Fair Pricing in Infrastructure-as-a-Service Clouds via Snapshots

ABSTRACT

Infrastructure-as-a-Service (IaaS) cloud computing has recently emerged as a new paradigm for cost effectiveness. This allows users to pay a flat hourly rate for running their applications on virtual machines (VMs). However, virtual machines belonging to different users co-located on a same physical machine may interfere with one another, causing performance degradation of individual applications. The magnitude of this degradation is highly variable as it depends upon the nature of co-located applications. Since neither providers nor clients have control over the nature of co-running applications, performance degradation due to interference can lead to a biased pricing scenario. For example, a user who pays flat hourly rate for its VM might end up paying more when contentious VMs from other users are co-located on a same server.

Providing fair pricing requires accurate estimation of performance degradation due to interference between co-located VMs. Estimating performance interference is challenging especially in IaaS where cloud providers do not have information on individual VM's content. Moreover, any estimation technique should not incur much overhead in production environments.

To precisely estimate unintended performance degradation of individual VMs, this paper proposes *snapshot* technique, which pauses all the other VMs except for one of the VMs only when that VM undergoes a phase change. To enable this, we introduce a novel phase detection mechanism which examines the execution behavior for each VM in co-located environments. In this work, we show that we are able to accurately identify phase changes and estimate performance degradation within 4% mean absolute error on SPEC benchmarks with a very low overhead of around 1% when four VMs are running on a four core machine.

1. INTRODUCTION

Infrastructure-as-a-Service (IaaS) cloud computing enables users to take advantage of the computing infrastructures under the pay-as-you-go scheme. Cloud providers rely on virtualization to provide the isolated computing resources called instances (or VMs) to each customer. High resource utilization is achieved by consolidating virtual machines into a single server. However, consolidation of virtual machines belonging to different users may lead to performance interference with each other. Although public clouds provide a variety of instance types to satisfy different requirements from users such as the number of virtual CPUs, the amount of memory and disk, users cannot specify the requirements for shared architectural resources like last level cache, memory bandwidth and memory controller. The sharing of architectural resources can significantly affect the performance of each application [1–9].

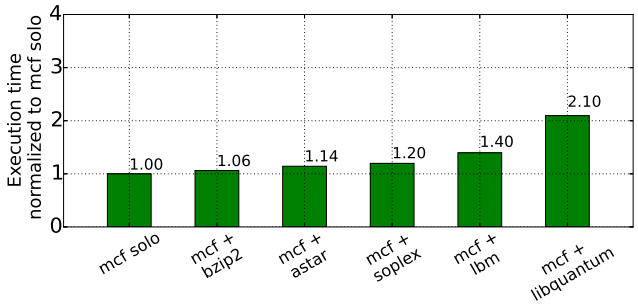


Figure 1: mcf with co-runners normalized to solo execution

Figure 1 shows the execution time when mcf is co-locating with different applications running on another VM. Each bar is normalized to solo execution of mcf. In such a scenario, libquantum, lbm, and soplex highly affect the performance of mcf. Especially, when mcf is running with libquantum, it would take 2.1x more time to complete compared to the case when mcf is running alone. In the perspective of users, this performance degradation is not acceptable because current IaaS cloud providers calculate the price based on the amount of time each user spends running applications. As a result, it causes biased pricing scenarios where a user might end up paying more when running with contentious co-runners due to increased execution time.

One of the possible solutions to the biased pricing problem is to accurately determine performance degradation for each VM. With this information, we could minimize the biased pricing scenarios. The most relevant prior work proposes runtime techniques for estimating performance degradation in HPC clusters [10, 11]. These techniques periodically pause all the other running applications except for one during a short amount of time to estimate solo performance while avoiding per-

formance interference. However, the prior studies show low accuracy in many applications, which is essential for fair pricing, and high overhead which makes it difficult to deploy in production environments. In addition, the prior studies are sensitive to the number of co-runners in both the accuracy and overhead.

The key observation of our approach is that we do not need to periodically estimate the performance degradation for each VM because the execution behavior of applications does not drastically change within a steady phase and the number of phase changes is not significant in many applications. Our *snapshot* technique pauses VMs only at phase changes. As a result, we can reduce the overhead of frequently pausing VMs at fixed time intervals. In addition, *snapshot* enables us to increase enough pausing time to accurately measure solo execution performance of each VM. Thus, even if the number of co-running VMs increases, *snapshot* is less susceptible to the accuracy and overhead problems. To enable *snapshot*, we propose a novel phase detection mechanism based on performance monitoring units (PMUs). Since each application has different execution behavior, we investigate the representative hardware events to detect phase changes across all SPEC applications used in our experiment. Through the selected performance monitoring units, we can effectively capture phase changes during runtime. Our Fair Pricing Runtime engine is based on our *snapshot* on top of this phase detection mechanism. The contributions of this paper are as follows.

- **Fair Pricing Runtime:** We introduce a lightweight, scalable, and deployable runtime system that enables fair pricing in public clouds through the precisely estimated solo performance of applications when co-located.
- **Snapshot:** To efficiently estimate solo performance of applications in public clouds, we use *snapshot* technique, which pauses all other VMs except for one during a short time, on phase changes.
- **Phase Detection Mechanism:** We design phase detection mechanism to figure out the phase changes of applications during runtime. We also efficiently eliminate false positives as well as spikes by leveraging the multi-queue based binary classification technique.

Using our runtime enigne, we are able to precisely estimate the performance degradation in co-located environments with a mean absolute error of around 4% and negligible overheads of less than 1%. Comparing to the prior work we are able to estimate performance degradation at 2x more accuracy with 5x less overhead.

2. BACKGROUND AND MOTIVATION

In this section, we discuss the background of public IaaS clouds. Next, we investigate the prior technique in detail and discuss its accuracy and overheads issues in estimating performance degradation.

2.1 Background: Public Clouds for IaaS

To build Infrastructure-as-a-Service(IaaS), public clouds have adopted the virtualization technology. Each customer is provided with one or more virtual machines commonly called instances. Each public cloud provides a variety of instance types to satisfy the various demands from users. These instance types can be broadly categorized as compute, memory, I/O, etc. In addition, customers choose suitable resource capabilities such as the number of virtual CPUs, the amount of memory and storage size based on their requirements. The pricing strategy differs based on the instance types and capabilities. To use the IaaS cloud services, you have to pay the fee under the pay-as-you-go scheme where the price is proportional to the execution time of the VM.

Although the pay-as-you-go scheme is an attractive pricing model, it overlooks the fact that the computing resources can be shared between different customers. The application performance of users could be easily affected by other customers who extensively use shared computing resources. If a virtual machine undergoes performance degradation due to resource contention, the victim users end up paying more price due to their increased execution time. To complement the fairness problem, we could consider having strict quality of service (QoS) between service providers and customers. However, it is difficult to define QoS metrics in public clouds because we would not know the type of the running applications. Even if we are aware of its broader category such as web servers or batch applications, the same type of application could have different execution characteristics. For example, there can be two users running web servers on clouds, but each web server might be customized for their own purpose. Then, the QoS metrics like request per second (RPS) might be difficult to use.

