PRUNE: Fair Pricing RUNtime Engine in Infrastructure-as-a-Service Clouds via Snapshots

ABSTRACT

Infrastructure-as-a-service (IaaS) clouds primarily use a pricing model that charges users a flat hourly fee for running their applications on shared servers. This pricing model leads to the unfortunate scenario whereby users incur higher fees when their application happens to be scheduled to co-run with a highly contentious application. Addressing this problem to enable **fair pricing** can allow the purveyors of IaaS clouds to develop robust pricing models that charge users a fee congruent to their resource usage rather than as a function of the behavior of other users' applications. However, the key technical hurdle that leaves this problem unsolved is the lack of a **scalable**, **accurate** and **lightweight** technique for live, continuous estimation of the performance degradation caused by co-running virtual machines (VMs) on cloud servers.

To address this problem, this paper introduces the **snapshot** technique, a runtime mechanism for estimating the degradation caused by co-running VMs in public clouds. Snapshot uses strategic phase-triggered millisecond-scale pauses to orchestrate a scalable, precise and low-overhead interference estimation engine that can be continuously deployed in IaaS cloud environments to facilitate fair pricing. We evaluate snapshot for a wide spectrum of workload scenarios and applications, demonstrating that it seamlessly scales up to 16 VMs and can estimate performance degradation to within 4% of ground truth while introducing a negligible performance overhead of less than 1%.

1. INTRODUCTION

Cloud computing has emerged as a key technology in a number of ways over the past few years, evidenced by the fact that 93% of the organizations is either running applications or experimenting with Infrastructure-as-a-Service (IaaS) cloud [1]. With IaaS cloud computing emerging to be the most preferred choice, it becomes increasingly important to leverage its benefits as efficiently as possible.

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 virtual machines) to each customer [2,3]. High resource utilization is achieved by consolidating virtual machines into a single server. However, consolidation of virtual machines (VMs) belonging to different users may lead to performance interference with each other. Although this kind of performance interference cannot be tolerated by users running latency critical applications [4–6], a wide variety of users whose applications are of batch type can utilize such infrastructures keeping in mind its cost effectiveness. Hence to satisfy such users public clouds provide a variety of shared instance types so as to satisfy different requirements from users such as the quantity of virtual CPUs, memory and disk, users cannot specify the requirements for shared architectural resources like last level cache, memory bandwidth and memory controller. This inability is problematic as sharing of architectural resources can significantly affect the performance of each VM [7–15].

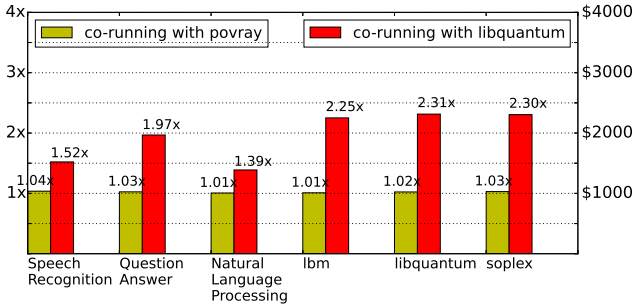


Figure 1: Slowdown in execution time of applications when running with co-runners. Users end up paying more when their applications are co-located with libquantum.

Figure 1 shows performance degradation when the applications on the x-axis from SiriusSuite [16], DjiNN & Tonic [17], and SPEC CPU2006 co-run with `libquantum` and `povray`. In such a scenario, the `Question Answer` application becomes 1.97x slower when co-running with `libquantum` due to contention of shared architectural resources. On the other hand, when co-locating with `povray`, the performance overhead is negligible. From the perspective of users, this slowdown in execution time is not acceptable because IaaS cloud providers calculate the price on an hourly basis based on the amount of time each application takes to execute. This practice has been noted to discourage users and to user who game the system [10].

To enable **fair pricing** on public clouds, we need a methodology to precisely measure performance degradation incurred by co-runners. This necessitates the estimation of solo performance of applications when running alone to figure out the degree of performance degradation due to co-runners. Although there have been prior works on predicting interfer-

ence [18–21], several key open challenges must be addressed to realize a deployable solution:

1. **Scalable** - Since the number of cores is steadily increasing in modern server platforms, cloud providers try to increase their utilization by consolidating a number of virtual machines on a single physical server. Solutions that estimate performance degradation should scale gracefully even as the number of co-runners increases.

2. **Accurate** - These solutions should be accurate in estimating performance degradation, as these performance estimations directly correlate to pricing which customers have to pay.

3. **Lightweight** - Production environments only allow solutions to pose a small amount of overhead. Companies such as Google can tolerate not more than 1% to 2% degradation for supporting runtime profiling technique [22].

In this paper, we propose *snapshot*, a scalable technique that estimates performance degradation during runtime with high accuracy and negligible overhead. The key insight underlying the design of *snapshot* is the fact that execution behaviors of applications do not significantly change within a single phase, and thus making multiple degradation measurements within a single phase serves only to introduce unnecessary overhead. Hence, triggering an interference estimation event on phase and co-phase boundaries can provide a lightweight runtime system with low overhead. As a result, the effective time for which we try to estimate performance degradation is negligible.

The performance degradation estimation technique in snapshot works by pausing a VM's co-runners when the VM changes phases, allowing snapshot to measure the solo performance of the VM. Comparing this solo performance measurement to measurements made once all co-runners have resumed allows degradation to be estimated. Triggering these *interference estimation events* is a carefully designed mechanism that leverages carefully selected performance monitoring units (PMUs) measurements in concert with lightweight processing to avoid detecting *false phase boundaries* – changes in application behavior that are the result of measurement fluctuations or changes in co-runner contentiousness – while detecting *true phase boundaries* – actual changes in application behavior. Thus, interference estimation events occur only when they allow snapshot to refine its estimates of performance degradation. As a result of this design, our technique is resilient to overhead issues that have proved prohibitive for a scalable and deployable design.

To enable *snapshot*, one of the most significant challenges is to precisely, efficiently, and continuously detect not only phase, but also co-phase changes in public clouds. In this paper, we design a solution that leverages PMUs to capture both phase and co-phase changes of applications running on VMs in a highly efficient manner. Since each application has different sensitivities on architectural resources, a single PMU type cannot represent performance behavior across all applications. We identify a mix of PMU types that can differentiate phase changes across applications by using a small training set of 8 workloads to isolate representative PMU types. We perform cross validation on these selected PMU types on a spectrum of application workloads to demonstrate generality. Using snapshot, we are able to continuously monitor the performance implications of inter-VM interference and enable a fair pricing solution on modern commodity servers. The new contributions of this paper are as follows.

