in proportion to the VM shares (weights). The difference between FS and PS can be discuss in term of the time granularity at which both schedulers operate. PS aim to provide an instantaneous form of CPU sharing among the active VM clients according to their weights. In contrast, FS provides a time-averaged form of proportional sharing based on the actual VM use of CPU measured over long time periods [24]. With the traditional Xen credit scheduler [26], each tenant (VM) is assigned a weight and a cap. If the CPU cap is Zero (0), then the guest can receive any extra CPU time. A non-zero cap CPU allocation normally expressed as a percentage limits the amount of CPU a tenant can receive. Therefore, as a standard, the Xen credit scheduler uses 30 ms time slices for CPU allocation and a guest (vCPU) receives 30 ms before being preempted to run another guest [27]. After every 30 ms, the priorities (credits) of all runnable guests are recalculated. However, with the dynamics state of the cloud environments, the priorities credit rule of Xen scheduler may lead to fluctuation in the allocation of CPU and thereby making CPU allocation inequitable to all users. CPU time-sharing among multiple VMs has been showed to provide much more predictable performance than I/O sharing [28]. Our approach considered both the PS and FS schedulers techniques in its design. Our design approach extends the Xen credit scheduler [29] to support more efficient, fair, and dynamic scaling of CPU sharing among co-tenants. Specifically, the ACS algorithm checks the status of individual applications based on runtime characteristics of the workloads to enforce fairness in the allocation of CPU. The ACS algorithm encrust on the (Xen)'s hypervisor by building on the *xm shed-credit -c* [30] to control the CPU resource allocation to each guest so that each guest receives relatively same level of slowdown. Thus, in designing the ACS algorithm, we modified the Xen management utility through the Linux kernel to allow for adaptive changing of the CPU allocation so that each VM receive relatively same level of slowdown based on the high-level application characteristics rather than arbitrary capping the CPU value for each tenants. Therefore, our algorithms adaptively change the CPU cap values of each VM to achieve fair slowdowns to all tenants. We design the ACS to have a controlled resource sharing rules i.e. the controlled factor (Y) of the application behavior, and the application performance effect (K_n). Therefore, (Y) and (K_n) together determine the magnitude of the resource sharing adjustment. The change of (Y) has impact on the application overall performance. One of the most commonly used statistical procedures to model relationships between variables is regression analysis [32]. Regression analysis relates a dependent variable (Y) with explanatory variables $(K_1, K_2, K_3, K_4, \dots K_n)$, used as predictors. A simple form of regression analysis is linear regression; in which we assume the dependent variable is a linear function of explanatory variables. Thus, we define (Y) and $(\bar{Y})$ as;

$$Y = a_0 + a_1 \cdot K_1 + a_2 \cdot K_2 + a_3 + \cdots + a_n . K_n \quad (5)$$

$$\bar{Y} = a_0 + a_1 \cdot K_1 + a_2 \cdot K_2 + a_3 + \cdots + a_n . K_n \quad (6)$$

The goal of linear regression analysis is to find coefficients $(a_0, a_1, a_2, \dots, a_n)$, to minimize error $|Y - \bar{Y}|$. Where (a) is a discount factor that give more weight on the recent observation of (Y) and $(\bar{Y})$, with due consideration on the application workloads past behavior. We have assumed the application workloads past behaviors are predetermined by the application benchmark workloads runtime in an interference free environment. Also, if a repeated application behavior does not deviate from the establish application workloads runtime, the application performance (K_n) would converge. If the application behavior deviates from reference performance interference free baseline value, the value (a) should quickly recalibrates. To reduce unfairness, it's imperative that we have a capability that allows for dynamic adjustment to the scheduler, and we need to understand the application characteristics. ACS uses the controller to manage the CPU cap for each VM so that unfairness is minimized. Assume that each VM has *slowdown_i* and the average slowdown is *slowdown_avg*. Thus, the *cap of VM* (i) at time (t) is calculated as:

$$cap(t)\_i = cap(t-1)\_i - K * (slowdown\_avg - slowdown\_i) \quad (7)$$

Where (K) is the integral gain parameter that controls the rate of cap adjustment

$$K = \frac{|slowdown\_i \ - slowdown\_avg|}{slowdown\_avg} \quad (8)$$

As such, if a VM receive lower slowdown than the average, it gains advantage over others and will have its cap value decreased. In contrast, a VM having higher slowdown than the average will see its cap increased. Figure 1 depicts the ACS process flow. We describe a typical ACS process flow process as follows:

a. Cloud users submit workloads;
b. Shared resources are made available to process submitted workloads;
c. Fine-grained profilers measure tenant interplay and competition for shared resource;
d. CPU allocation to tenants is determine by the workload behavior and tenant state;
e. Workloads performance is manage with reasonable level of prediction.