The most accurate way of eliminating biased pricing schemes is by figuring out performance degradation that occurs due to co-running applications during runtime. However, it is challenging in co-located environments like public clouds. Especially, the architectural resources such as last level cache and memory bandwidth, which are shared among users, makes it difficult to estimate performance degradation for each user's VM in clouds. In this paper, we focus on the batch applications which are highly affected by the architectural shared resources. There have been prior efforts to estimate the performance degradation in co-located environments. We will discuss the issues of the prior technique in the next section.

2.2 Motivation: Accuracy and Overheads of Prior Work

To precisely measure the amount of performance degradation caused by co-running applications, the prior studies propose a technique called *shuttering* which pauses all the running VMs for a very short time except for one VM repeatedly at fixed time intervals [10,11].During the pausing time, a VM can monopolize computing resources on a system and they can easily extract the

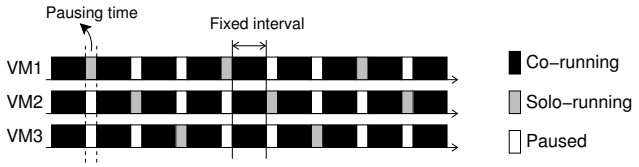


Figure 2: Pausing all other VMs during fixed time intervals

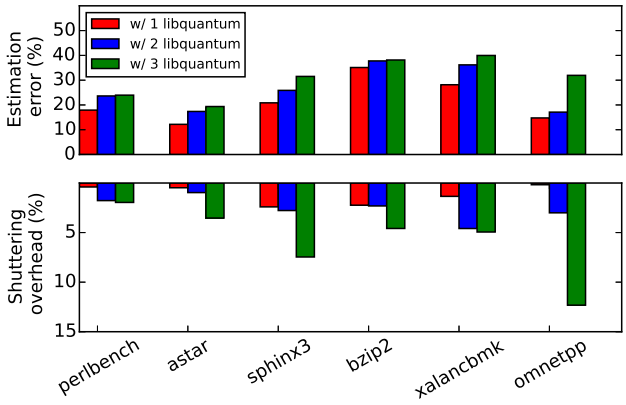


Figure 3: Error rates and overheads in shuttering

solo performance of applications. Figure 2 shows how the prior technique works to estimate solo performance of an application running in a VM.

Although this is a very simple and straightforward technique, it suffers from low accuracy in estimating the solo performance of applications. The top part of Figure 3 shows the error of estimating solo performance of applications. The baseline is CPI of each application without any co-runners. Each bar is normalized to the baseline when using the shuttering technique. The error rate is not negligible and increases to around 40% in `bzip2` and `xalancbmk`. The main reason is that the pausing time (3.2ms) used in prior work is not be enough to capture solo performance of an application. This is because the shared cache would not be warmed up to be containing the entire working set of the application which is to be measured. As a result, the measured application would spend most of its pausing time filling in the shared cache, giving much less time to observe how the application performs when it monopolizes computing resources. Moreover, as the number of co-runners increases, the shared cache becomes much more polluted due to the contention among the multiple co-runners. The effective time of precisely estimating solo performance of applications becomes much lower.

In addition, the bottom of Figure 3 shows the increased execution time for estimation of performance degradation as the number of co-runners increases. The source of the increased overhead is that the cache blocks belonging to the paused VMs would be eventually evicted at the end of the pausing time. As a result, when the paused VMs are resumed, they would pay an additional

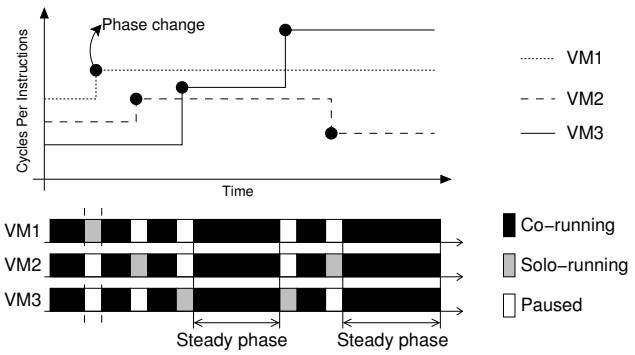


Figure 4: Snapshot technique

cost to warm up the cache. Moreover, as the number of co-running VMs increases, the overhead effects due to the prior is technique aggravated.

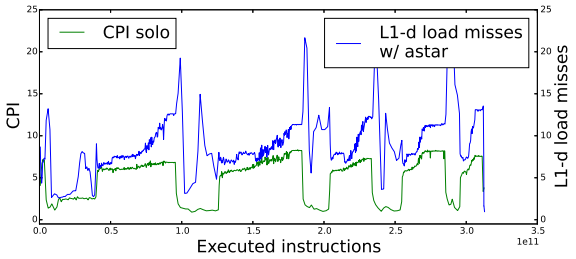
3. FAIR PRICING RUNTIME ENGINE

In this section, we introduce an efficient and scalable technique, called *snapshot*, to precisely identify unintended performance degradation in public clouds. To enable *snapshot*, we propose a phase detection mechanism based on selected performance monitoring units (PMUs). Our Fair Pricing Runtime Engine is based on *snapshot* technique on top of the phase detection mechanism.

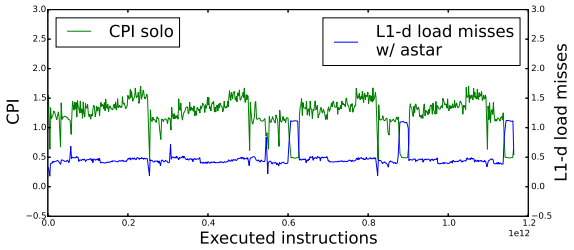
3.1 Snapshot technique

Our goal is to achieve high accuracy with low overhead in estimating unintended performance degradation on public clouds. We exploit two key observations which lead to our *snapshot* technique. Firstly, the execution behavior of applications does not drastically change within a steady phase. In other words, the CPI of a particular phase of the application remains fairly constant. Hence, we do not need to periodically estimate performance degradation during a steady phases. When a phase changes occur, we only have to take a snapshot once to investigate performance degradation during that steady phase. Secondly, the number of phase changes is not significant in many applications. It gives us an opportunity to optimize our snapshot technique for common cases where applications have few phase changes. In addition, even if applications have irregular behavior during its runtime, our snapshot technique has an ability to trace the performance degradation more accurately than prior approach. Therefore, if we accurately detect phase changes under the contentious environments, we can significantly reduce the overhead and achieve high accuracy for estimating performance degradation.

