Snapshot Shuttering for Fair Pricing in Data Centers

Abstract

Infrastructure as a service (IaaS) Cloud computing has recently emerged as a new paradigm for cost effectiveness that allows users to pay a flat hourly rate to run their applications on a virtual machine (VM) running on a server providing access to some combination of CPUs, memory and network resources. However, co-located workloads running on different user's VM's on the same physical node may interfere with one another. This situation might leads to an unfair scenario causing degradation to performance of the individual applications. The magnitude of degradation is highly variable as it depends upon the nature of the co-located application. Since, there is no control over the nature of co-runner running in the same node, such a situation can lead to a biased pricing scenario where a user might end up paying more for running his application if he is co-located with a contentious co-runner.

To address this limitation, we present Fair Pricing, a dynamic, lightweight and robust runtime engine that accurately predicts the degradation due to co-location and charges users in a fair manner. Fair Pricing employs a methodology called Snapshot Shuttering which examines the runtime phases of applications with the help of multiple hardware Performance Monitoring Units (PMUs) and attempts to detect a phase change. It then uses this information to perform snapshot shuttering — a technique to accurately deduce the interference due to co-runners. In this work, we show that we can accurately identify phase changes and predict inter application interference within 4% mean absolute error on SPEC benchmarks with a very low overhead around 1% when there exist up to 4 simultaneously running applications.

1. Introduction

Low resource utilization in modern day data centers have become a primary concern [1]. Though there is significant potential for parallel execution of multiple applications across different cores, within a single node, most of them remain unutilized. Much of this is due to the degradation in performance of the induvidual applications as a result of sharing of resources like L3 cache and memory bandwidth [2] [3] [4] [5] [6].

To tackle this situation, modern day Infrastructure as a Service (IaaS) cloud computing providers have separate billing schemes based on whether to allow co-location or not. For example amazon EC2 has On Demand instances and Reserved Instances which allow co-location of applications belonging to other users unlike Dedicated Instances which disallow co-location [7]. This lets the user choose schemes that are suitable for them based on their time and budget constraints [8] [9] [10].

Although there seems to be advantages associated with these schemes, they gives rise to a new set of challenges one of which is fairness in pricing. Under such schemes, the nature of the co-runners are unknown due to the highly variable magnitude of performance degradation. In some cases, a cache contentious co-runner causes greater degradation to an application than a compute intensive co-runner would. In general the magnitude of performance degradation increases as the number of co-runners running on the same node increase. Current day pricing schemes in data centers do not take into account these factors. Such a situation might lead to a scenario in which the execution time of a particular application can vary significantly when run at different times under the same infrastructural setup. Since pricing in IaaS cloud is directly proportional to the execution time, this variability may incur drastic unwarranted penalties.

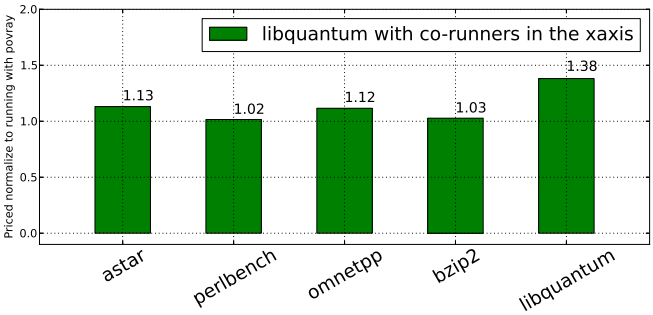

Figure 1: Price to be paid when running libquantum when co-located with 3 instances of the co-runners in the xaxis normalized to the price to be payed when running libquantum when co-located with 3 instances of povray

Figure 1 shows the degradation due to co-location caused when a user is running libquantum, a library for quantum computer simulation while co-locating 3 instances of the applications present in the xaxis. Under the current scheme a user running libquantum would end up paying 39% more when co-located with 3 more instances of libquantum as co-runner than when co-located with 3 instances of povray as a co-runner. A possible remedy to this solution is to be able to accurately predict the degradation due to the interference caused by the co runners and to discount the applications accordingly. Since the vendors already benefit from the increased throughput per machine due to co location, these discounts can be viewed as passing over the saving from co-locations back to user whose Quality of Service has been degraded as a result [11] [12] [13].

Accurately predicting degradation due to co-location with a minimum overhead possess a significant challenge [14]. Move over, the impact on shared resources as the number of co-runners increase raises the difficulty in predicting the degradation due to co-location as shown in the Figure 2.

There are a number of prior work that try to address this problem. Bubble Up [15] uses static techniques to predict the degradation due to co-location. While these might be suitable for applications whose nature is well defined, in scenarios like Amazon EC2 where you would be unaware of the nature of

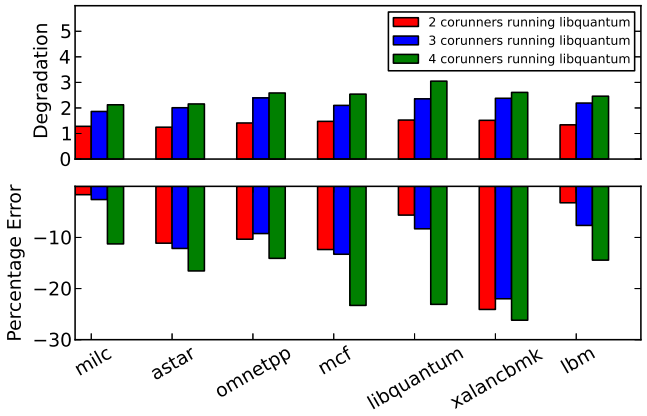

Figure 2: Increase in degradation due to co-location with increase in the number of co-runners(top). Increase in predicting degradation due to co-location by prior work with increase in the number of co-runners (bottom).

applications running, usage of static techniques might not be an effective solution. Prior work targeted at enabling fair pricing HPC [16] systems uses a technique called precise shuttering. One of the issues that it failed to address is the effectiveness of their solution when the number of co-runners increase. Hence, we need a runtime which is accurate, dynamic, lightweight and robust even when number of co-runners scale up.