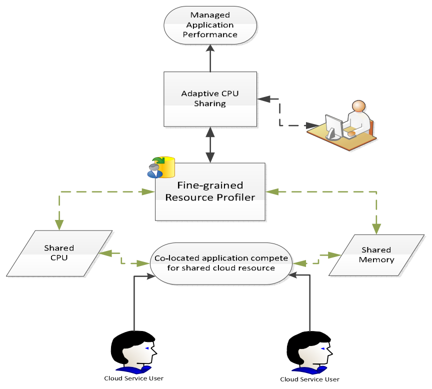
Managed Application Performance
Adaptive CPU Sharing
Fine-grained Resource Profiler
Shared CPU
Co-located application compete for shared cloud resource
Shared Memory
Cloud Service User
Cloud Service User

Figure 2: Schematic Overview of the Steps involved towards Application-centric Fairness in Multi-Tenant Clouds

Algorithm 2 below supports the Adaptive CPU sharing and helps to enforce fair CPU sharing among co-tenants.

Algorithm 2: Algorithm for Adaptive CPU sharing model

```
Input: VM id, VM Weight, CPU credit; X
Domain information ⟵ (v->domain = d;)
VM state ⟵ running(r), blocked (b), paused (p), shutdown(s)
Foreach VM id do
  check current VM state, vm;
  echo VM state ⟵ CPU_list; VM-INTR, VM-SPEC
  echo VM weight allocation ⟵ sched-credit; VM-INTR, VM-SPEC
  echo " is the application slowing down?"
  select yn in "Yes" "No"; do
  case $yn in
   [Yes] adjust CPU Percent; break; [No] exit;;
 /* Ensure VM fair credit weight */
  assign VM weight VM-INTR ⟵ sched-credit; =$(xm sched-credit -d 1 -w 256); VM-INTR
  assign VM weight VM-SPEC ⟵ sched-credit;=$(xm sched-credit -d 2 -w 256); VM-SPEC
  echo CPU allocation ⟵ sched-credit; CPU_allocation_ VM-INTR, CPU_allocation_ VM-SPEC
 if [ "$CPU_list" == "VM-INTR"] && [ "$state" == b]; then
  /* allocate maximum CPU percent to running VM*/
    CPU_allocation_VM-SPEC ⟵$(xm sched-credit -d 2 -c X)
    echo "fairCPU_VM-SPEC:"$CPU_allocation_VM-SPEC
    exit(1);
  fi
 if [ "$CPU_list" == "VM-SPEC"] && [ "$state" == b]; then
   /*Allocate maximum CPU percent to running VM*/
    CPU_allocation_VM-INTR ⟵$(xm sched-credit -d 1 -c X)
    Echo "fairCPU_VM-INTR:"$CPU_allocation_VM-INTR
    exit (1);
fi
  echo Adaptive CPU allocation achieved by all client VM⟵
"CPU_Credit:"$CPU_Credit_reallocation=$(xm sched-credit)
End
```

V. EXPERIMENTAL RESULTS

In this section, we are interested in analyzing the relationship between the low-level machine activities and high-level application performance. We conducted series of experiments, and as mentioned in section IV, we have three test cases. We started our experiments with the standard Xen resource allocation to establish application performance baseline based on the workloads execution runtime (average). Thus, we analyze results from our experiments datasets. CPU Steal Time and Application Performance

Table 5, 6, and 7 shows the benchmark workloads runtime and fine-grained CPU Steal Time (Milliseconds) metrics from experimental Test Cases 1-3.

TABLE 5: TEST CASE 1 - BENCHMARK WORKLOADS RUNTIME (AVERAGE) AND CPU STEAL METRICS

Benchmarks	workload slowdown (average) runtime (VM-SPEC)	Ave. Steal time (VM-SPEC)	Ave. Steal time (VM-INTR)
Mcf	8.61	2	0
Milc	11.8	2	0
Gobmk	10.46666667	2	0
Bzip2	10.23333333	2	0
Soplex	7.766666667	2	0
Libquantum	16.08333333	2	0