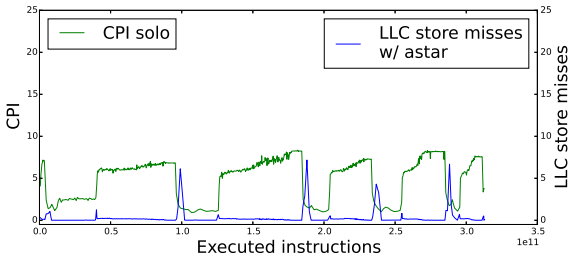
Moreover, since we take a snapshot only at phase changes, it enables us to snapshot for a longer time so as to accurately measure performance degradation of applications. If the measurement time is long enough to evict the cache blocks belonging to the co-running VMs and to bring the working set of the VM (which



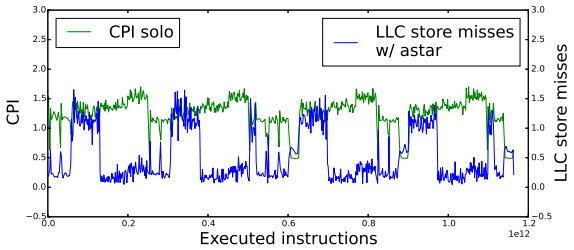
(a) mcf with L1-d cache load misses



(b) milc with L1-d cache load misses



(c) mcf with LLC store misses



(d) milc with LLC store misses

Figure 5: Detecting phase changes when running with 3 instances of `astar`

is to be measured) in the cache, we could achieve high accuracy. In addition, even as the number of co-runners increases, our *snapshot* technique is less susceptible to overhead issues as phase changes rarely occur.

Figure 4 shows how our *snapshot* technique estimates performance degradation by taking snapshot only when one of the applications undergoes phase changes. During the snapshot, we can easily estimate solo performance (CPI) of the application. By using this information, we will be able to estimate CPI of solo execution by using the following equation.

$$CPI_{(solo)} = \frac{CPI_{(1)} \times T_{(1)} + CPI_{(2)} \times T_{(2)} + ... + CPI_{(n)} \times T_{(n)}}{T_1 + T_2 .... + T_n} \quad (1)$$

- $CPI_{(solo)}$ is estimated CPI of solo execution of an application.
- $CPI_{(i)}$ is estimated CPI of solo execution of the application during phase i.
- $T_{(i)}$ is time for which the application remains in phase i.
- n is total number of phase changes for the application.

The $CPI_{(co-location)}$ is directly measured when the application is running with the co-runners. From the values of $CPI_{(co-location)}$ and $CPI_{(solo)}$, we estimate the degradation using the the following equation.

$$PerfDeg = \frac{CPI_{(co-location)}}{CPI_{(solo)}} \quad (2)$$

Thus, accurately detecting phase changes is very important to make *snapshot* technique viable. We will discuss this in the next section.

3.2 Phase Detecting Mechanism

In this section, we introduce our phase detecting mechanism to enable *snapshot* technique in public clouds. If there are no co-runners, we could easily detect phase changes by capturing CPI changes during solo execution of an application. However, public clouds, which allow the co-location of VMs on a single server, make it difficult to precisely estimate phase changes. This is due to the fact that CPI of an application could be highly affected by other applications running in co-located VMs. As a result, there is a possibility of falsely detecting phase changes due to the interference from co-runners. To make our *snapshot* technique more effective, we need to minimize the number of falsely detecting phases because if we frequently detect wrong phase changes due to the interference, the number of unnecessary shuttering would increase. As a result, it would increase the overhead of our *snapshot* technique.

To efficiently detect true phase changes, which are inherent phase changes of applications, while avoiding falsely detecting phases, we need additional information apart from CPI. For this purpose, we observed usage patterns of architectural resources over time. Our hypothesis is that true phase changes of applications can be clearly identified by observing architectural resource usages. We took advantage of performance monitoring units (PMUs) to observe various architectural events. PMUs are well known for its low overheads and easily deployable into real systems.

Figure 5 shows that each application requires different types of PMU to precisely detect phase changes for two applications. The x-axis indicates the cumulative number of instructions executed as time progresses. The left y-axis and green line show CPI of the applications when running alone and the right y-axis and the blue line show estimating phase changes by a selected

App.	PMC Types		
astar	**CPI**	branch	L1-d load miss
bzip2	**LLC store miss**	CPI	L1-d load miss
cactus.	**L1-d load miss**	L1-d load	CPI
dealII	**CPI**	L1-d load	branch
mcf	**L1-d load miss**	CPI	LLC load
milc	**LLC store miss**	L1-d load	branch
xalan.	**LLC store miss**	LLC load	L1-d load
tonto	**L1-d load miss**	branch	CPI

Table 1: PMU types ordered by their effectiveness in detecting phase changes.

performance counter for the application when running with three instances of `astar` as co-runners. From Figure 5a, we find out that the performance counter, *L1-d cache load misses*, can effectively detect inherent phase changes of `mcf` in co-located environments whereas Figure 5b shows that the same type of PMU is unable to detect inherent phase changes of `milc`. On the other hand, *LLC store misses* is able to detect inherent phase changes for the `milc` as shown in Figure 5d whereas the same type of PMU is unable to detect inherent phase changes of `mcf` as shown in Figure 5c. These results motivate us to need multiple types of PMU to capture phase changes across applications.

3.2.1 Finding Representative Types of PMU for Phase Detection

To capture a wide range of execution behavior across applications, we monitor a set of 6 different types of PMU, *CPI, L1-D cache load miss, L1-D cache load access, LLC store misses, LLC store access*, and *branch instructions*. We select eight training SPEC 2006 applications which are known to have varying phase changes. We investigate which types of event can capture phase changes for those eight applications. In this experiment, we used a quad-core machine and 3 instances of `astar` as co-runners.

We first identify the best type of PMU for each application individually. This is identified based on two criteria. Firstly, the best type of PMU should detect all inherent phase changes present in solo execution of the application. Secondly, it should minimize falsely detecting phase changes due to the interference from co-runners. In other words, among the six types of PMU, which can detect all inherent phases present in solo execution of the application, the best type of PMU is the one which has the lowest number of falsely detected phase changes. The reason behind this is that minimizing the number of falsely detecting phase changes will reduce the overheads of pausing the VMs unnecessarily. Based on the best type of PMU obtained for the training applications, we can select a subset of PMU types to detect phase changes.

Table 1 shows the preferred types of PMU for eight applications ordered by the effectiveness in which they are able to detect phase changes. Due to the space limit, the table shows only three types of PMU among six types. Based on this order, we can choose a common subset of PMU types which can detect phase changes effectively across the training set of applications. Using this method, we obtain three types of PMU, *CPI, LLC store miss* and *L1-d cache load miss*, to capture phase changes for the eight training SPEC applications.