To address these three challenges, we propose a novel Fair Pricing Runtime Engine. It is a dynamic, lightweight approach which at runtime effectively detects an application's degradation due to co-runners with high accuracy and low overhead. We have designed a deployable lightweight software solution which can work well across multiple architectures which does not require changes to the current production system software in deployment.

The key techniques that we have used in this paper is called Snapshot Shuttering. This consists for two parts :- phase detection and ground truth estimation. It examines the runtime phases of the application portrayed by multiple performance counter types to detect phase changes [17] [18] [19]. Once a change in the runtime phase of an application has been detected, it then estimates the ground truth for standalone execution for the respective VM by pausing VMs running the co-running applications. Once an application completes execution, it then predicts the CPI of the application for standalone execution by aggregating the weighted sum of individual CPIs during every phase, the weights being the time for which an application stays for in every phase.

To the best of our knowledge, this paper is the first to address the concern due to the challenges present in Fair Pricing of users running applications when the number of co-running applications scale up. The contributions of our paper are as follows :-

- Fair Pricing Runtime Engine - a lightweight, robust and deployable runtime system that enables Fair Pricing in Data centers.
- Snapshot Shuttering - Performance Monitoring Units (PMU) based phase examination technique to estimate ground truth of standalone execution of an application.
- Phase changing mechanism - We also have introduced a Queue based multivariate binary classification technique, motivated from prior work [cite][cite] so as to capture phase change at a finer granularity as well as to discard noise which examining applications to detect phase change.

Using snapshot shuttering, we are able to precisely predict the performance degradation due to co-locations of SPEC applications [20] with a mean absolute error of around 2.8 % and often less than 1% and with a very small amount of overhead.

2. Background and Motivation

In this section, we briefly describe the broad classification of compute based instances that Amazon Web Services (AWS) provide for customers. We then discuss the pricing strategies of Amazon Web Services and the solution proposed in the prior work for enabling Fair Pricing in HPC systems.

2.1. Amazon Web Services

Amazon Web Services (AWS) provides services for on-demand computing resources in the cloud, with a pay-as-you-go pricing scheme. Clients typically rent Virtual Machines called instances where each instance can be configured reasonably based on the user's requirements. These instance types can be broadly categorized into Compute Optimized, Memory Optimized, GPUs etc. Within those broader categories clients can choose sub categories which are defined by their coarse grained micro architectural configurations namely number of VCPUs, size of RAM and SSD Storage.

With respect to co-location, Amazon broadly classifies its instances into two categories based on whether to allow co-location or not. AWS Dedicated Instances are instances that do not share hardware with other AWS clients, which clearly denies the possibility of a co-location scenario. Unlike Dedicated Instances, AWS Reserved / On-Demand Instances allows co-location of applications belonging to other users in the same node. The payment of these instances range from a few cents to a few dollars, per hour, per running instance.

Current day datacenters house individual nodes with numerous cores. A high level execution manager is responsible for scheduling applications belonging to multiple users in the same node based on some resource utilization heuristics. This co-location of multiple applications might lead to a situation where the applications might start running slower due to the contention of shared resources like last level cache and bandwidth to memory. For example, the degradation of user facing applications like web search, maps, email and other Internet services which latency-sensitive can be measured directly to see if extensive co-location hinders the application?s QoS target, as negotiated by the client and the service provider if there exists such policies. On the other hand a broad category of applications which are batch type, do not have a well defined performance metric like latency-sensitive applications to quantify their degradation. Also, in many cases the nature of the application is unknown before its execution. One cannot assign fixed QoS target, based on application specific metrics, to such a class of applications. As a result of this datacenter operators and system designers typically disallow co-locations which

results in unnecessary over-provisioning of compute resources reducing the overall resource utilization of datacenters.

2.2. Prior work and its limitations

To address the prediction of degradation due to co-location, prior work presents Fair Pricing on HPC Systems, a runtime technique to precisely predict the degradation due to co-location in High Performance Computing applications.

Prior work uses the help of a centralized controller which monitors all the applications that are running in a single node. Its principle responsibility is to conduct shuttering, a mechanism to measure and quantify the performance interference among the co-running applications [16] [21]. In essence, the controller periodically pauses all but one application for a very short period and monitors the performance impact on the lone application. This is being done in a round robin fashion to all the applications that are co-running to estimate the performance impact on every individual lone application. To measure this, their controller probes Performance Monitoring Units of each active job to acquire the performance data and logs it.