TABLE 6: TEST CASE 2 - BENCHMARK WORKLOADS RUNTIME SLOWDOWN (AVERAGE) AND CPU STEAL METRICS

Benchmarks	workload slowdown (average) runtime (VM-SPEC)	Ave. Steal time (VM-SPEC)	workload slowdown (average) runtime (w/milc) (VM-INTR)	Ave.Steal time (VM-INTR)
Mcf	1.116918312	34	2.159604517	4980
Milc	1.213276836	40	2.357344636	5011
Gobmk	1.025477706	40	2.000000000	5010
Bzip2	1.01791531	58	1.991525424	5040
Soplex	1.62446352	40	1.411016949	3920
Libquantum	1.187564767	34	2.419491525	4920

TABLE 7: TEST CASE 3 - BENCHMARK WORKLOADS RUNTIME SLOWDOWN (AVERAGE) AND CPU STEAL METRICS

Benchmarks	workload slowdown (average) runtime (VM-SPEC)	Ave.Steal time (VM-SPEC)	workload slowdown (average) runtime (w/milc) (VM-INTR)	Ave.Steal time (VM-INTR)
Mcf	1.873790166	64	1.629943503	180
Milc	1.508474576	253	1.45480226	339
Gobmk	1.162420382	177	1.059322034	243
Bzip2	1.858306195	18	1.706214689	77
Soplex	1.46137339	272	1.209039548	283
Libquantum	2.18016529	192	1.378531074	264

To characterize the relationship between the *benchmark workload slowdown (t)* and the *CPU steal time(x)*, we calculate the *linear correlation coefficient (r)* between the *benchmark workload slowdown (t)* and *CPU steal time (x).* We denote by;

- (x); *CPU steal time*
- (t); benchmark workloads slowdown

$$r = \frac{n\sum xt-(\sum x)\ (\sum t)}{\sqrt{n(\sum t^2)-(\sum t)^2}\sqrt{n(\sum x^2)-(\sum x)^2}} \tag{9}$$

Using (9), and metrics from Table 6, Test Case 2, we, establish that the *benchmark slowdown (t)* is proportional to the *CPU steal time (x)* with ***r = 0.8361***. Linear correlation coefficient *(r)* is closer to 1, this indicate a strong positive relationship between (t) and (x). This finding validates the relationship between CPU steal and its impact on benchmark workloads runtime in term of application performance.

Figure 3 shows the fine-grained CPU steal time for Test Case 1, Test Case 2, and Test Case 3.

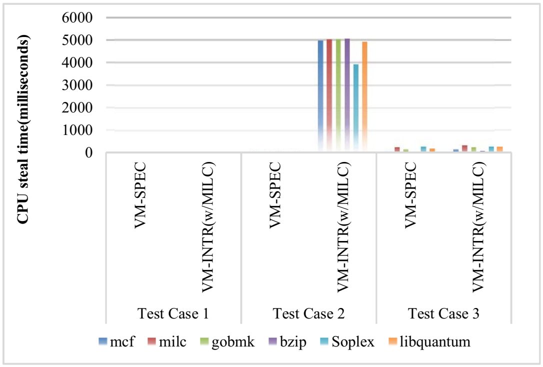

Figure 3: SPEC CPU2006 benchmark workloads showing CPU Steal Time in different consolidation scenarios.

In figure 3, Test Case 1 depicts zero CPU steal time due to lack of co-tenant interference. In Test Case 2, Tenant #2 experienced unfair high CPU steal due to co-tenant interference. In Test Case 3, Tenant #1 and #2 experience shared CPU steal time. This finding underscore the underlying indicator for unfairness in application slowdowns; thereby, leading to unpredictable application performance in multi-tenant cloud environment.

A. Memory Access and Benchmark Workload Performance

To further support predictable application performance in multi-tenant cloud, we analyze the relationship between high-level application performance and low-level memory access. Studies have shown that effective memory access among co-tenants is very important in multi-tenant clouds. Therefore; the memory miss rate become more significant.

Table 8, and Figure 4 shows the consolidated *L2 cache miss_rate* metrics from *L2 cache demand request miss*, and *L2 cache demand request* events captured during our experiments for each dataset captured as we submit the benchmark workloads for processing.