- **Snapshot Technique:** We introduce *snapshot*, a scalable, accurate and lightweight mechanism for estimating performance degradation for applications running in public clouds.
- **Phase Triggered Measurement:** We design a novel phase triggered mechanism based on a set of performance monitoring units for identifying execution behaviors of applications and evaluate how the selected PMUs generalize to a variety of applications.
- **PRUNE - Pricing RUNtime Engine:** We describe a runtime system for fair pricing that enables customers to pay as though their application runs in isolation. Since it does not require any modifications of hypervisors or guest operating systems, our runtime system can be deployed in public clouds today.
- **Real-system Evaluation:** We perform a thorough evaluation of our runtime system on real systems for a variety of applications including SiriusSuite [16], DjiNN&Tonic Suite [17] and SPEC CPU2006. In addition, we perform scalability studies to evaluate the effectiveness of proposed runtime for future IaaS clouds.

Using our runtime engine, we are able to precisely estimate the performance degradation in co-located environments with a mean absolute error of around 4% and negligible overhead of less than 1% for a wide spectrum of workload scenarios that are being executed in current day IaaS clouds. We have also evaluated our technique on different microarchitectural platforms to demonstrate its platform independent nature. Compared to prior interference prediction techniques [18, 19], our technique shows up to 4x more accuracy with 5x less overhead, and is deployable in current and future IaaS public clouds.

2. BACKGROUND AND MOTIVATION

2.1 IaaS Public Clouds

Most public clouds take advantage of virtualization technology to provide an isolated computing infrastructure to each customer as well as to improve the utilization of their cloud datacenters. Commercial cloud providers serve a variety of types of virtual machines to satisfy the diverse requirements of users. For each VM allocation, customers choose suitable resource capabilities such as the number of virtual CPUs, the amount of memory and storage size based on their demands. The price range for a given VM allocation differs based on instance types and capabilities.

2.2 Pricing Schemes and Challenges

To charge fees to customers, public clouds such as Amazon EC2 [23] and Google Compute Engine [24] charge an

	System Approaches			Architectural Support	
	PRUNE	Quality Time	Fair Pricing	MISE	STFM
Deployability in public clouds	✓	✓	✓	X	X
Overhead (worst case)	5%	40%	40%	Hardware Overhead	Hardware Overhead
Error (worst case)	10%	45%	45%	20%	83%
Phase analysis	✓	X	X	X	X

Table 1: Comparison to prior work (for 4 cores)

hourly rate based on execution time of the VM. Although the pay-as-you-go scheme is an attractive pricing model for its simplicity, 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 using the shared computing resources. If a virtual machine undergoes performance degradation due to resource contention, the victim ends up paying more due to their increased execution time.

To mitigate the fairness problem, we could consider having strict quality of service (QoS) agreements between service providers and customers. However, it is difficult to define QoS metrics in public clouds because the applications running inside VMs are not visible to the IaaS platform. Even with insights into the nature of broad categories of applications such as web serving or business analytics, applications within the same category often 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, QoS metrics like request per second (RPS) would be difficult to use.

The most accurate way of eliminating biased pricing schemes is to measure performance degradation due to co-running applications during runtime. However, it is challenging in co-located environments like public clouds as architectural resources such as last level cache and memory bandwidth, which are shared among users, make it difficult to estimate performance degradation for each user's VM.

2.3 State-of-the-Art Solution and Its Limitations

There are two classes of prior approaches for estimating the amount of performance degradation in co-located environments [18–21]. These techniques periodically pause all the co-running applications for a short time, allowing one application to continue running. During this pause, the running application monopolizes the computing resources on the system to determine the degree of performance degradation. Although this approach is a straightforward technique, there are three limitations: 1) not scalable, 2) low accuracy, and 3) high overhead. Due to those limitations, prior techniques are not deployable in modern IaaS clouds.

2.4 Related Work

There have been many prior studies to detect performance interference in a variety aspects of architectural resources. We categorize the prior work into two broad types. We look first into the system and OS level approaches and then address the architectural supports for detecting the interference.

System/OS approaches: There are many efforts introducing software frameworks and proposing the new designs of operating systems [7,8,20,25–28]. 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 [7]. To precisely estimate the performance interferences without profiling on a dedicated server, Bubble-up [25] and Cuanta [8] 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 [18] and QualityTime [19] 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 [29]. 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 [30]. Blagodurov et al. proposed that the scheduler needs to consider the effects of NUMA [31]. Also, there are many prior studies to solve the contention problems such as shared last level cache and NUMA by scheduling virtual machines [9, 14, 15].

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 [32]. Mutlu and Moscibroda focused on DRAM specific architectures such as row buffers and multi banks [33]. 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 [34]. 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. Subramanian et al. further investigated the shared memory subsystem in terms of fairness [35]. Suh et al. firstly discussed the cache partitioning scheme to efficiently use the shared resources [36]. Qureshi et al. proposed utility based cache partitioning technique to achieve high performance [37]. 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 [38, 39]. To prevent the replacement from other applications, pinning the way of cache was introduced [40]. Traditionally, there are many prior studies in terms of detecting phase changes. These works proposed architectural supports for efficiently detecting phase changes on CMP and SMT [41–48]. Recently, to minimize the effects of cache pollutions, virtualization-aware prefetching techniques are introduced [49–51].

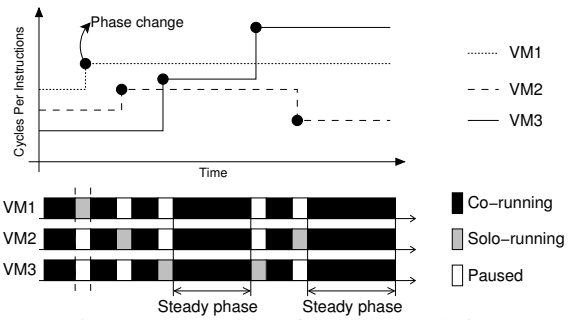


Figure 2: Overview of snapshot technique

3. PRUNE: PRICING RUNTIME ENGINE

In this section we propose **P**ricing **RUN**time **E**ngine (PRUNE), an online runtime engine for enabling fair pricing in public clouds. PRUNE is based on *snapshot* technique on top of the phase triggered mechanism. *Snapshot* is an efficient and scalable technique to precisely detect, quantify, and account for performance degradation in public clouds. PRUNE performs a snapshot of the system during whenever an application's phase changes.

To enable *snapshot*, we propose a phase triggered mechanism based on Performance Monitoring Units (PMUs). We provide a robust solution towards detecting runtime phase changes at co-located environments using PMUs.

3.1 Snapshot Technique

Our goal in the design of PRUNE is to achieve high accuracy with low overhead in estimating unintended performance degradation on public clouds. A key challenge towards accomplishing this is to estimate the solo execution performance of the application even during the presence of co-runners. To achieve this, our *snapshot* technique selectively pauses all the co-running VMs for a sufficient amount of time. During this time, we allow the un-paused VM to monopolize computing resources present in the system by taking a *snapshot* of the system. In other words, we measure the performance of the application during those pause periods. We use the aggregated value of these measurements to estimate the solo performance of that particular of the application.

An important challenge towards realizing this technique is to identify the right situations to perform snapshots. This is because excessive/unnecessary pausing of co-runners might cause drastic overhead problems. These right situations are identified as phase changes by us based on the following observations. Firstly, the execution behavior of applications does not drastically change within a single phase. It means that we need to estimate performance degradation once during every phase. On the other hand, most of the applications that we measure show that the number of phase changes are not significant [cite][cite]. It gives us an opportunity to optimize our snapshot technique for common cases where applications have few phase changes. Therefore, we can estimate performance degradation with negligible overheads.