Fair Pricing is shown to be effective at predicting degradation due to co-locations for pairs of HPC applications. However, there are several primary limitations of this work. As the number or co-runners scale up, the prediction error increases enormously. This is due to the fact that the frequency at which every individual application running in the node is sampled in order to estimate ground truth decreases directly impacting the accuracy of estimation. Another limitation with prior work is when it tries to predict degradation for applications whose execution phases are flat, their shuttering technique repeatedly estimates ground truth unnecessarily for the entire execution as the variation among the estimated samples at subsequent intervals are very low within noise range throughout its execution period, meaning that estimation need not be repeated.

3. Snapshot Shuttering to build a Fair Pricing Runtime Engine

In this section we give an overview of our proposed Snapshot Shuttering technique for building a Fair Pricing Runtime Engine. Fair Pricing Runtime Engine is a lightweight and accurate runtime to measure the cross core interference on shared resources across multiple applications so as to effectively predict the degradation due to co-location. It runs on top of the host operating system monitoring the performance of VMs executing different applications belonging to different users as shown in Figure 3.

From Figure 3 we can see that every virtual machine is associated with 2 physical cores and its own private main memory. Fair Pricing runtime engine runs on the host operating system which monitors the VMs that are running on that node.

The principle methodology which we employ for Fair Pricing is by use of a technique called snapshot shuttering. The primary objective of snapshot shuttering is to periodically pause all the co-runners but one for a short time interval and to measure the performance interval of on the lone running application.

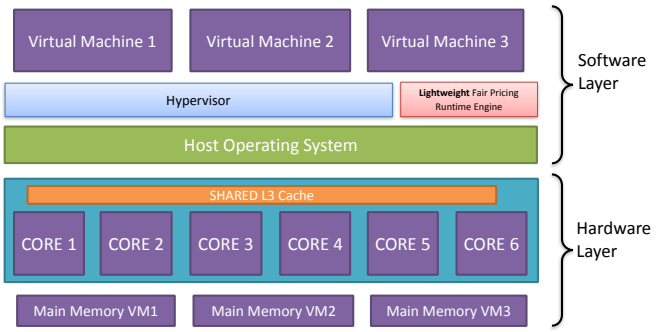

Figure 3: Overview of fair pricing runtime engine in the software stack

3.1. Phase changing mechanism

Batch applications exhibit time varying behavior. To capture such time varying behavior, researchers have proposed detecting program phases[cite][cite] where a phase is defined as a period of execution with stable behavior. We know that the CPI of the application in a particular phase is quite constant. So, we dynamically try to detect phase changes in the applications that are co-located with each other to obtain an indication for significant CPI variations of that application. When there is a phase change detected, we pause the co-runners and estimate the ground truth CPI for the application whose phase has been changed. We use that as an estimated CPI throughout a particular phase for the application. By this way, we greatly reduce the overhead that is caused by constantly pausing the co-runners at unnecessary frequent intervals even within a single phase as employed by the prior work [16].

There is considerable prior work done for static and dynamic detection of runtime phases in applications [**?**] [22] [23] [24] [25] [**?**] [26]. We employ a dynamic, low overhead phase detection technique using hardware Performance Monitoring Units (PMUs) that is inspired from prior work to suit our system. We study the characteristics of multiple performance events of every single application over time to identify phase changes among individual applications that are co-located. We demonstrate that these events should exhibit significant variation in their behavior of an application so as to be detected as a phase change for that application.

Figure 4 illustrates how phase changing mechanism works. The top part of Figure 4 shows the phases of 3 application with time. The bottom part of Figure 4 shows how snapshot shuttering works when phase changing is being detected. We can see that whenever a phase change is detected on a particular VM, the runtime pauses the execution of the other co-locating applications and tries to estimate the ground truth of the standalone run of the application.

3.2. Noise elimination

With the help of performance monitoring counters, we are able to obtain the phase characteristics of multiple applications over time. We obtain them by measuring their characteristics at coarse grained intervals of one second and plotting them with respect to time. Even for such coarse grained intervals we observe frequent spikes in the reported CPI of execution. These