TABLE 8: L2 CACHE MISS RATE METRICS FROM TEST CASES 1-3

Benchmarks	Test Case 1	Test Case 2	Test Case 3
Mcf	0.05747975	0.1169545	0.079881
Milc	0.17263025	0.19241325	0.14037925
Gobmk	0.028304	0.0831910	0.0973515
Bzip	0.025819	0.03417075	0.04921675
Soplex	0.08920625	0.10622325	0.1461455
Libquantum	0.05807225	0.0733105	0.075655

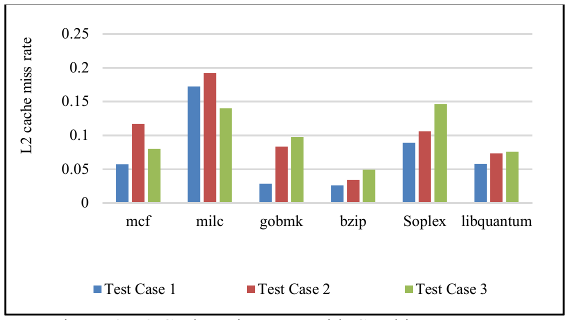

Figure 4: L2 Cache Miss_Rate with Combine Test cases

Further, in Figure 5, 6, and 7, we show the outcome of low level memory access events (L2 cache miss rate) for the quad-core processor with various experiments test cases.

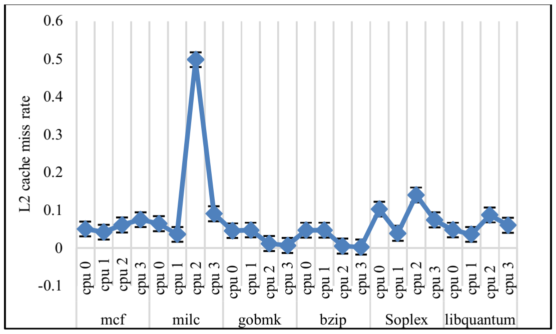

Figure 5: Test Case 1 - Quad-Core Processor Activities

Figure 5, shows moderate spike in L2 cache miss rate due to lack of interference and co-tenant competition.

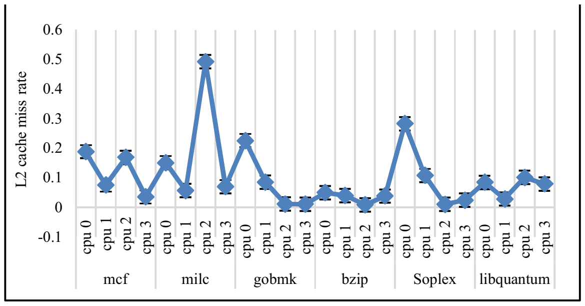

Figure 6: Test Case 2 - Quad-Core Processor Activities

Figure 6, shows significant spike in the L2 cache miss rate because of high level of contention for memory access between the two tenants.

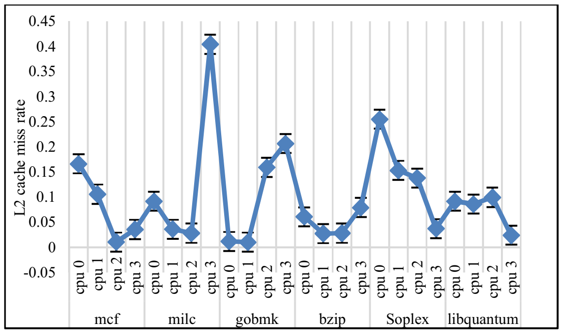

Figure 7: Test Case 3 - Quad-Core Processor Activities

Figure 7, shows significant spike in the L2 cache miss rate because of high level of contention for memory access between the two tenants. These findings validate the relationship between effective memory access and application performance. Therefore, we can conclude that memory as a shared resource contributes to application performance variation in multi-tenant cloud environment. Thus, memory tuning is a viable approach that can be use to promote application-centric fairness in multi-tenant clouds.

VI. EVALUATION OF EXPERIMENTAL RESULTS

In this section, we analyze and evaluate our experimental results and findings. Next, to prove this research hypothesis, we are interested in further analysis of results and metrics from our experiments. Therefore, to further test the hypothesis of our approach, we reviewed several performance prediction techniques; Weighted Means [31, Linear Models (LM) [32], and Support Vector Machines [33]. Using the weighted means technique, we focus our measurement on the application performance “mean” average. Therefore, we define the application execution runtime (average) as the primary measure of the application performance of individual application. The average “mean” method help to measure application performance scores relatively across Test Cases 1-3. Therefore, to quantitatively express the application performance runtime slowdown “mean” average, we analyze results from our experiments Test Cases.