In order to measure runtime performance degradation of the application we need to know the performance of the application during co-location $CPI_{(co-location)}$ as well as the performance of the application when it is running alone $CPI_{(solo)}$. Using these quantities, performance degradation of the applications can be easily identified using the following equation.

$$PerformanceDegradation = \frac{CPI_{(co-location)}}{CPI_{(solo)}} \quad (1)$$

$CPI_{(co-location)}$ is the CPI of the application when the co-runners are running. $CPI_{(co-location)}$ is directly measured when the application is running along with the co-runners. On the other hand we utilize the performance measurements of the application that we had obtained during the pause periods to estimate the solo execution performance $CPI_{(solo)}$. To be more precise, we aggregate the performance estimation of every individual phase of the VM to calculate the performance degradation for the entire execution of the application as shown by Equation 2.

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

where,

- $CPI_{(solo)}$ 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 phases in the application.

Figure 2 shows how our *snapshot* technique estimates performance degradation. We can see from Figure 2 that whenever there is a phase change, the co-running VMs are paused by our runtime system to measure solo performance execution.

To make *snapshot* technique viable, accurately triggering the snapshots during phase changes is very important. We discuss this in the next section.

3.2 Phase Triggered Mechanism

The key challenge of phase triggered mechanism is to precisely identify changes. A phase is defined as an interval within a program's execution with similar behavior [52]. Changes in program behavior is referred to as a endogenous phase change which can be seen by drastic increase or decrease in the Cycles Per Instruction (CPI) of an application. This drastic variation of CPI is clearly visible when an application is running alone making endogenous

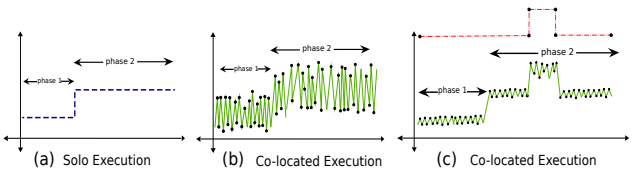


Figure 3: Exogenous phases. (a) Solo Execution of application. (b) Fluctuations in PMU type during co-location. (c) Co-phase interference during co-location

phase change detection very easy. However, public clouds which allow co-location of multiple VMs makes it difficult to identify endogenous phase changes. In the presence of co-runners, there are a number of unnecessary phase changes being detected due to fluctuations in PMU measurements or phase changes in co-running applications. These unnecessary phases referred to as exogenous phase changes do not reflect endogenous phase changes. Detecting phase changes during such situations will result in unnecessary pausing of co-runners leading to high overhead. In the next section, we try to investigate the the situations during which exogenous phase changes occur so as to mitigate its counter effects .

3.2.1 Exogenous phase changes - Scenarios

- **Continious Fluctuation of PMU measurements** -

  Streaming PMU measurements belonging to a particular phase lie within a certain range of values. Whenever there is a phase change, incoming PMU measurements would lie within a different range of values. This is a common case during the solo execution of application. On the other hand, in the presence of contentious co-runners, fluctuation of incoming PMU measurements makes it difficult for us to determine if they belong to the same phase or a new phase. This fluctuation is commonly observed for PMU types like Last Level Cache, CPI etc. This is because, during the presence of contentious co-runners, there is excessive thrashing of these resources from time to time which results in such fluctuations. This phenomenon is illustrated in Figure 3. Figure 3 (a) shows the phase of an application when it is running alone. Figure 3 (b) shows the PMU measurements when the application is running along with a co-runner. We can clearly see in Figure 3 (b) that some PMU measurements from phase 1 lie closer to the range of PMU measurements in phase 2 and vice versa. This is a common case with when the PMU measurements are of type Last Level Cache which is shared between multiple applications.

- **Co-phase interference** - Co-running applications may change phases. This would interfere with the phase of the observed application resulting to an phase change due to the change in range of incoming PMU measurements though there isn't a endogenous phase change actually. This phenomenon is called co-phase interference. In this situation, we need PMU measurements that can distinguish between endogenous phase changes and co-phase interference. From Figure 3 (c) we see that the the PMU measurements showing co-phase variation is as equally significant compared to the endogenous phase changes. This can result in triggering unnecessary additional phase changes.

From this, it is clear that detecting endogenous phase changes during co-located environments is challenging. Frequently triggering a phase change will result in increasing the overhead due to pausing of co-runners by our snapshot technique. Hence we need to design the runtime system to accurately trigger snapshots during endogenous phase changes simultaneously minimizing triggering of unnecessary phase changes.

3.2.2 Finding Representative Types of PMU for Phase Detection

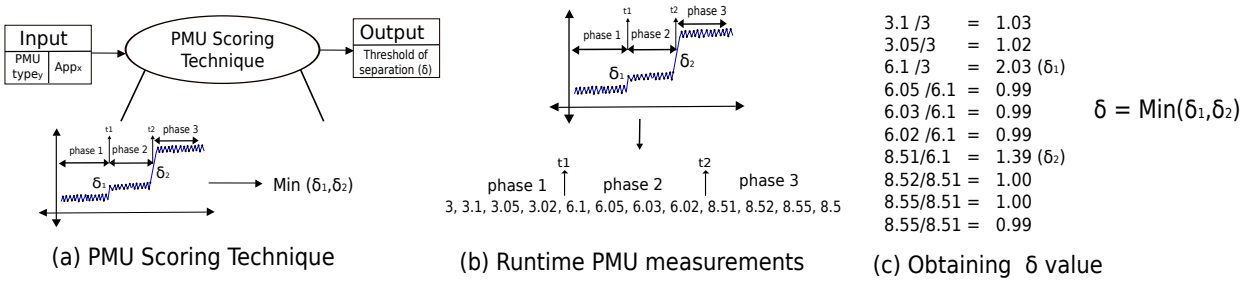
In order to trigger snapshots during endogenous phase changes we need a technique that could simultaneously distinguish phase changes across a variety of applications and discard fluctuations and co-phase interference. We investigated the runtime information provided by PMUs to achieve this goal. The key challenge here is to identify the right PMU type to perform phase detection across a wide variety of applications. This is because our experiments show that some PMU types are much more effective in discarding unnecessary phase changes at co-located environments. Hence we investigate each and every available PMU type for gauging its effectiveness towards achieving the above mentioned goal.

To utilize PMUs for capturing phase changes across a wide variety of applications, we initially take a training set of applications. We try to tune our parameters based on which phase changes are clearly identified for the training set. Later in section 5, we use the same parameters to cross validate our experiments on a set of different benchmarks under different co-running applications. We carefully choose our training set of applications to cover a wide range of contentiousness, sensitivity and phase changing attributes [53]. The list of training applications are shown in Table 2. We use `astar` as our training co-runner. `Astar` is know to be both contentious as well as phasy. This can train our model to be resistant against both fluctuations as well as co-phase interference.