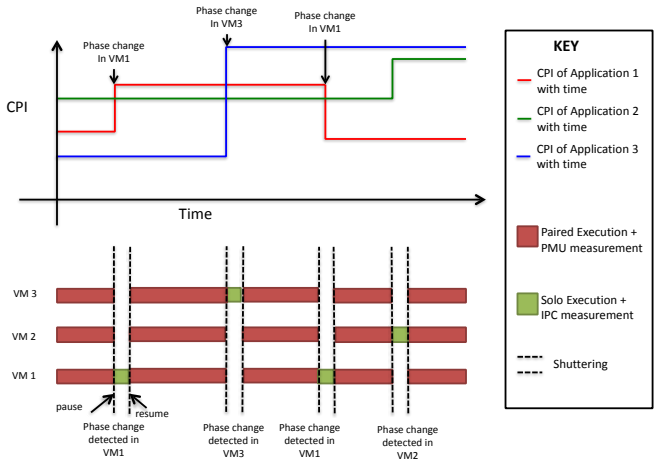

Figure 4: Shows how shuttering is done based on phase changing methedology

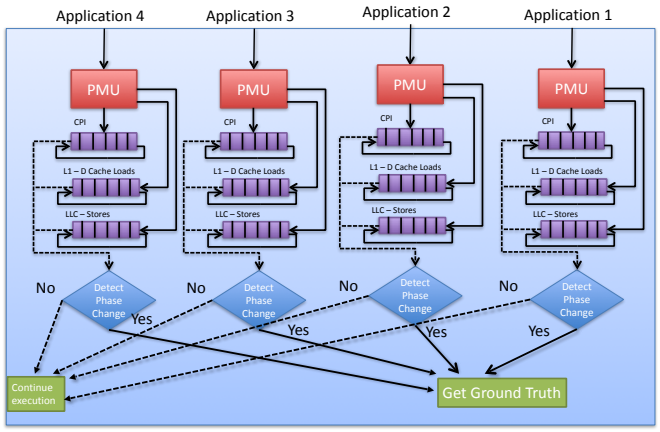

Figure 5: Overview of noise elimination in runtime

spikes are short but significant variation in execution characteristics due to some micro architectural behavior limitation like cache warm up or page fault, for a short interval of time. Given as such, the phase changing methodology that we employ should not classify them as a new phase.

To address this challenge, we employ a binary classification technique. As shown in the 5, we use separate queues for different PMU types of a particular application. Each queue has a window of size k which stores the k latest events that has been reported by PMUs for that application.

At the start of every phase we estimate ground truth. We declare a phase change in the application whenever a significant number of the elements present in the queue show high variation compared to the ground truth. In, this way we are able to eliminate incorrect phase change classification due to short durational variations in the program.

Figure 5 illustrates how binary classification is done for every individual PMU type corresponding to individual application. Binary classification is done separately for PMU type of every application and whenever there is a phase change observed by either of the PMU types, then ground truth for that application for which phase change is detected is being estimated.

4. Snapshot Shuttering Methodology in Fair Pricing Runtime Engine

4.1. Algorithms present in Fair Pricing Runtime Engine

In this section, we present the algorithms that constitute the Fair Pricing Runtime Engine. The main core routine is Algorithm 1s. The other algorithms are being invoked as function call from the core routine algorithm.

Below, we define a list of common data structures and constants used by the algorithms.

- A [ ], an array of co-located VMs running applications
- C [ ], an array of counter types
- T, the length of measurements of performance counters in seconds
- The types of performance counters measured.
- Queue[ ][ ], a two dimensional array of performance counter measurements stored for every VM and its respective performance counter type.
- PC_A is a list of all VMs that have been classified by the Binary Classification algorithm to require a Phase Change.
- G_VM, the VM to get the ground truth.
- $\delta_{counter}$ Threshold uniqe to each counter so as to issue phase change

Algorithm 1 Fair Pricing Runtime Engine

```
1:  while true do
2:      Measure(&A,T,&C)
3:      PC_A=BinaryClassification(&QueueVM,&C)
4:      for VM in A do
5:          for counter in counterType do
6:              if PC_A[VM][counter] then GetGroundTruth(VM) break
7:              end if
8:          end for
9:      end for
10: end while
```

Algorithm 2 Measure(A[ ], T, C[ ])

```
1:  for VM in A do
2:      for counter in counterType do
3:          StartCounterMeasurement
4:      end for
5:  end forsleep S seconds
6:  for VM in A do
7:      for counter in counterType do
8:          StopCounterMeasurement
9:          Queue[VM][counter].enqueue(MeasuredCounterValue)
10:         Queue[VM][counter].dequeue()
11:     end for
12: end for
```

Algorithm 3 BinaryClassification(QueueVM[],C[])

```
for VM in A do
    for counter in counterType do
        if 70%oftheelementsinqueue >= δ_counter then
PC_A[VM][counter] = true; break
        end if
    end for
end for
```

Algorithm 4 Getting Ground Truth(G_VM)

```
for VM in A do
    for counter in counterType do
        if G_VM != VM then
            pause VM
        end if
    end for
end for
Measure(&G_VM, T, C[ ])
for VM in A do
    for counter in counterType do
        if G_VM != VM then
            resume VM
        end if
    end for
end forsleep S seconds
```

4.2. Description of Algorithms in Fair Pricing Runtime Engine

At start of every iteration, the core routine algorithm obtains the Performance Monitoring Unit (PMU) measurements for each PMU type for an interval of 1 second for every application which is co-located. It then appends every single measurement to their respective queues for specific PMU types of that application and discards the oldest value for every queue.

The queue is then subjected to a binary classification technique as shown in Algorithm 3. This technique tells whether a significant portion of the elements present in that queue is different from so as to indicate a phase change. In a situation where it declares a phase change, it obtains the average of the elements in the queue. Subsequent PMU measurements will now be compared against this average to indicate variations in phase changes.

It is to be noted that we are using more than one PMU type to indicate phase change as shown in Figure 5. The reason behind this is based on the following observation. Even with coarse granular intervals between for measurements we could observe in some applications spikes in execution. These do not attribute to a phase change and using a single value prediction might mis-classify them as a phase change. Hence we need a queue based phase detection mechanism.

From Figure 5 we can see that every application has its associated PMU. For every application, the PMUs maintains a queue of CPI, L1-Dcache Loads and LLC-stores as shown in the figure. It determines phase classification based on the the inputs given by each type of PMU. We use a conservative approach to detect phase changes by declaring a phase change whenever any of the PMU type declares a phase change. This is because missing out a phase which runs for a longer time would result in drastic increase in prediction accuracy. However predicting phase changes which are not causes no harm as the frequency of phase changes are very low and the resulting overhead associated with a few extra phase changes are bearable within tolerable overhead limits.