We denote by;

- *(P),* overall application performance
- (t), the application *workloads runtime*
- *(n),* instances of each submitted application workloads

If we have the application workloads runtime containing the value; $t_1, t_2, t_3, t_4, \ldots, t_n$, the arithmetic mean P is define as;

$$P = \left(\frac{\sum_{i=1}^{n} t_{1.}}{n}\right) \tag{11}$$

Thus, we calculate the overall application performance(P_n), “mean” (average) as;

$$P_n = \left(\frac{\sum_{i=1}^{n}(t_1+t_2+t_3+t_4+\ldots t_n)}{n}\right) \tag{12}$$

To evaluate the performance of our algorithm, we analyze results from the various Test Cases submitted. We measure benchmark workload slowdown as a disunion between the application runtime with interference *(t_share_i)* and application runtime without interference *(t_alone_i)*.

First, we use experimental results from Test Case 1 (Table 9 and Figure 8) and Test Case 2 (Table 10 and Figure 9) respectively. For instance, In Table 9, the *milc* benchmark workload (average) normalized runtime baseline in an interference free environment is **11.8minutes** (VM-SPEC). Figure 8 shows *milc* benchmark workload baseline runtime.

TABLE 9: TEST CASE 1 - BENCHMARK APPLICATION WORKLOADS RUNTIME SLOWDOWN (AVERAGE) METRICS

Benchmarks	**Ave. workload runtime (VM-SPEC) interference free**
Mcf	8.61
Milc	11.8
Gobmk	10.46666667
Bzip2	10.23333333
Soplex	7.766666667
Libquantum	16.08333333

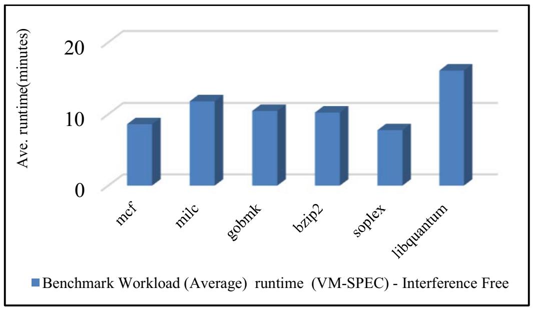

Figure 8: Test Case 1 - Benchmark Workloads Runtime (Average) in an Interference Free Environment with Standard Resource Management

In Table 10, the *milc* benchmark workload runtime slowdown (average) in a co-located environment with standard resource sharing is calculated as **1.21minutes** (VM-SPEC) and **2.35 minutes** (VM-INTR).

TABLE 10: TEST CASE 2 - BENCHMARK APPLICATION WORKLOADS RUNTIME SLOWDOWN (AVERAGE) WITH STANDARD RESOURCE

Benchmarks	**workload slowdown (average) runtime (VM-SPEC)**	**workload slowdown (average) runtime (w/milc) (VM-INTR)**
Mcf	1.116918312	2.159604517
Milc	1.213276836	2.357344636
Gobmk	1.025477706	2.000000000
Bzip2	1.01791531	1.991525424
Soplex	1.62446352	1.411016949
Libquantum	1.187564767	2.419491525

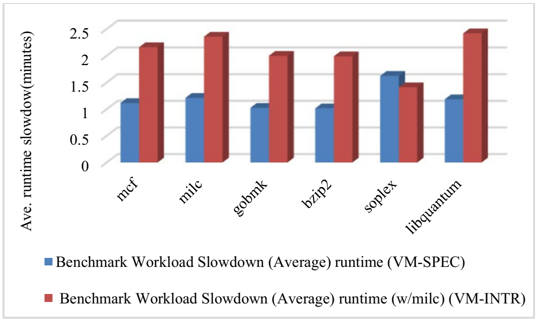

Figure 9: Test Case 2 - Benchmark Workloads Runtime Slowdown (Average) in a Co-Located Interference Prone Environment with Standard Resource Management

In figure 9, benchmark workloads submitted in VM-INTR (Tenant #2) shows significant slowdowns compare to workloads submitted in VM-SPEC (Tenant #1) even with workloads of similar characteristics e.g. *milc*.

Further, we expand equation (12) to demonstrate the value add of ACS and its impact on the overall application benchmark workload performance. We measure the benchmark workloads runtime slowdown (average) when the benchmark workloads execute in standard resource sharing mode, and when the benchmark workloads execute in ACS mode. Thus, we denote by;

- (P_K); application *workloads runtime slowdown (average)* with standard resource sharing as;

$$(P_k) = \left(\frac{\sum_{i=1}^{n} t_{(k1+k2+k3+k4+k5+k6\ldots\ldots kn)}}{n}\right) \quad (13)$$

- (P'_z); application *workloads runtime slowdown (average) with ACS* sharing as;

$$(P'_z) = \left(\frac{\sum_{i=1}^{n} t_{(z1+z2+z3+z4+z5+z6\ldots\ldots zn)}}{n}\right) \quad (14)$$

Therefore, we use metrics from Table 10 (Test Case 2) and Table 11 (Test Case 3) to support our experimental evaluation.