- **1 - Solo profiling** We first profile the solo execution of the training set of applications to obtain its CPI over the number of retired instruction (time). From this we identify the endogenous phase changes present in the application. We obtain a vector of timestamps where each entry denotes the timestamp at which a phase change occurs in an application.

- **2 - Colo profiling** As a first step towards identifying useful PMU types, we profile the training application to obtain PMU measurements with a training co-runner. We then observe this profile to identify if phases are being detected appropriately at the timestamps which are obtained from solo profiling.

- **3 - Qunantifying the effectiveness of PMU type** The end goal of this process is to obtain the best PMU type to detect endogenous phase changes across a varied class of applications. To enable that we should quantify the effectiveness of individual PMU types for every single application. This is done by the PMU scoring technique as described in the Section 3.2.2.



(a) PMU Scoring Technique (b) Runtime PMU measurements (c) Obtaining δ value

Figure 4: Overview of trigger scoring technique

Application	PMU rank		
	1st	2nd	3rd
astar	**CPI**	branch	L1-D load miss
bzip2	**LLC store miss**	CPI	L1-D load miss
cactusADM	**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
xalancbmk	**LLC store miss**	LLC load	L1-D load
tonto	**L1-D load miss**	branch	CPI

Table 2: PMU types ordered by their effectiveness

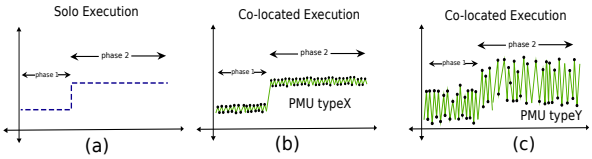


Figure 6: Resistance to fluctuations by PMU types

- **4 - Ranking PMU types** Lastly we rank the PMU types for each application from the quantity obtained from the previous step. From observing the best PMU type for every single application, we try to obtain a single PMU type that can be utilized to detect phase changes across a varied class of application.

3.2.3 *PMU scoring technique*

In order to identify the best PMU types for every single application, we quantify the effectiveness of each PMU type in detecting endogenous phase changes across every single applications present in the training set.

[Input and Output] The input to trigger scoring technique is a an application and its corresponding time varying PMU measurements during co-location for a particular PMU type. The output given by the trigger scoring technique is called the threshold of separation (δ) which quantifies the effectiveness of a particular PMU type.

[Objective function] The objective function of the trigger scoring technique is to quantify the effectiveness of PMU measurements of a particular PMU type to detect all inherent phases present in the solo execution of application and to minimize triggering unnecessary phases. This can be done by identifying appropriate PMU types which show resistance to fluctuations and co-phase interference. For example, from looking at Figure 6 we are clearly able to see that compared to the PMU type shown in Figure 6 (c) `[PMU typeY]`, the PMU type shown in Figure 6 (b) `[PMU typeX]` is much more tolerant to fluctuations in PMU measurements. PMU scoring technique tries to quantify this observation so as to rank the effectiveness of PMU types in detecting phase changes in application. This will enable us to choose a common PMU types to detect phase changes across applications.

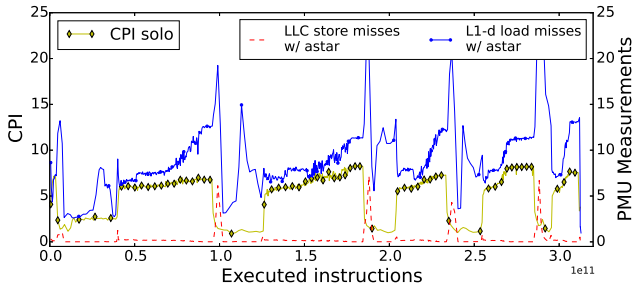
[Threshold of separation] The PMU scoring technique quantifies the effectiveness of PMU type using a value called the threshold of separation (δ). This value is exclusive to every PMU type, application pair and is obtained for all combinations of PMU type, application pair. This value should be high enough to discard variations due to fluctuations in PMU measurements and co-phase interference but low enough to capture the magnitude of intra application phase change.

[Method to obtain threshold of separation (δ)] In order to obtain the effectiveness of a PMU type, we first profile the application to obtain PMU measurements during co-location. We then quantitatively measure the magnitude of variation potrayed by the PMU measurements at every single phase boundary that is obtained from the solo execution profile. This can be quantitatively measured by observing the variation of incoming PMU measurements at phase boundaries. Figure 4 clearly illustrates this methodology to obtain the value of threshold of separation (δ). At every phase boundary, we determine the magnitude at which streaming PMU measurements should be varied so as to identify them. From the example given at Figure 4 (b) and 4 (c) we can see that at timestamp t1 which is a phase change, the PMU measurement change from 3.02 to 6.1. The magnitude of change at this timestamp is is δ_1 which is shown in Figure 4 (c). Similarly we obtain all the δ_i values at phase change happening at every timestamp t_i. The final score of the PMU is the minimum of all the obtained δ values.

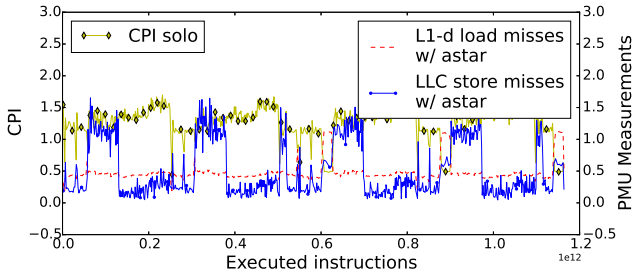
3.2.4 *Ranking PMU types*

The motivation behind ranking PMU types is to obtain a common set of PMU types that can detect runtime phase changes across a varied class of applications. In that context, we need to choose PMU types that depict phase changes much more significantly compared to fluctuations or co-phase interference. To fulfill that, it is clear that PMU types that possess high values of (δ) can be more effective. Hence, we rank PMU types for an application based on this metric as shown in Table 2.

A counter intuitive observation from our experiments as depicted in Table 2 show that there is no single PMU type that can detect phase changes across all the training set of applications. In another words, there can be a situation where



(a) `mcf`



(b) `milc`

Figure 5: Phase changes triggered by PMU types when running with `astar`. Single PMU type is insufficient

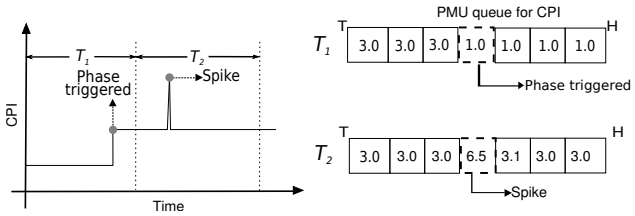


Figure 7: Differentiating spikes and phases

the same architectural resource would fail to capture the phase boundaries across a different applications. This is attributed by the fact that the nature of applications running in public clouds are varied extensively.