In the event of phase change as declared by either of the PMU types corresponding to particular application, Algorithm 4 tries to estimate the ground truth representing the CPI of the lone run of that application. It pauses all the other co-running applications and measures the performance statistics of the lone run of a single application. This is the estimated ground truth till the phase changes for that application.

4.3. Tuning Snapshot shuttering mechanism

The implementation of Snapshot Shuttering mechanism that we use presents a number of challenges. There are a lot of parametric decisions that we have to consider to achieve high prediction accuracy.

Perf profiler tool for linux based on `perf_events` interface supports a list of measurable events. Selecting a subset of events to monitor the phase characteristics of a wide variety of applications for phase detection was a major challenge. Application specific micro-architectural insights towards choosing PMU types were very helpful for us. Most significant phase changes was explicitly portrayed by micro-architectural events like branch-misses, L1-data cache and LLC-loads. The explanation towards that is as follows:-

A single procedures or code segment that executes the same basic blocks repeatedly show similar CPI measurements with time and they constitute what defines as a single phase. Typical examples of these are iterative processing applications performing different tasks. When a they are being executed iteratively, PMC measurements like branches, L1-Dcache accesses remain fairly same due to the execution of repetitive code segments. When a there is a change in the executing code segment, PMU measurements change as the the control flow and the data flow characteristics of the newly executing code segments will be different to the previous one. This might be reflected at different granularities by CPI measurements for different phase changes within applications, they are hugely reflected by such PMU measurements and hence can be useful information to detect phase changes for varying granularity. In our experiments we observed that L1-dcache accesses and Last Level Cache characteristics are able to identify phase changes across applications in a much accurate manner.

In the prior work, shuttering has been done once every 200ms for the individual applications in a round robin fashion. In our work, we check for phase changes at a much coarser granularity of about once every 1 second for all applications. We use this the snapshot of phase of the application to observe phase changes and to decide whether to perform shuttering or not. The size of the Queues for PMC type is 7 recent samples. The runtime engine declares a phase change if 5 of them are significantly out of phase. The magnitude by which each PMC type should be off is different for individual PMC types in a particu-

lar architecture. Every time a phase change is being declared, we wait 5ms before we start to measure. This serves as a cache warm up time for sampling the performance characteristics of the lone application.

5. Quantifying Overhead

In this section we present a theoretical model for calculating the overhead based on how prior work as well as Fair Pricing Runtime Engine is implemented.

5.1. Methodology for calculating overhead for precise shuttering

The methodology by which the prior work calculates overhead is as follows. The precise shuttering mechanism works by alternating an application's execution environment between one where the co-runners are executing and another where they are effectively paused to estimate ground truth. After, every interval of time they pause all the co-runners except one and measure its CPI (shuttering). This is serves as the estimated CPI for the ground truth execution of that application. They perform shuttering for all the applications in a round robin fashion. To estimate degradation for an application, they calculate the ratio of CPI between when the co-runners are running and when the co-runners are paused.

Mathematically the overhead due to precise shuttering runtime in seconds is as follows:-

$$\frac{\sum t_{pause} X Num_{co-runner}}{Freq_{shutter} X Num_{application}} \quad (1)$$

where,
t_{pause} - Time (in milli seconds) for which every co runner would pause during shuttering
$Num_{application}$ - Total number of applications running
$Num_{co-runner}$ - Number of co-runners
$Freq_{shutter}$ - Time between subsequent shutters

5.2. Methodology for calculating overhead for Snapshot Shuttering

The experimental methodology by which overhead is calculated in the proposed Snapshot Shuttering technique is analogous to that of precise shuttering. The over head due to the Snapshot Shuttering mechanism is caused due to the time for which an application is paused whenever there is a phase change is either of the co-runners and that concerned application is trying to estimate its ground truth CPI. The amount of time spent as overhead in the previous technique is a function of execution time of the application while the amount of time spent as overhead in our technique is a function of number of phase changes in the co-runners.

The generalized equation of overhead in seconds is as follows:-

$$\frac{t_{pause} \sum_{i=1}^{Num_{corunners}} Num_{phases_i}}{1000} \quad (2)$$

where,
t_{pause} - Time (in milli second) for which every co runner would pause during Snapshot Shuttering
$Num_{co-runner}$ - Number of corunners
Num_{phases_i} - Frequency at which shuttering is performed

6. Experimental Setup

Table 1: Machine Specification in our experimental results

Processor	Microarchitecture	Kernel
Intel(R) Xeon(R) CPU E5-2420 0 @ 1.90GHz	Sandy Bridge-EN	3.8.0
Intel(R) Xeon(R) CPU E5-2407 v2 @ 2.40GHz	Ivy Bridge	3.11.0
Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz	Haswell	3.11.0

We evaluate Fair Pricing methodology on three commodity multi-core processors are shown in Table 1. We use SPEC CPU2006 [20] with `ref` inputs as our workloads.

To mimic the nature of execution present in current day data centers, we run our applications inside Virtual Machines (VMs), assuming each VM belongs to a particular user. We setup VMs using Kernel-based Virtual Machine (KVM) [27] [28]. It is a full virtualization solution for Linux on x86 hardware containing virtualization extensions. Each Virtual Machine is called a KVM guest and has private virtualized hardware, network card and disk.