TABLE 11: TEST CASE 3 - BENCHMARK APPLICATION WORKLOADS RUNTIME SLOWDOWN (AVERAGE) WITH ADAPTIVE CPU SHARING

Benchmarks	**workload slowdown (average) runtime (VM-SPEC)**	**workload slowdown (average) runtime (w/milc) (VM-INTR)**
Mcf	1.873790166	1.629943503
Milc	1.508474576	1.45480226
Gobmk	1.162420382	1.059322034
Bzip2	1.858306195	1.706214689
Soplex	1.46137339	1.209039548
Libquantum	2.18016529	1.378531074

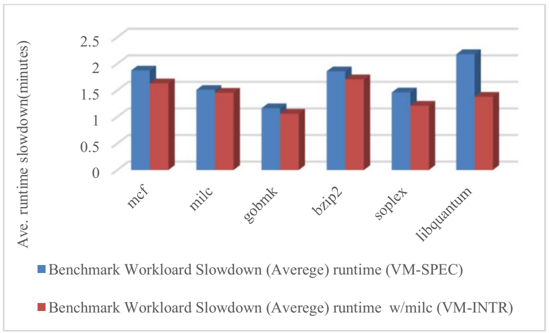

Figure 10: Test Case 3 - Benchmark Workloads Runtime Slowdown (Average) a Co-Located Interference Prone Environment with Controlled Resource Management

In Table 11 and figure 10, the benchmark workloads runtime slowdown shows reduction in unfairness slowdown between the background and foreground application workloads runtime. On the hand, in Table 10 and figure 9 the application workloads runtime shows unequitable slowdown between the background and foreground application workloads runtime.

Figure 9 and figure 10 shows the contrast between benchmark workloads runtime slowdown (average) in standard resource sharing mode and Adaptive CPU sharing mode. Figure 10 depicts that the ACS approach reduce unfairness and among co-tenants.

Additionally, we are interested in ensuring that the individual application runtime slowdowns with our approach is compared to, if not better than the individual application runtime slowdown with the standard Xen resource scheduler.

We are more interested in finding out the extent to which the overall application performance is affected in term of performance degradation based on the application runtime average. Subsequently, we can quantitatively compare the application performance in term of the workloads slowdown when running with standard Xen CPU sharing and Adaptive CPU sharing using metrics from our experiments.

To this end, two separate *milc* experimental metrics from Test Case 2 and Test Case 3 were evaluated to establish the impact of the Adaptive CPU sharing on application performance in multi-tenant cloud environment.

- *Milc* benchmark workloads runtime slowdown (average) with standard resource sharing (Test Case 2).
- *MILC* benchmark workloads runtime slowdown (average) with adaptive CPU sharing model (Test Case 3).

We compare the slowdown runtime (average) of *Milc* benchmark workloads when running in *standard resource sharing* (P_K), and *Milc* benchmark slowdown runtime (average) when running in *Adaptive CPU sharing* (P'_Z).
Table 12 and 13, shows the consolidated runtime slowdown (average) for *milc* benchmark workloads with standard resource sharing and *milc* benchmark workloads runtime slowdown (average) with Adaptive CPU sharing respectively.

TABLE 12: TEST CASE 2 - MILC BENCHMARK WORKLOADS RUNTIME SLOWDOWN (AVERAGE) WITH STANDARD RESOURCE SHARING

Runtime instance (n)	**workload slowdown (average) runtime (w/milc) (VM-INTR)**
$T_{(K1)}$	2.159604517
$T_{(K2)}$	2.357344636
$T_{(K3)}$	2.000000000
$T_{(K4)}$	1.991525424
$T_{(K5)}$	1.411016949
$T_{(K6)}$	2.419491525

TABLE 13: TEST CASE 3 - MILC BENCHMARK WORKLOADS RUNTIME SLOWDOWN (AVERAGE) WITH ADAPTIVE CPU SHARING

Runtime instance(n)	**workload slowdown (average) runtime (w/milc) (VM-INTR)**
$T_{(Z1)}$	1.629943503
$T_{(Z2)}$	1.45480226
$T_{(Z3)}$	1.059322034
$T_{(Z4)}$	1.706214689
$T_{(Z4)}$	1.209039548
$T_{(Z6)}$	1.378531074