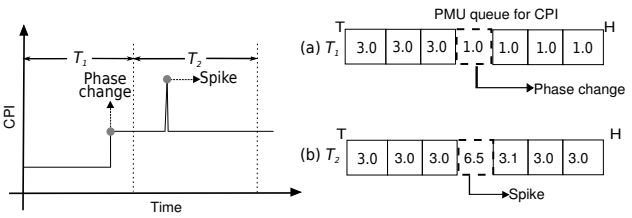


Figure 6: Differentiating spikes and phase changes

3.2.2 Eliminating Spikes to Avoid False Phase Detection

While observing phase changes of the training applications, we notice that there are spikes in the reported PMU measurements. These spikes occasionally occur during a short interval of time and show significant variations in the execution behavior. In such a case, our phase detection methodology should not treat spikes as phase changes.

To distinguish between spikes and true phase changes, we employ a binary classification technique. We maintain a queue per each type of event. Each queue contains k latest values that have been measured by a type of PMU. The value of k is empirically determined by repeating experiments with different k values and optimizing for value which is enough to differentiate phase changes and spikes. We know that whenever there is a phase change associated with an application, subsequent PMU measurements fall under a different range which corresponds to a completely new phase. On the other hand, whenever there is a spike, the PMU measurements show drastic changes for one or two values and the rest falling under the same range. In order to eliminate such drastic changes due to spikes, we declare a phase change only when a significant number of the values present in the queue belongs to a new range. In this way, we are able to eliminate incorrectly detecting phase changes due to spikes.

Figure 6 shows an example differentiating spikes and phase changes. During T_1 in Figure 6, the range of a significant number of elements changes to 3.0 compared to a previous phase which is around 1.0. In such a case, we consider this as a phase change where we take a *snapshot* of the new phase by pausing all co-runners. On the other hand, during T_2, the CPI of every element in the queue is closely around 3 except for one which is around 6.5. This change is considered as a spike and is not classified as a new phase.

3.2.3 Detecting Phase Changes

In the previous sections, we discussed how to select three types of event to detect phase changes and how to eliminate spikes while detecting phase changes. In this section, we introduce how we can incorporate these two techniques to identify unintended performance degradation in public clouds.

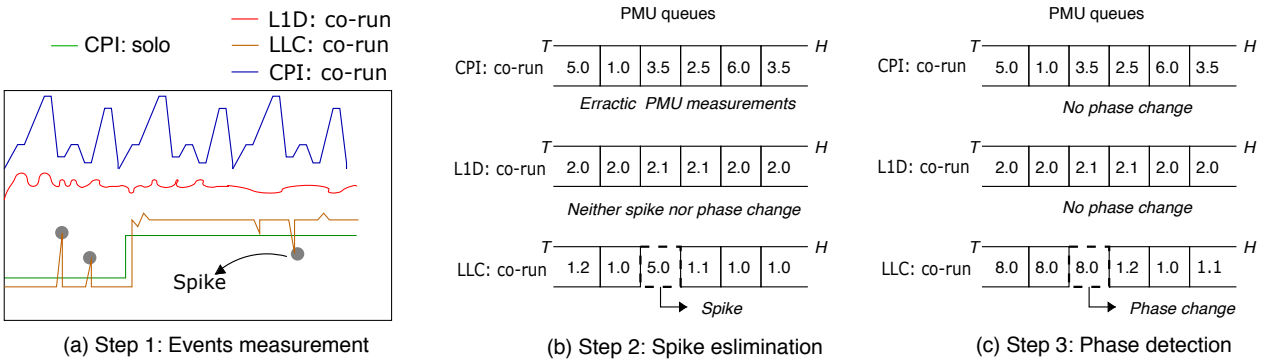


Figure 7: Process of detecting phases

Figure 7 describes the process of the phase detection taking place in the Fair Pricing Runtime engine. Our runtime engine collects three types of events every second as shown in Figure 7(a). It inspects whether there is a significant change in the range of the measured events by comparing it to the corresponding PMU measurements at the most recent phase change. In this process, it also discards spikes associated with PMU measurements as shown in Figure 7(b). To avoid missing true phase changes, we use a conservative approach to call for a phase change even if one of the PMU types out of the three detects a phase change. If we do not detect true phase changes, it will significantly reduce the accuracy in estimating CPI of solo execution. On the other hand, mispredicting phase changes causes only negligible overhead if the frequency of such events is low. Once a new phase is detected, the runtime engine then pauses co-running VMs so as to estimate solo performance of the applications. In the example as shown in Figure 7(c), CPI:co-run, L1D:co-run and LLC:co-run indicate the PMU measurements for CPI, L1 d-cache load misses and LLC-store misses, respectively. From this figure, we are able to see that `LLC store misses` is able to detect the phase changes present in the application whereas CPI and L1 d-cache are unable to detect phase changes in runtime.

3.3 Pricing for Fairness

In this section, we will see how we use the proposed *snapshot* technique for enabling fair pricing in public clouds. Pricing in public clouds is based on an hourly basis which could create a biased scenario in co-located environments. We need a mechanism to charge each user based on how individual applications perform when they were running alone.

With the help of *snapshot*, we are able to accurately estimate the performance degradation in co-located environments with a very low overhead. This can in turn be used to price individual users based on the amount of time by which they have been degraded. From the estimated performance degradation, the price of each user can be calculated by the following equation.

$$P_i = BasePrice/PerfDeg_i \quad (3)$$

- P_i: price paid by user i.

Algorithm 1 Fair Pricing Runtime

$perfScoreVM_i$ = <CPI, LLC, L1D > ▷ A queue of performance counters per VM
$perfDegVM_i$ = <VM_1, ..., VM_n> ▷ Performance degradation for each VM

/* **Step 1:** Obtaining PMU measurements for all VMs */
for each VM_i in 1 ... n **do**
  $perfScoreVM_i$[CPI] <= gather_CPI(VM_i)
  $perfScoreVM_i$[LLC] <= gather_LLC_store_miss(VM_i)
  $perfScoreVM_i$[L1D] <= gather_L1d_cache_miss(VM_i)
end for

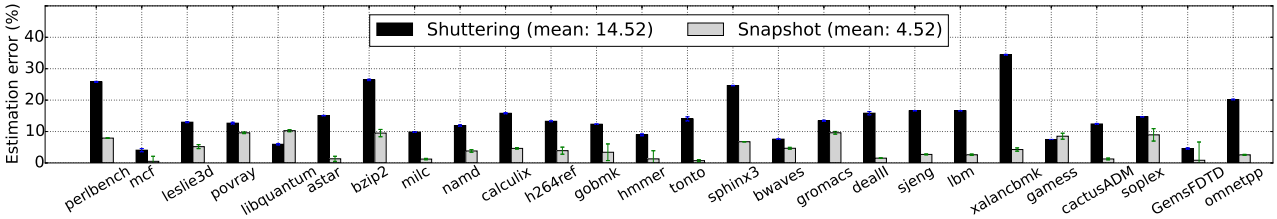
/* **Step 2:** Detecting phase changes by PMU types */
for each VM_i in 1 ... n **do**
  for each $PMUtype_j$ in 1 ... m **do**
    if check_phase_change (VM_i[j]) == true **then**
      pausing all co-running VMs except for itself;
      $perfDegVM_i$ <= difference(gather_CPI(VM_i) - $perfScoreVM_i$[CPI])
      resuming all paused VMs
    end if
  end for
end for