The configuration of each KVM guests in our environment are as follows:- 2 Virtual CPUs for every KVM guest. 2GB of dedicated RAM for every KVM guest. 16GB of dedicated disk space for every KVM guest. Linux version 3.2.0

We use the linux perf tool to measure hardware Performance Monitoring Units (PMUs) [29].

7. Evaluation

In this section we evaluate the effectiveness of our Snapshot Shuttering technique to predict degradation due to co-location. We use metrics like accuracy, overhead, pricing fairness to evaluate our technique. We also compare our technique against precise shuttering methodology and show that we achieve a higher accuracy in predicting degradation due to co-location with the same amount of overhead.

The values that we fix for our experimental parameters are as follows:-

- Performance counters are measured at a frequency of once every second. Phase change is also test for with the same frequency.
- After every phase change, co-runners are paused for 105ms, with 5 ms being the cache warm up time and 100ms being the performance counter measurements to obtain ground truth for lone application run.
- Queue window is assigned as a fixed stream of 7 recent performance counter measurements.
- The performance counter types that we use to analyze runtime phases are LLC-stores, L1-dcache-loads. The runtime issues a phase change when 5 or more elements in the queue are either LLC-stores are 50% more than or 25% less than

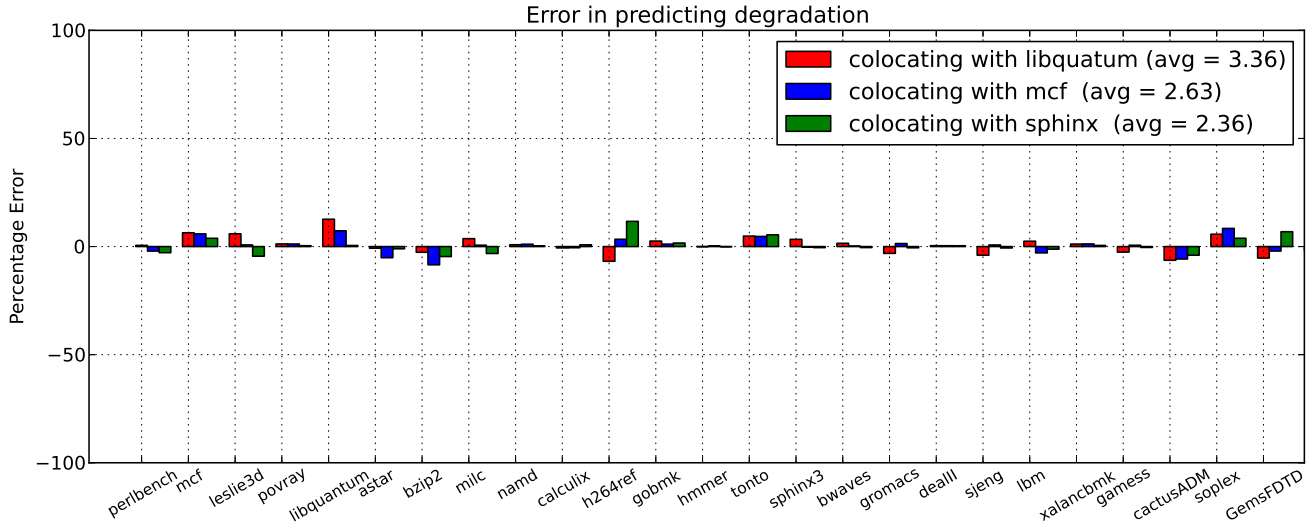

Figure 6: Accuracy in prediction of degradation due to co-location using Snapshot Shuttering technique

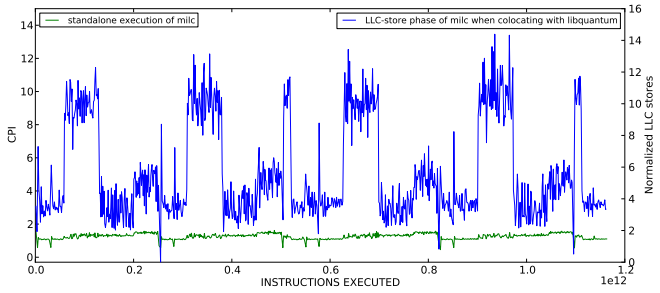

Figure 7: Phase of the CPI of milc when it is running alone VS phase of LLC stores of milc there are 3 instance of libquantum co-runners running along

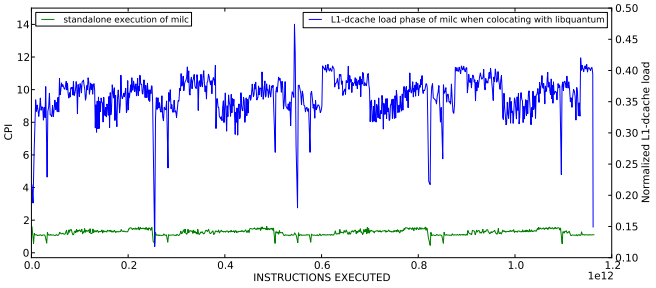

Figure 8: Phase of the CPI of milc when it is running alone VS phase of L1-d-cache loads of milc there are 3 instance of libquantum co-runners running along

the average of the sample that is collected in the previous phase as detected by that counter. Similarly, the bounds with respect to L1-dcache-loads are 100% more or 50% less respectively. Phase change mechanism is executed if either of the PMC types issue a phase change.