We use Figure 5 to illustrate an example scenario where a single PMU type is not able to identify phase boundaries across two different applications. Figure 5 shows that each application requires different types of PMU to precisely detect phase changes. The x-axis indicates the cumulative number of instructions executed as time progresses. The left y-axis and yellow (diamond) line show CPI of the applications when running alone and the right y-axis and the blue (circle) / red (dashed) line show estimating phase changes by a selected 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 5b whereas the same type of PMU is unable to detect inherent phase changes of `mcf` as shown in Figure 5a. These results motivate us to need multiple types of PMU to capture phase changes across applications. To achieve this we undertake an approach where we observe a set of architectural resources in contrast to a single resource to identify inherent phase changes across multiple applications.

3.2.5 Implementation issues - Eliminating Spikes

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 inherent phase changes, we employ a binary classification technique. We maintain a queue per each PMU type. Each queue contains k latest values that have been measured by the PMU type. 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. This parameter is also cross validated. 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 another words, we replace the comparison of a single value as shown in the previous section to a queue of values. In this way, we are able to eliminate incorrectly detected phase changes due to spikes.

Figure 7 shows an example differentiating spikes and phase changes. During T_1 in Figure 7, 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.

3.3 Putting All Together

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

Figure 8 describes the process of the phase triggered taking place in our pricing runtime engine. Our runtime system collects three types of events every second as shown in Figure 8(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 8(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

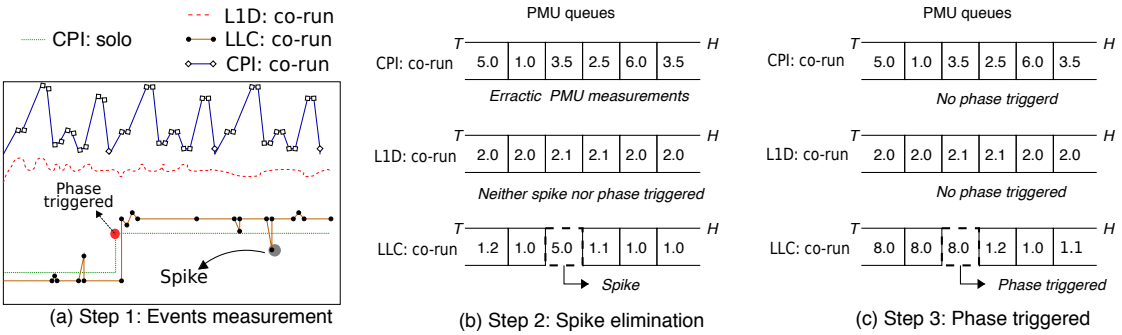


Figure 8: Process of detecting phase changes

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 8(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.

3.4 Pricing for Fairness

In this section, we describe how the proposed *snapshot* technique can be used 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.

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

where,

- P_i: price paid by user i.
- $PerfDeg_i$: degradation suffered by user i from equation 1.

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 algorithm 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.

Algorithm 1 PRUNE: Pricing RUNtime Engine

$perfScoreVM_i$ = <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:** Triggering 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

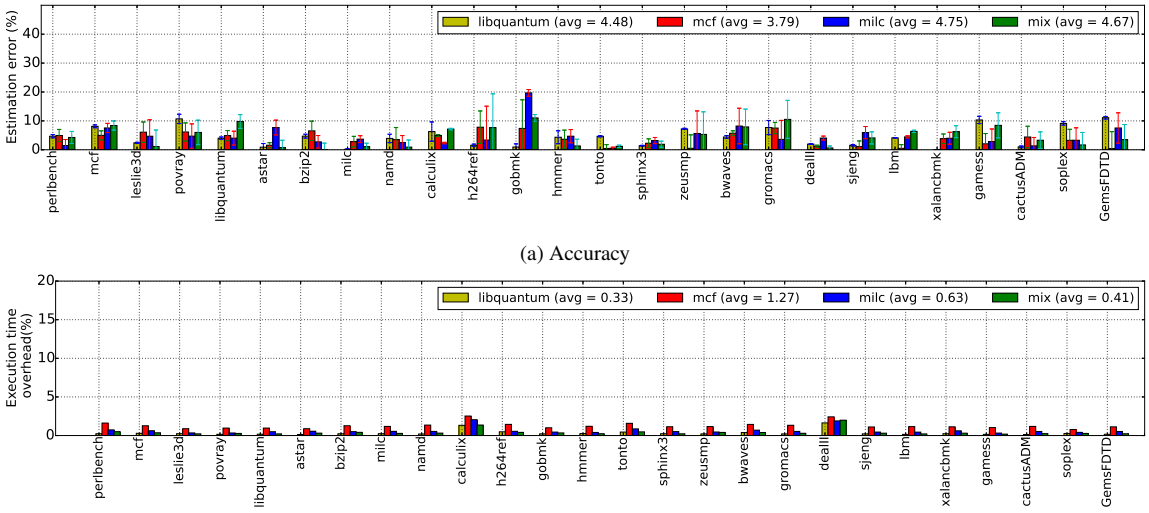
/* **Step 3:** Calculating price based on estimated degradation */
for each VM_i in 1 ... n **do**
  if check_perfDeg_vm(VM_i) == true **then**
    reflect the unintended performance degradation in its bill
  end if
end for

4. EXPERIMENTAL ENVIRONMENTS

In this section, we present the experimental platform in which we evaluate our *snapshot* technique. We also enumerate the benchmarks that used in this paper.

4.1 Experimental Setup

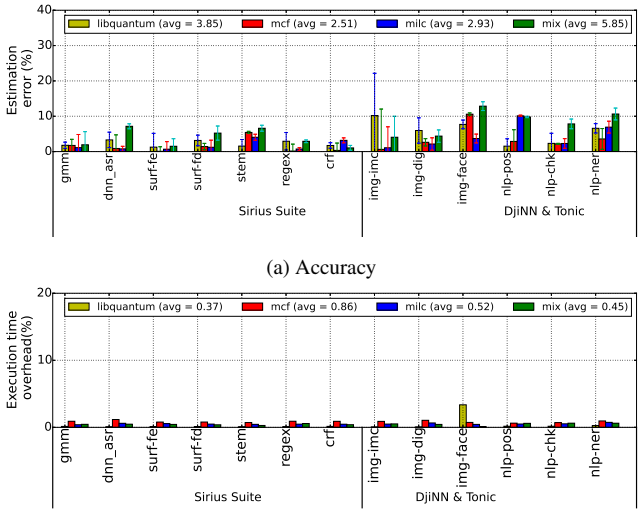
We use Intel Xeon E5-2407 v2 (4-cores) and E5-2630 v3 (16-cores), respectively. To mimic IaaS public clouds, we take advantage of Linux KVM as the hypervisor and run applications on virtual machines [54]. Each virtual machine



(a) Accuracy

(b) Overhead

Figure 10: Accuracy and overhead of SPEC CPU 2006



(a) Accuracy

(b) Overhead

Figure 9: Accuracy and overhead of SiriusSuite and DjiNN&Tonic Suite

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 [**?**].