/* **Step 3:** Calculating price based on estimated degradation */
for each VM_i in 1 ... n **do**
  if check_perfDeg_vm(VM_i) == true **then**
    reflect the unintended performance degradation in its bill
  end if
end for

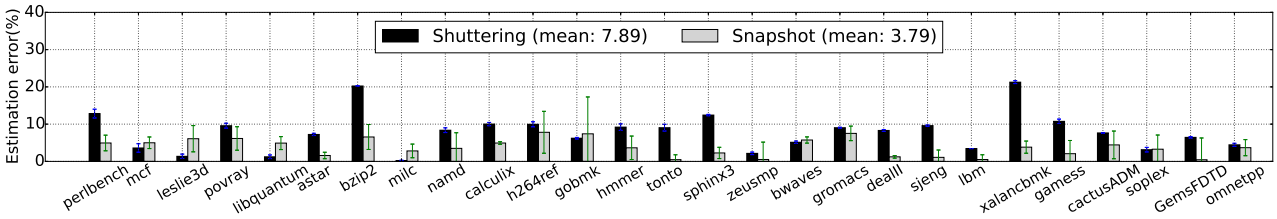
- $PerfDeg_i$: degradation suffered by user i from equation 2.

The $BasePrice$ in this equation is the share of price which each user would pay without taking into account the performance degradation due to co-running applications. The division of the base price with $PerfDeg_i$ charges each user i with a price proportional to the performance degradation. This is due to the amount of degradation that the application has been subjected to due to its co-runners.

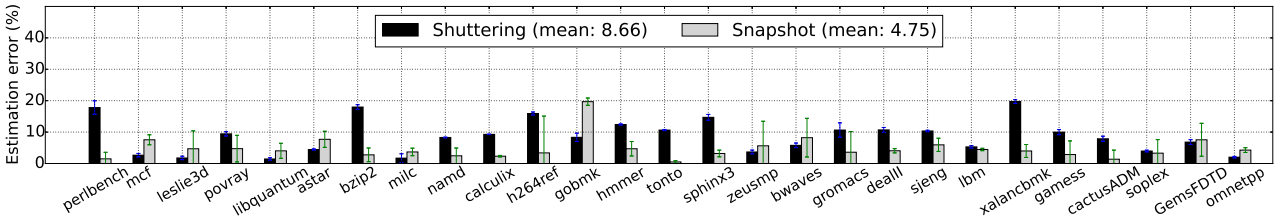
Algorithm 1 summarizes each step incurred by our Fair Pricing Runtime engine. The runtime consists of three functions. In step 1, it measures three types of PMUs every second for each VM. Step 2 of the algo-



(a) co-running with 3 instances of `libquantum`



(b) co-running with 3 instances of `mcf`



(c) co-running with 3 instances of `milc`

Figure 8: Accuracy of snapshot VS. shuttering in estimating performance degradation

rithm tries to detect phase changes based on the PMU measurements. If there is a phase change, then it pauses all other co-running VMs to measure CPI of solo execution for the application. Finally, we can calculate the price by reflecting the estimated performance degradation on the equation.

4. EVALUATION

In this section, we evaluate the effectiveness of our *snapshot* technique. Firstly, we look into how the *snapshot* technique can achieve the high accuracy to estimate performance degradation for each VM in co-located environments. In addition, we examine how our phase detection mechanism effectively works capturing phase changes at runtime. Secondly, we show the runtime overheads increasing execution time. In the experiments, we also compare our technique with the prior work and show that we achieve a high accuracy in estimating performance degradation with less overhead in most cases.

4.1 Experimental Setup

We evaluate our Fair Pricing Runtime engine on a Intel Xeon E5-2407 v2 (4-cores). We use *SPEC 2006* with `ref` inputs. To mimic IaaS public clouds, we take advantage of Linux KVM as the hypervisor and run applications on virtual machines [12]. In this evaluation, we leverage the hardware assisted virtual machine. Each virtual machine has 1 virtual CPU, 4GB main memory, and 16GB disk. We use Ubuntu 12.04 as guest operating systems with Linux kernel 3.11.0. We use the `perf` tool to measure hardware events [13].

Table 2 shows the experimental parameters that we have taken into consideration while building the runtime system.

Parameter	Shuttering	Snapshot
Mesurement frequency	200ms	1s (Checking phase changes is also performed at the same frequency.)
Pausing time	3.2ms	75ms (measurementtime) + 5ms (cache warm up time)

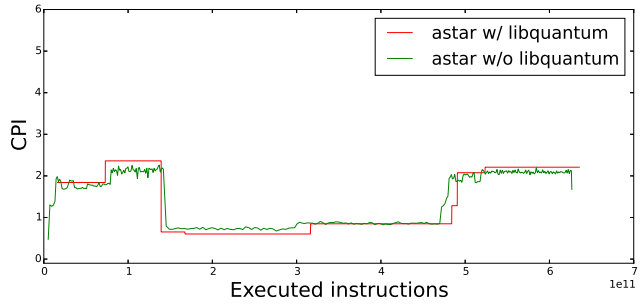
Table 2: Parameters for the runtime engine

4.2 Experimental Results

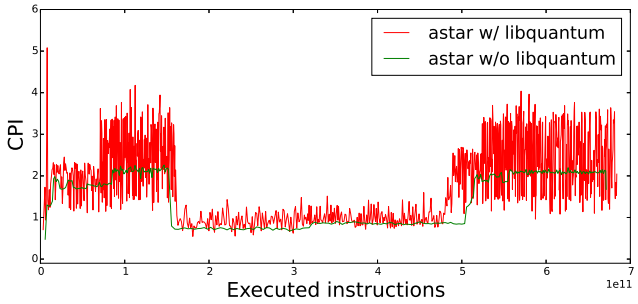
In this section, we evaluate the effectiveness of *snapshot* in terms of accuracy and overhead.