7.1. Accuracy in prediction of degradation due to co-location by Fair Pricing Runtime Engine

In this sub-section we give a overview of how Snapshot Shuttering is able to accurately predict degradation due to co-location. For these experiments, we co-locate four SPEC benchmarks together and try to measure the effectiveness of our solution in predicting degradation due to co-location. Most of our experiments are with contentious co-runners as predicting degradation in the presence of contentious co-runners is much more difficult due to their impact on shared memory sub systems.

Figure 6 describes the accuracy of SPEC applications when they are co-located with three different types of co-runners libquantum, mcf and sphinx3 respectively. A single instance of the applications in the xaxis is launched in a single VM along with 3 instances of the applications that are co-locating in 3 separate VMs. The yaxis represent the error that occurs in predicting degradation due to co-location. We obtain that be comparing our estimated degradation with the ground truth obtained by running application without our runtime. For our experiments we run each benchmark three times and take the average of the three runs to eliminate the minor run to run variability that exits while running SPEC applications.

On observing these graphs we obtain several interesting results. One of the main insights that we get out of our experiments is that prediction becomes more difficult as the contentiousness of the co-runner increases. A simple case can be observed from Figure 6. The average error with respect to predicting degradation due to co-location is around 3.36% when libquantum is the co-runner but it gets lesser and lesser when sphinx is the co-runner. It should be noted here that the magnitude of contentiousness arranged in the decreasing order is libquantum, mcf, sphinx respectively as shown clearly in [30].

We also observe that the average error that is caused due to predicting degradation due to co-location for around 75% of the application is less than 1.5%. There are very few situations where the peak error exceeds more than 10%. One of them is when libquantum is co-located with 3 more instances of

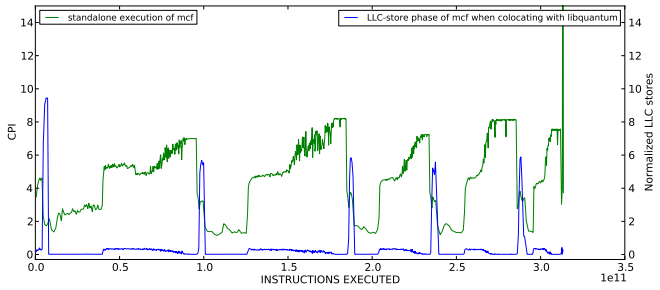

Figure 9: Phase of the CPI of mcf when it is running alone VS phase of LLC stores of mcf there are 3 instance of libquantum co-runners running along

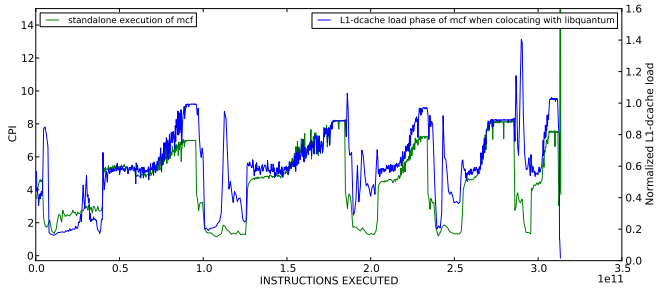

Figure 10: Phase of the CPI of mcf when it is running alone VS phase of L1-d-cache loads of mcf there are 3 instance of libquantum co-runners running along

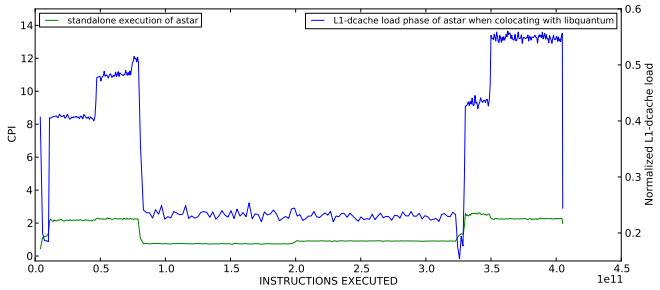

Figure 11: Phase of the CPI of astar when it is running alone VS phase of astar when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are libquantum co-runners running along

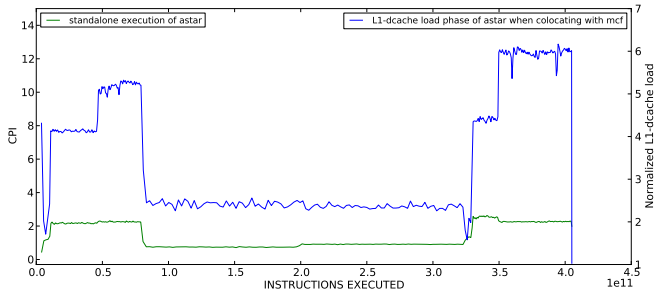

Figure 12: Phase of the CPI of astar when it is running alone VS phase of astar when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are mcf co-runners running along

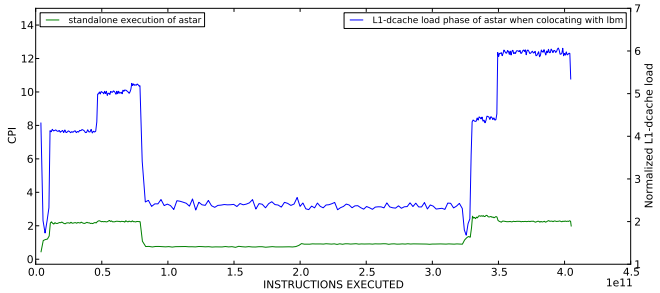

Figure 13: Phase of the CPI of xalancbmk when it is running alone VS phase of xalancbmk when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are lbm co-runners running along