4.2 Benchmark

To evaluate the effectiveness of our technique, we use the latest cloud datacenter applications from *SiriusSuite* [16] and *DjiNN&Tonic suite* [17]. Recent trends have also shown the demand for intelligent personal assistants such as Apple's Siri, Google's Google Now, Microsoft's Cortana, and Amazon's Echo. There is a promising direction executing such applications in public clouds. In this context, SiriusSuite and DjiNN & Tonic suite contain a class of applications which implement speech recognition, image matching, natural language processing, and question answering systems. We also evaluate *SPEC 2006* with `ref` inputs. SPEC 2006 consists of a class of scientific computing application which are standard CPU benchmarks. With the advent of high performance computing(HPC) in clouds, a class of academic research have started using public clouds for running their applications. Table 3 shows the class of benchmarks and its use cases in Amazon Web Service.

Benchmark	Class of applications	AWS use cases [55]
Sirius Suite	Machine learning	NTT Docomo (voice recognition) [56]
DjiNN & Tonic	Deep neural network	PIXNET (facial recognition) [57]
SPEC 2006	General purpose & Scientific	Penn State [58]

Table 3: Benchmark descriptions

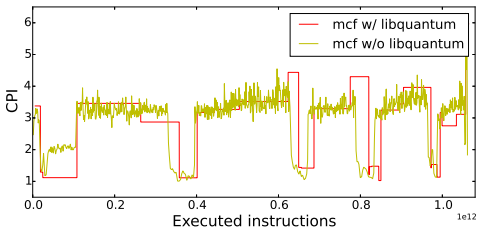
5. EVALUATION

In this section, we evaluate the our *snapshot* technique. We discuss the accuracy in estimating performance degradation and its overhead. We also test scalability by consolidating 16VMs on a single server and analyze detailed phase level behaviors. Finally, we compare our technique with prior work.

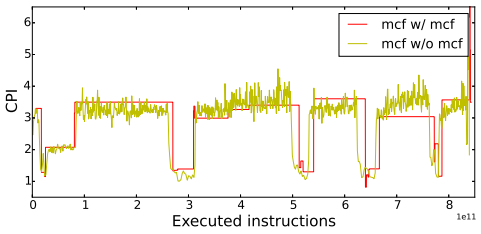
5.1 Analysis of Phase Level Behaviors

Emerging Web Service Applications Today, speech, vision and machine learning based web services have emerged in public clouds [55–58]. To evaluate our snapshot, we leverage the emerging applications, SiriusSuite [16] and DjiNN&Tonic Suite [17]. Firstly, we look into the accuracy of these web service applications. Then we look at the overhead of our runtime system.

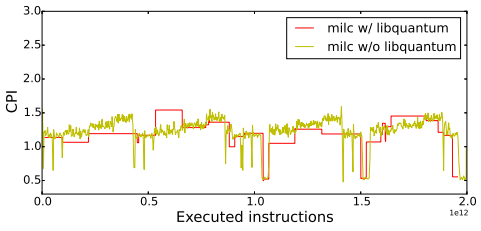
Figure 9a shows the accuracy of *snapshot* technique when four virtual machines are running on a single server. The x-axis shows the benchmarks from SiriusSuite and DjiNN&Tonic Suite and the y-axis presents the error in estimating performance degradation. Each bar indicates different co-runners. First three bars represent a single type of co-runners `libquantum`, `mcf` and `milc`, respectively. The last bar indicates a multiple type of co-runners ,`libquantum`, `mcf`, and `milc`, running on three virtual machines, respectively. From many prior studies, these applications are well known to highly incur resource contention in shared architectural resources.



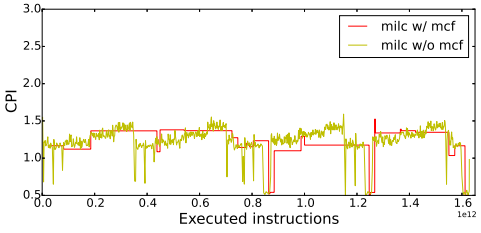
(a) `mcf` (error 0.51%)



(b) `mcf` (error 3.41%)



(c) `milc` (error 1.23%)



(d) `milc` (error 2.52%)

Figure 11: Phase level behavior of snapshot for `mcf` and `milc` when running with 3 `libquantum` co-runners. Snapshots are triggered effectively at phase boundaries

We calculate the error by comparing the estimated performance degradation from our runtime engine with the actual performance degradation. We run each benchmark three times and take the mean to minimize run to run variability. Even running with `libquantum`, our snapshot shows very low error rates across all applications.

Next, we dicuss the overhead of proposed technique. For a deployable design, we have to achieve low overhead to minimize the interference into the applications running on production system. Figure 9b indicates the overhead estimating performance degradation. Like Figure 9a, each bar means the different co-runners. The y-axis presents the execution time overhead due to snapshot. We can see that our technique incurs extremely low overhead in many cases. On average, the overhead is only 0.34%. Even if we colocate applications with `libquantum`, it shows the resilient behavior. Again, our snapshot technique works based on the phase triggered mechanism. We could minimize a number of unnecessary estimation for identifying performance degradation.

General Purpose and Scientific Applications In addition to the emerging applications, we also evaluate SPEC CPU 2006 applications. SPEC CPU consists of general purpose and scientific applications. In public clouds, these types of applications are also running and can be co-located. Figure 10a and 10b show the accuracy and the overhead when co-locating a single type of co-runners and a multiple type of co-runners. On SPEC applications, we can see the similar trend with SiriusSuite and DjiNN&Tonic Suite. Our technique shows low error rates in most applications and the overhead is also negligible.

5.2 Analysis of Phase Level Behaviors

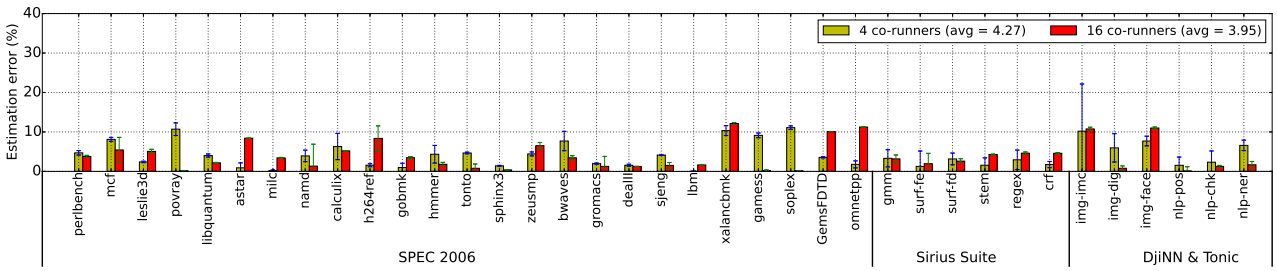
In earlier section, we can see the effectiveness and efficiency of our snapshot. In this section, we look into how the technique can achieve high accuracy as well as low overhead by analyzing the phase level behavior for a selected set of applications. We select two applications, `mcf` and `milc` to analyze the execution behaviors. These applications show a number of phases among SPEC. As co-runners, we use `libquantum` and `mcf`. Figure 11a and 11b show the execution behavior of `mcf` as time goes by. In each graph, the yellow line depicts the phase of CPI when running without any co-runner and the red line shows how the snapshot technique estimates the phase of CPI when running with 3 instances of `libquantum` and `mcf`, respectively. We can see that our snapshot can effectively trace the phase changes even if we change co-runners. The closer the red line is to the yellow line, the lesser the error. Each error rate is 0.51% and 3.41%, respectively. For `milc`, Figure 11c and 11d present that our technique can effectively trace its phase changes on both cases when running with `libquantum` and `mcf`. The error rates are 1.23% and 2.52%, respectively.