4.2.1 Accuracy of Fair Pricing Runtime Engine

In this sub-section, we evaluate the accuracy of our *snapshot* technique in estimating the degradation. To make a scenario similar to that present in public clouds, we run four virtual machines, where each virtual machine has 1 virtual CPU, on a quad-core machine and run SPEC applications on each VM. In this experiments, we measure the accuracy for a single SPEC application when running with three contentious co-runners. The reason why we choose contentious co-runners is that estimating performance degradation is very difficult due to their heavy impact on the shared last level

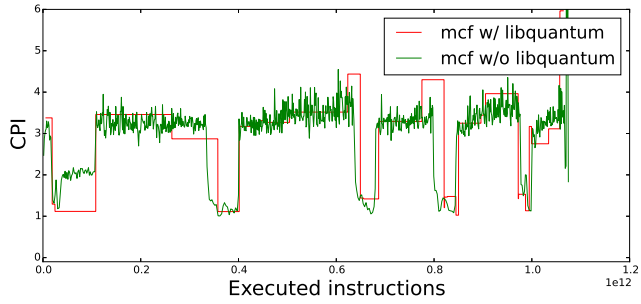


(a) Snapshot (error 1.31%)

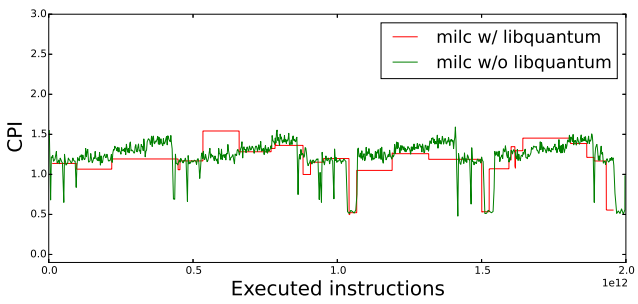


(b) Shuttering (error 15.06%)

Figure 9: Phase level behavior of snapshot and shuttering for `astar`



(a) mcf (error 0.51%)



(b) milc (error 1.23%)

Figure 10: Phase detection by snapshot for `mcf` and `milc` when running with 3 `libquantum` co-runners

cache and memory bandwidth.

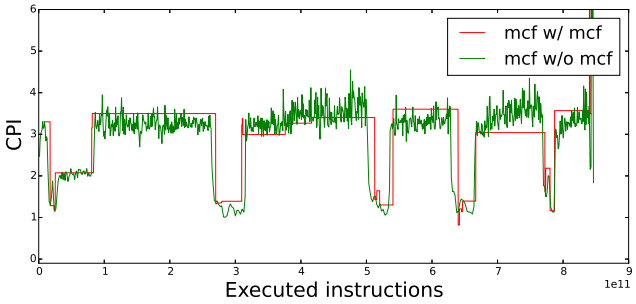
Figure 8 shows the accuracy of shuttering and *snapshot* techniques in co-located environment. The x-axis shows SPEC 2006 applications and the y-axis presents the error in estimating performance degradation. To see the effectiveness of our technique, we ran SPEC applications with 3 instantiations of the same co-running application. We perform the same experiments for the three different co-runners, `libquantum`, `mcf`, and `milc`, respectively. From many prior studies, these applications are well known to highly incur resource contention in shared architectural resources. The black bar indicates the prior work, and the gray bar represents our *snapshot* technique. We calculate the error for each technique by comparing the estimated performance degradation using both the runtime systems with the actual performance degradation. We run each benchmark three times and take the mean to minimize minor variability that exits while running SPEC applications. The legend in figures shows the mean error rate of all applications.

Through these experiments, we can see that the estimation error is much lower for *snapshot* technique than shuttering. When running with three instances of `libquantum` in Figure 8a, the mean error of prior technique is 14.5%. On the other hand, our *snapshotting* technique shows the estimation error of 4.5%. For `perlbench`, `bzip2`, `sphinx3`, `xalancbmk`, and `omnetpp`, the improvements are remarkable. When co-locating three instances of mcf and milc the error due to *snapshot* technique is 4.38% and 4.75%, respectively as shown by Figures 8b and 8c .

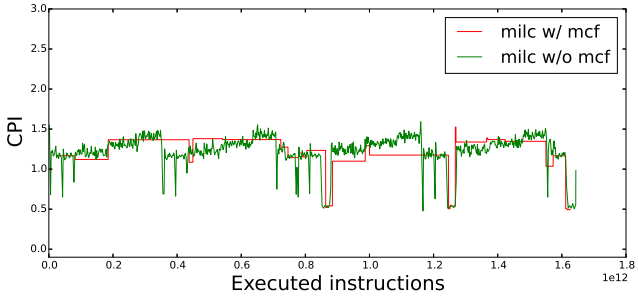
It is challenging to estimate the performance degradation when running with contentious co-runners as they quickly pollute the shared last level cache and excessively use the shared memory bandwidth. Such cases require more pausing time to bring its working set into the cache. If there is not enough time to warm up the architectural resources especially the cache, estimating solo performance of the application would be less accurate. We already know that the pausing time used in prior technique is not enough. Since they pause at regular time intervals, increasing the pausing time will cause significant execution time overheads of applications. On the other hand, our *snapshot* technique could increase the pausing time because it pauses the co-runners only at phase changes instead at regular intervals as being done by the prior technique.

4.2.2 *Phase Level Evaluation for Accuracy*

To see the behavior of *snapshot* in more details, we evaluate our technique phase by phase. As a case study, we choose `astar` as a target application and run 3 instances of `libquantum` as co-runners. Figure 9a and 9b present how *snapshot* and shuttering estimate the solo execution of `astar`, respectively. The green line depicts the phase of CPI for `astar` when running without any co-runners and the red line shows how each technique estimates the phase of CPI when running with 3 instances of `libquantum`. The closer the red line is to the green line, the lesser the error. Figure 9a shows that *snapshot* not only detects phase changes accurately, but also traces steady state periods of solo execution while



(a) mcf (error 3.41%)



(b) milc (error 2.52%)

Figure 11: Phase detection by snapshotfor `mcf` and `milc` when running with 3 `mcf` co-runners

minimizing the false positives. On the other hand, shuttering shows high error rate even within steady states because the pausing time is not enough to accurately estimate the solo execution. This is can be observed from the high variation during estimation as shown in Figure 9b.

Figure 10 and 11 shows how *snapshot* techinque efficiently traces the phase changes for `mcf` and `milc`. When running with `libquantum`, Figure 10a and 10b show that the red line shadows the green line. The error rates are 0.51% and 1.23%, respectively. We also run same application with `mcf` as a co-runner. Figure 11a and 11b exhibit that *snapshot* efficiently detects phase changes with low error rate when having `mcf` as a co-runner.

4.2.3 Overhead of Fair Pricing Runtime Engine

In this section, we evaluate the execution time overhead incurred by our runtime system. Figure 12 compares the overhead estimating performance degradation in co-located environments when using shuttering and *snapshots*. Each sub figure indicates the different co-runners, `libquantum`, `mcf`, and `milc`, respectively. We can see that the overhead of our *snapshot* technique is negligible. Even with `libquantum`, which is known to be highly contentious, the average across all applications is 0.45%. Even though shuttering has low overhead for quite a few applications, some of them such as `mcf`, `lbm`, `cactusADM`, `GemsFDTD`, and `omnetpp` exhibit high execution time overhead. The reason why the overhead is high in shuttering is that the number of cache misses increases due to the frequent pausing of co-runners. When running with `mcf` or `milc`, which is known to be less cache contentious than `libquantum`, Figure 12b and 12c show the decreased overhead of shuttering.