Using equation 13, 14, and metrics from Table 12 and Table 13, we compare the *overall application runtime slowdown (average)* (P_n), between *milc* benchmark workloads runtime slowdown (average) (P_K) – (standard resource sharing), and *milc* benchmark workloads runtime slowdown (average) (P'_Z) – (Adaptive CPU sharing);
Thus, we calculate;

- $(P_K) =$ *milc* benchmark workloads runtime slowdown (average) with standard resource sharing = **2.05649(minutes)**
- $(P'_Z) =$ *milc* benchmark workloads runtime slowdown (average) with ACS = **1.40630(minutes)**

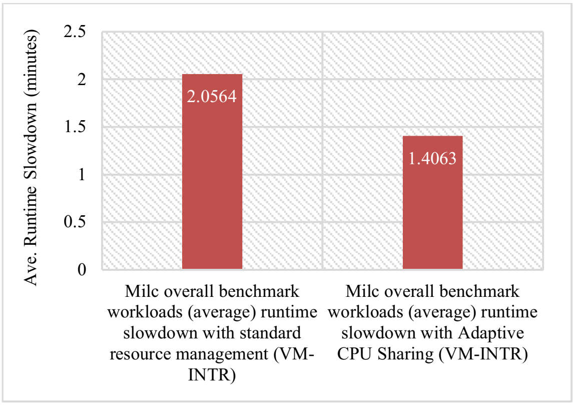

Figure 11: Milc Benchmark Workloads Runtime Slowdown (Average) with Standard Resource Sharing, and Adaptive CPU Sharing

Figure 11, shows the overall application performance difference (P_n) between *milc* benchmark workloads runtime slowdown (average) (P_K), and *milc* benchmark workloads runtime slowdown (average)(P'_Z).
Based on the *milc* benchmark workloads average runtime, our approach improves the overall *milc* benchmark workloads runtime (average) slowdown by 37.54% compared to *milc* benchmark workloads runtime (average) slowdown with Xen standard resource controller.

VII. CONCLUSION AND FUTURE WORK

This paper addresses the issue of unpredictable application performance in multi-tenant. Our approach in addressing unpredictable application performance due to resource contention among co-tenants is unique in that we have use fine-grained low-level system metrics information to inform CPU allocation to each tenant and reduce unfairness in application performance degradation. Through series of experiments, we identified various application performance characteristics, and we reduce the imbalances in resource allocation and improve application performance slowdown (average) without intrusive modification of the hypervisor. We consider both the contention on CPU time as well as contentions on the shared hardware resources (memory) to accomplish efficient prediction of application performance.

Our approach adaptively allocates CPU to ensure equal slowdown runtime for all co-running applications. Therefore, dynamic allocation of CPU as a shared resource is a viable solution to mitigate interference and reduce unfairness among cloud tenants in multi-tenant cloud environments. Further, we demonstrate that contention on shared resources, such as last-level CPU caches, has salient impact on application performance. To monitor hardware-level statistics in a fine-grained manner and ultimately realize online performance prediction, we developed a set of tools to record hardware events, and obtain important CPU allocation at the hypervisor level, and throttle CPU allocation to reduce co-located applications performance degradation. The over-arching goal of this work is achieve by reducing the overall application performance degradation in an equitable manner among cloud tenants. Additionally, based on our experimental results, ACS incurs lower unfairness that the default Xen scheduler.

Therefore, we propose ACS as a novel approach that helps mitigate co-tenant interference, reduces unfairness by minimizing the application slowdowns, and improves in-cloud application overall performance. Experimental tests were performed on a private multi-tenant cloud environment with micro-benchmark workloads that simulate certain aspects of real-world application workloads; however, results may vary with actual workloads carried out on different environment.

REFERENCES

[1] Anthony Ayodele, Jia Rao, and Terrance E. Bolt, "Performance Measurement and Interference Profiling in Multi-tenant Clouds" in Proceeding of IEEE 8th International Conference on Cloud Computing (CLOUD 2015), June 27 – July 2, 2015, New York, USA.
[2] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In ASPLOS-19, 2010.
[3] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In ISPASS, 2001.

[4] Di Xu, Chenggang Wu, Pen Chung Yew, Jianjun Li, and Zhenjiang Wang, " Providing fairness on shared-memory multiprocessors via process scheduling", in Proceeding of the 12th Joint International Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS, June 11-15, 2012, London, UK.