5.3 Scalability Study

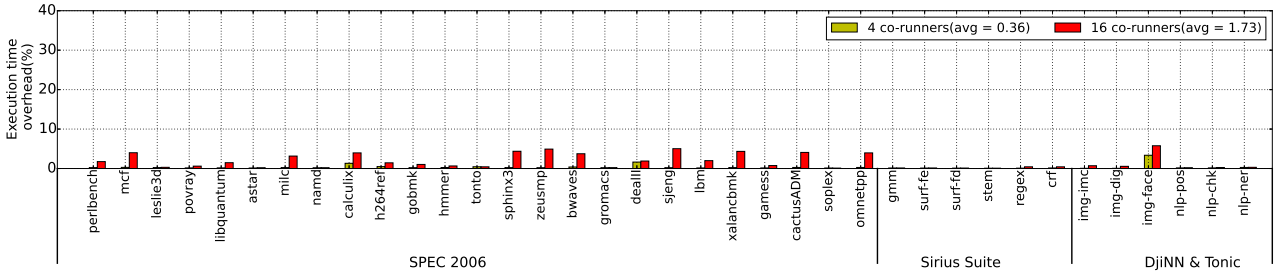
To see the effectiveness of our technique in terms of scalability, we evaluate the accuracy and overhead by increasing the number of co-runners. Figure 12a and 12b show the trend of accuracy and overhead when increasing the number of VMs from 4 to 16. Left bar indicates four virtual machines running on Intel Xeon E5-2407 v2 (4-cores)and right bar means sixteen virtual machines running on Intel Xeon E5-2530 v3(16-cores). Like Figure 13, the accuracy and overhead are not significantly increased on 16VMs. We can see that the accuracy and overhead are not highly affected by the number of co-runners. As we discussed earlier section, we need enough time for measuring solo performance without any interference during a snapshot. Our snapshot technique works based on the phase triggered mechanism. We do not have to measure the performance degradation within a single phase. It gives an opportunity to minimize the overhead of our technique. By reducing the number of measurements, it enables our snapshot to spend more time measuring solo performance when a phase change occurs. It results in a high accuracy in measuring solo performance.

5.4 Comparison of Prior Work and Snapshot

In this section, we compare our snapshot with prior tech-

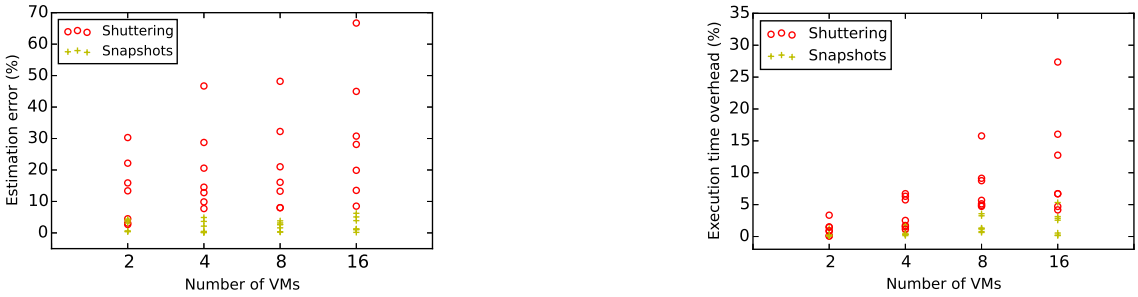


(a) Accuracy



(b) Overhead

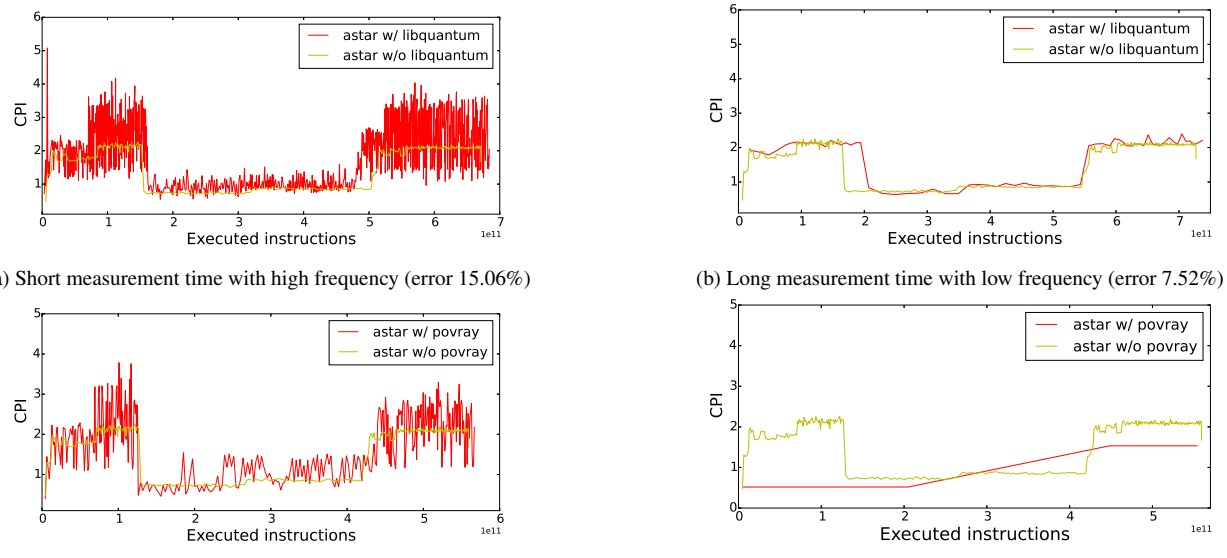
Figure 12: 4VM vs. 16 VMs for accuray and overhead on snapshot. Accuracy is unaffected and overhead is slightly affected when we execute even 16 applications at the same time



(a) Accuracy

(b) Overhead

Figure 13: Accuracy and overhead of snapshot when increasing the number of VMs. Shuttering suffers heavily as the co-runners scales up.