On the other hand, the overhead of *snapshot* is slightly higher than the overhead of shuttering when running with `mcf`. This is due to the fact that `mcf` has a lot of inherent phase changes. As a result, three instances of `mcf` as co-runners will lead towards frequently pausing the co-runners. We also observe that for applications such as `calculix` and `dealII`, the overhead of *snapshot* is slightly higher than other applications. The reason behind this is the fact that both applications exhibit highly irregular phases and the number of false positives is high.

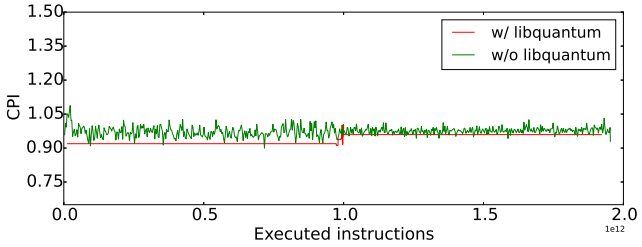


Figure 13: Phase behavior of `lbm` (error 2.59%)

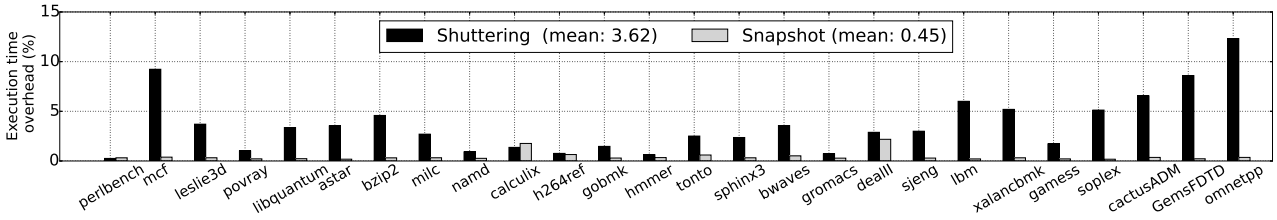
When running with `libquantum` and `milc`, most applications show the overhead less than 0.5% because these applications have a single phase throughout their execution. Therefore, the total pausing time while executing these applications is negligible. Figure 13 shows that `lbm` has a single phase. Our runtime only pauses the co-runners at its initial execution period to obtain solo CPI of `lbm`. There is no subsequent shuttering because of the absence of phase changes. Such a case incurs an extremely low overhead of around 0.1%.

4.2.4 Accuracy and Overheads of Different Types of Multiple Co-runners

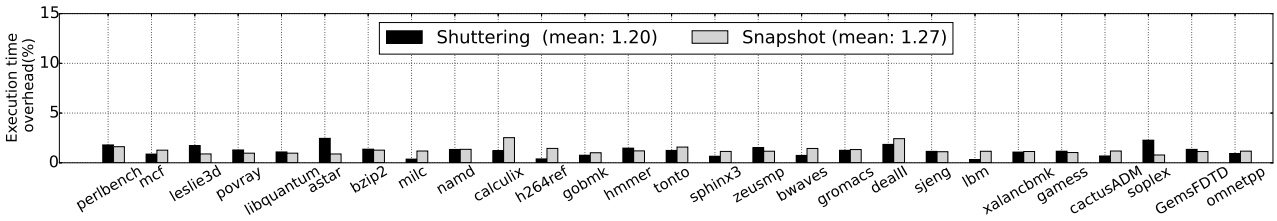
To see the effectiveness when running with multiple different co-runners, we evaluate the accuracy and overheads of every individual application present in SPEC when running along with 3 different co-runners `libquantum, milc and mcf respectively` in 3 different VMs on the same server. These 3 co-runners are selected based on covering a range of properties like contentiousness and sensitivity towards the last level cache and the number of inherent phase changes present in the co-running applications. Figure 14 shows the accuracy and overheads of SPEC applications when running with multiple different co-runners. Even with three different co-runners, we can see that our *snapshot* achieves high accuracy with negligible overhead.

5. RELATED WORK

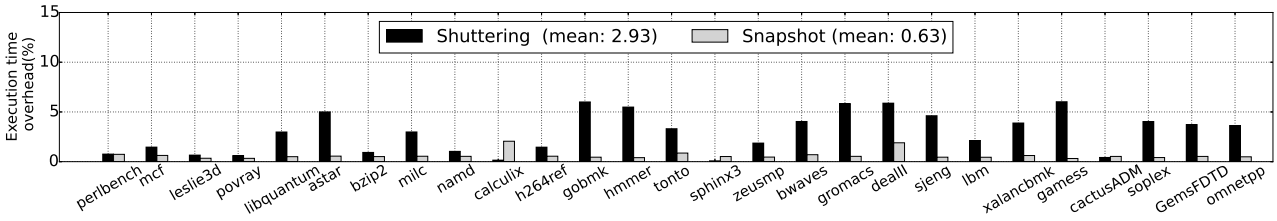
There have been many prior studies to minimize performance interference in a variety aspects of architectural resources. We categorize the prior work into two



(a) co-running with 3 instances of `libquantum`

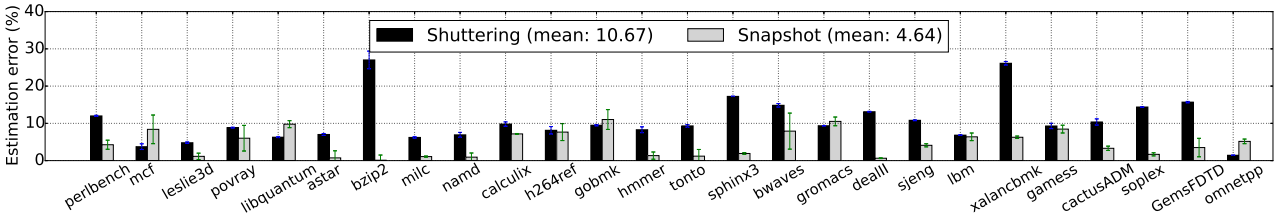


(b) co-running with 3 instances of `mcf`

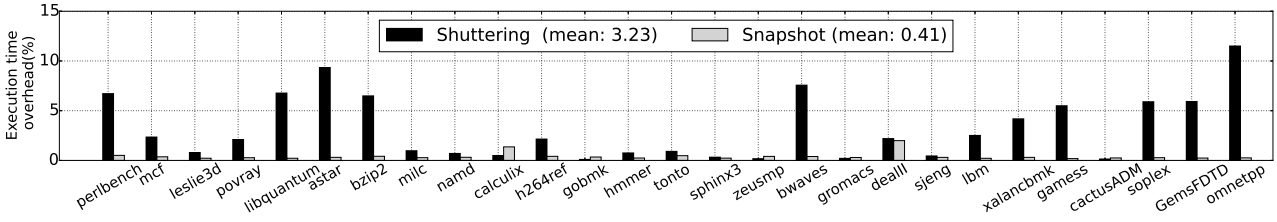


(c) co-running with 3 instances of `milc`

Figure 12: Overheads of snapshot VS. shuttering while estimating performance degradation



(a) Accuracy



(b) Overheads

Figure 14: Accuracy and overheads of snapshot when running with co-runners libquantum, mcf and milc

broad types. We look into the system and OS level approaches and then address the architectural supports for minimizing the interference.