[5] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2

[6] Understanding CPU Steal Time – When should you be worried? http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when-should-you-be-worried.

[7] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai, " Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor", in Proceedings of the 22nd international conference on Parallel architectures and compilation techniques, September 7-11, Edinburg, Scotland.

[8] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Federova, and Manuel Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors", ACM Computing Surveys, Vol. V, No. N, September 2011, Pages 1–31.

[9] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel1, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment", in Proceedings of ACM/USENIX Conference on Virtual Execution Environments, June 11-12, Chicago, Illinois, USA

[10] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova, "Contention-Aware Scheduling on Multicore Systems", ACM Transactions on Computer Systems, Vol.28, No.4, Article 8, December 2010.

[11] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah, "Q-clouds: managing performance interference effects for QoS-aware clouds", in Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010), April 13-16, 2010, Paris, France.

[12] Jason Nieh, Chris Vaill, and Hua Zhong "Virtual-Time Round-Robin: An O(I) Proportional Share Scheduler", USENIX Annual Technical Conference, pp. 245 – 259, 2001.

[13] Carl A. Waldspurger, and William E. Weihl, " Lottery Scheduling: Flexible Proportional-Share Resource Management", Lottery Scheduling: Flexible Proportional-Share Resource Management, Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994.

[14] Jon C.R. Bennett, And Hui Zhang, " WF^{2}Q: Worst-case Fair Weighted Fair Queueing", INFOCOM 1996, pp. 120 – 128.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, "Xen and the Art of Virtualization", in Proceeding of the nineteenth ACM symposium on Operating systems principles (SOSP'03), October 19–22, 2003, Bolton Landing, New York, USA

[16] Corey Malone, Mohamed Zahran, and Ramesh Karri, "Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs", in Proceeding of the Sixth ACM workshop on Scalable trusted computing, Oct 17-21, Chicago, IL, U.S.A

[17] Ruslan Nikola, and Godmar Back, "Perfctr-Xen: A Framework for Performance Counter Virtualization", in Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11). ACM, New York, NY, USA, 15-26.

[18] M. Petterson. Perfctr library. http//user.it.uu.se/~mikpe/linux/perfctr/, 2011

[19] S. T. King, G. W. Dunlap, and P. M Chen, "Operating System Support for virtual machines", in Proceeding of the 2003 USENIX Annual Technical Conference.

[20] L2 Cache, http://www.cpu-world.com/Glossary/L/Level_2_cache.html

[21] Intel® 64 and IA-32 Architectures - Software Developers Manual (Vol. 3 (3A & 3B): System Programming Guide, p. 535). (n.d.). Intel Corporation.

[22] Intel® 64 and IA-32 Architectures - Software Developers Manual (Vol. 3 (3A & 3B): System Programming Guide, p. 1385). (n.d.). Intel Corporation.

[23] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Zu, "Optimizing Virtual Machine Scheduling in NUMA Multicore Systems", in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA-19), February 23-27, 2013, Shenzhen, China.

[24] Ludmila Cherkasova, Diwaker Gupta, Amin Vahdat, Comparison of the three CPU Schedulers in Xen, ACM SIGMETRICS Performance Evaluation Review, v.35 n.2, p.42-51, September 2007.

[25] Essick R. B., "An Event based Fair Share Scheduler", In Proceedings of the 1990 Winter USENIX Conference, Washington, DC, pp. 147-161, Jan 1990.

[26] Credit Scheduler http://wiki.xensource.com/xenwiki/CreditScheduler.

[27] Ryan Hnarakis, "In Perfect Xen, a Performance Study of the Emerging Xen Scheduler", Dec 2013.

[28] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, Feb 2009

[29] Xen Users' Manual Xen v3.3 http://wiki.xenproject.org/mediawiki/images/4/47/Xen_3_user_manual.pdf

[30] J. Faraway. Practical Regression and Anova in R. July 2002.

[31] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu, "an Analysis of Performance Interference Effects in Virtual Environments" in Proceeding of IEEE International Symposium on Performance Analysis of Systems & Software, 2007. ISPASS 2007.

[32] Sajib Kundu, Raju Rangaswami, Kaushik Dutta, and Ming Zhao, "Application Performance Modeling in virtualized environment" in Proceeding of IEEE International Symposium High Performance Computing Architecture, 2010.

[33] V.N. Vapnik, The Nature of Statistical Learning Theory. New York, NY, USA, Springer, 1995.