(a) Short measurement time with high frequency (error 15.06%)

(b) Long measurement time with low frequency (error 7.52%)

(c) Short measurement time with high frequency (error 5.2%)

(d) Long measurement time with low frequency (error 28.03%)

Figure 14: Phase level behavior of shuttering for `astar` when running with `libquantum` and `povray`

nique called snapshot [18]. Figure 15 shows the accuracy and overhead of shuttering and *snapshot* techniques when running with fifteen instances of `libquantum` as co-runners. The x-axis shows the benchmarks and the y-axis presents the error in estimating performance degradation. Through these experiments, we can see that the estimation error is much lower for *snapshot* technique than shuttering. The mean error of shuttering is 12.8%. On the other hand, our *snapshotting* technique shows the low error rates, 3.95%. For the overhead, the average is 9.23% on the shuttering technique, but our technique shows only 1.73% of execution time overhead. The reason why our technique shows high accuracy and low overhead is that our technique does not have to periodically measure the performance interference. Instead, our snapshot is only triggered when a phase change occurs.

To compare scalability between shuttering and our snapshot, we evaluate the accuracy and overhead for a set of applications by increasing the number of co-runners from 2 to 16VMs. Figure 13a and 13b show the accuracy and overhead for a set of applications, respectively. By consolidating more virtual machines on a single server, the number of phase changes increases dynamically. Nevertheless, our snapshot technique shows scalable performance in terms of overhead as well as accuracy.

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] Forbes, "http://www.forbes.com/sites/benkepes/2015/03/04/new-stats-from-the-state-of-cloud-report/."

[2] Wikipedia, "Amazon Elastic Compute Cloud." `https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud`. Accessed: 2015-08-10.

[3] Wikipedia, "Google Compute Engine." `https://en.wikipedia.org/wiki/Google_Compute_Engine`. Accessed: 2015-08-10.

[4] Y. Zhang, Z. Zheng, and M. Lyu, "Exploring latent features for memory-based qos prediction in cloud computing," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pp. 1–10, Oct 2011.

[5] "Private vs public clouds which one to choose." `http://www.logicworks.net/blog/2015/03/difference-private-public-hybrid-cloud-comparison/`. Accessed: 2015.

[6] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing - the business perspective," *Decis. Support Syst.*, vol. 51, pp. 176–189, Apr. 2011.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 223–238, ACM, 2015.

[17] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 27–40, ACM, 2015.

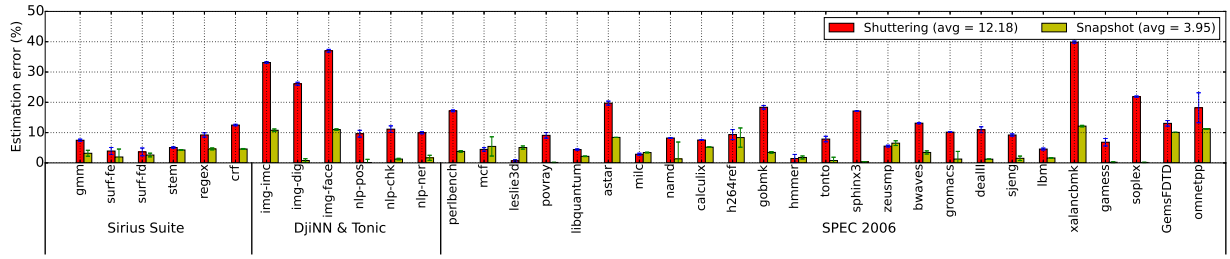
[18] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on high performance computer systems with node sharing," *Scientific Programming*, vol. 22, no. 2, pp. 59–74, 2014.

[19] 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.

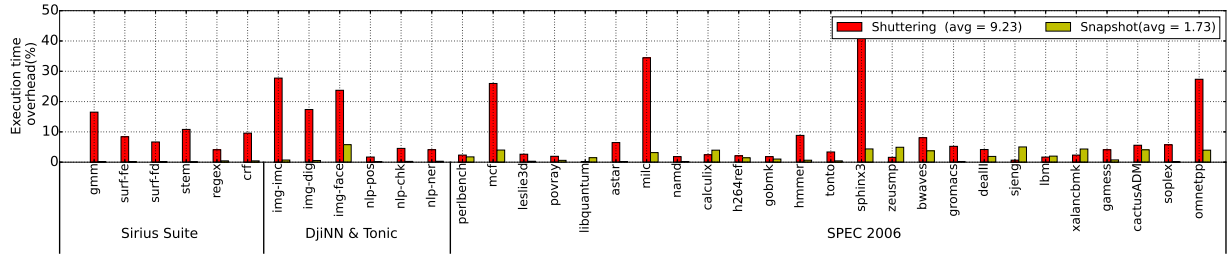
[20] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY,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

[21] 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.

[22] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for



(a) Accuracy



(b) Overheads

Figure 15: Shuttering vs. snapshot for accuracy and overhead on 16VMs

data centers," *IEEE micro*, no. 4, pp. 65–79, 2010.

[23] A. AWS, "Amazon EC2 Pricing." `https://aws.amazon.com/ec2/pricing/`. Accessed: 2015-08-12.

[24] G. C. Platform, "Google Compute Engine Pricing." `https://cloud.google.com/compute/pricing`. Accessed: 2015-08-12.

[25] 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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "Mise: Providing performance predictability and improving fairness in shared main memory systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 639–650, Feb 2013.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 217–, IEEE Computer Society, 2003.

[42] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Cpu accounting in cmp processors," *IEEE Comput. Archit. Lett.*, vol. 8, pp. 17–20, Jan. 2009.

[43] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Itca: Inter-task conflict-aware cpu accounting for cmps," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), pp. 203–213, IEEE Computer Society, 2009.

[44] C. Luque, M. Moreto, F. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Cpu accounting for multicore processors," *Computers, IEEE Transactions on*, vol. 61, pp. 251–264, Feb 2012.

[45] C. Luque, M. Moreto, F. J. Cazorla, and M. Valero, "Fair cpu time accounting in cmp+smt processors," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 50:1–50:25, Jan. 2013.

[46] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, (New York, NY, USA), pp. 336–349, ACM, 2003.

[47] J. Lau, S. Schoenmackers, and B. Calder, "Transition phase classification and prediction," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, (Washington, DC, USA), pp. 278–289, IEEE Computer Society, 2005.

[48] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in smt processors," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), pp. 133–144, ACM, 2009.

[49] 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.

[50] 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.

[51] 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.

[52] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *J. Instruction-Level Parallelism*, vol. 7, 2005.

[53] L. Tang, J. Mars, and M. L. Soffa, "Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, (New York, NY, USA), pp. 12–21, ACM, 2011.

[54] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, (Ottawa, Ontario, Canada), pp. 225–230, June 2007.

[55] Amazon, "Amazon web services User Case Studies." `https://aws.amazon.com/solutions/case-studies/`. Accessed: 2015-08-12.

[56] Amazon, "AWS Case Study: NTT Docomo." `http://aws.amazon.com/solutions/case-studies/ntt-docomo/`. Accessed: 2015-08-12.

[57] Amazon, "AWS Case Study: PIXNET." `http://aws.amazon.com/solutions/case-studies/pixnet/`. Accessed: 2015-08-12.

[58] Amazon, "AWS Case Study: Penn State." `http://aws.amazon.com/solutions/case-studies/penn-state/`. Accessed: 2015-08-12.