System/OS approaches: There are many efforts introducing software frameworks and proposing the new designs of operating systems [1, 2, 14–18]. Q-Cloud measures the resource capacity for satisfying QoS in a dedicated server called staging server and then decides the placement which server will be profitable to minimize the interference. Nevertheless, the QoS could be violated by allowing co-location. To avoid this situation, the system provides additional resources from head-room by reserving presubscribed amount of resources. If the placement meets the target QoS, the head-room would be utilized in a best-effort manner [1]. To precisely estimate the performance interferences without profiling on a dedicated server, Bubble-up [14] and Cuanta [2] designed the synthetic workloads to understand the degree of interference when co-locating applications. Bubble-up probes the interference by using synthetic workloads and determines whether to allow co-location or not so as to meet the QoS of latency critical applications running on datacenters. POPPA [10] and QualityTime [11] proposed similar runtime approaches to measure performance of each application in co-located environment. The most accurate way of measuring performance for an individual application is to observe its solo execution. They perceived this concept by pausing other co-runners during a small amount of time. Then, the target application can monopolize resources without any interference. Meanwhile, Soares et al. studied the concept of pollute buffer in shared last level caches to prevent filling the shared caches as non-reusable data. It focused on improving the utilization of shared caches through OS-level page allocation [19]. Zhuravlev et al. extended the CPU scheduler to alleviate the degree of interferences in a native system. The goal of this work is to schedule the threads by evenly distributing the load intensity to caches [20]. Blagodurov et al. proposed that the scheduler needs to consider the effects of NUMA [21]. Also, there are many prior studies to solve the contention problems such as shared last level cache and NUMA by scheduling virtual machines [3, 8, 9].

Architectural Supports: There are various approaches mitigating performance interference and guaranteeing fairness in shared caches, memory controller and bandwidth. Nesbit et al. employed the network fair queuing model in the memory scheduler to meet the fairness [22]. Mutlu and Moscibroda focused on DRAM specific architectures such as row buffers and multi banks [23]. They pointed out that modern DRAM controllers only consider maximizing throughputs instead of fairness. To alleviate the problem, they introduced the memory scheduling technique to ensure the fairness between threads. Ebrahimi et al. extended the fairness problem in memory subsystems by including shared last level cache and memory bandwidth [24]. This work focused on the source incurring performance interference and proposed throttling mechanism by controlling injection rates of requests to alleviate the contention of shared resources. Suh et al. firstly discussed the cache partitioning scheme to efficiently use the shared resources [25]. Qureshi et al. proposed utility based cache partitioning technique to achieve high performance [26]. They developed utility monitors to track the efficiency of caches for each application and then decided the degree of cache partitioning to minimize the total cache misses from all applications. Rafique et al. studied the cache and bandwidth managements by cooperating operating system and hardware [27,28]. To prevent the replacement from other applications, the pinning way in cache was introduced [29]. Recently, to minimize the effects of cache pollutions, virtualization-aware prefetching techniques are introduced [30–32].

6. CONCLUSION

In public clouds, the application performance of users can be easily affected by other applications belonging to different users. Nevertheless, public cloud providers do not control the unintended performance degradation. It leads to a biased pricing scenario. This work presented Fair Pricing Runtime, a novel approach to estimate performance degradation of each application in co-located environments and then we reflect the amount of unintended performance degradation on their price. Fair Pricing Runtime has negligible performance overhead and operates without any special hardware or programmer supports. Using this mechanism, we could estimate performance degradation with 4% mean absolute error with a very low overhead of around 1%.

7. REFERENCES

[1] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 237–250, ACM, 2010.

[2] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 22:1–22:14, ACM, 2011.

[3] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2012.

[4] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 281–292, ACM, 2012.

[5] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: Accelerating resource allocation in virtualized environments," *SIGPLAN Not.*, vol. 47, pp. 423–436, Mar. 2012.

[6] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX

ATC'13, (Berkeley, CA, USA), pp. 219–230, USENIX Association, 2013.

[7] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard)," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 131–143, ACM, 2015.

[8] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 325–336, IEEE Press, 2014.

[9] J. Rao, K. Wang, X. Zhou, and C. zhong Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 306–317, Feb 2013.

[10] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on high performance computer systems with node sharing," *Scientific Programming*, vol. 22, no. 2, pp. 59–74, 2014.

[11] A. Gupta, J. Sampson, and M. Taylor, "Quality time: A simple online technique for quantifying multicore execution efficiency," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 169–179, March 2014.

[12] Qumranet, "Kvm: Kernel-based virtualization machine," tech. rep., Qumranet, 2007.

[13] A. C. de Melo, "The new linux perf tools,"

[14] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-44, (New York, NY, USA), pp. 248–259, ACM, 2011. Acceptance Rate: 21

[15] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 607–618, ACM, 2013.

[16] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 89–100, ACM, 2013.

[17] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh, "Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 181–192, ACM, 2013.

[18] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu, "Going vertical in memory management: Handling multiplicity by multi-policy," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 169–180, IEEE Press, 2014.

[19] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 258–269, IEEE Computer Society, 2008.

[20] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 129–142, ACM, 2010.

[21] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2011.

[22] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 208–222, IEEE Computer Society, 2006.

[23] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 146–160, IEEE Computer Society, 2007.

[24] 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 *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 335–346, ACM, 2010.

[25] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, (Washington, DC, USA), pp. 117–, IEEE Computer Society, 2002.

[26] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[27] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, (New York, NY, USA), pp. 2–12, ACM, 2006.

[28] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, (Washington, DC, USA), pp. 245–258, IEEE Computer Society, 2007.

[29] S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Adaptive set pinning: Managing shared caches in chip multiprocessors," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 135–144, ACM, 2008.

[30] D. Daly and H. Cain, "Cache restoration for highly partitioned virtualized systems," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[31] J. Zebchuk, H. Cain, X. Tong, V. Srinivasan, and A. Moshovos, "RECAP: A region-based cure for the common cold (cache)," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[32] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 394–405, IEEE Computer Society, 2014.