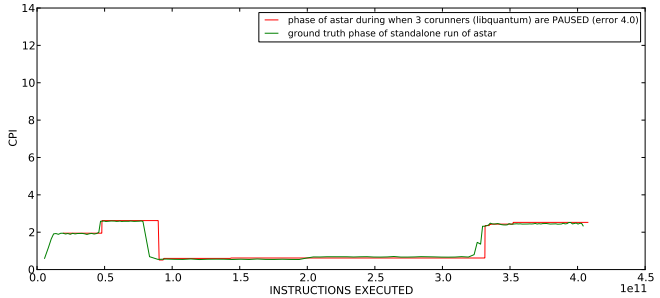

Figure 14: Phase of the CPI of astar when it is running alone VS phase of astar when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are libquantum co-runners running along

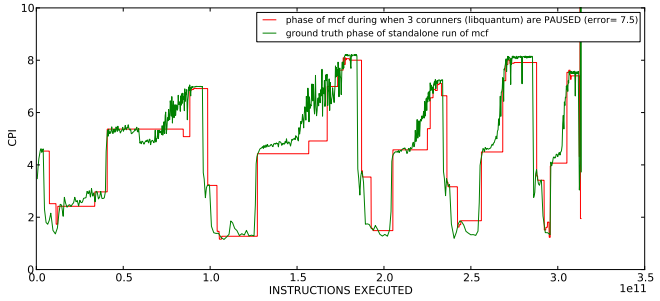

Figure 15: Phase of the CPI of mcf when it is running alone VS phase of estimated CPI of mcf when by pausing libquantum co-runners using Snapshot Shuttering technique

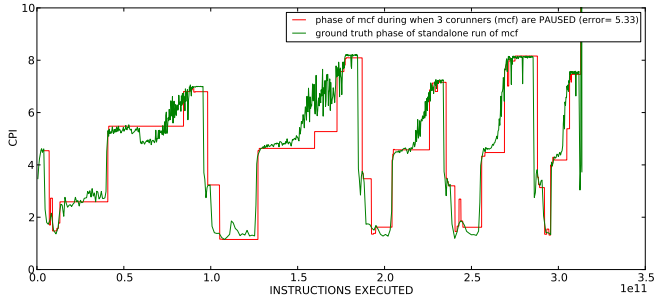

Figure 16: Phase of the CPI of mcf when it is running alone VS phase of estimated CPI of mcf when by pausing mcf co-runners using Snapshot Shuttering technique

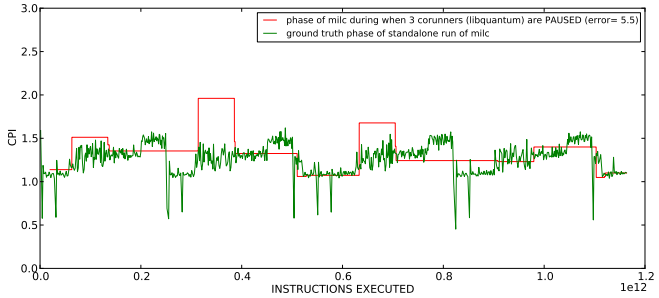

Figure 17: Phase of the CPI of milc when it is running alone VS phase of estimated CPI of milc when by pausing libquantum co-runners using Snapshot Shuttering technique

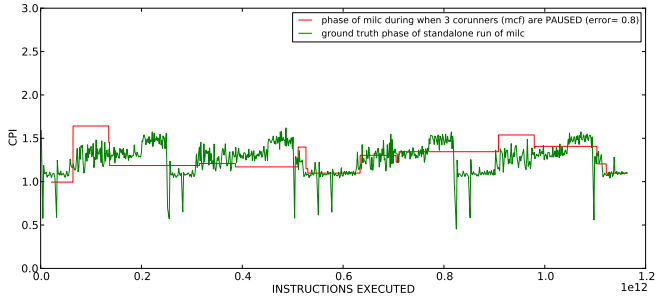

Figure 18: Phase of the CPI of milc when it is running alone VS phase of estimated CPI of milc when by pausing mcf co-runners using Snapshot Shuttering technique

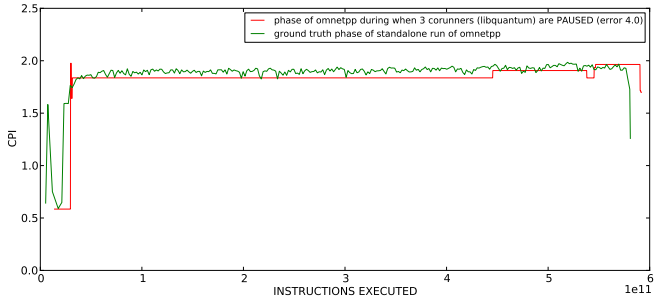

Figure 19: Phase of the CPI of omnetpp when it is running alone VS phase of estimated CPI of omnetpp when by pausing libquantum co-runners using Snapshot Shuttering technique

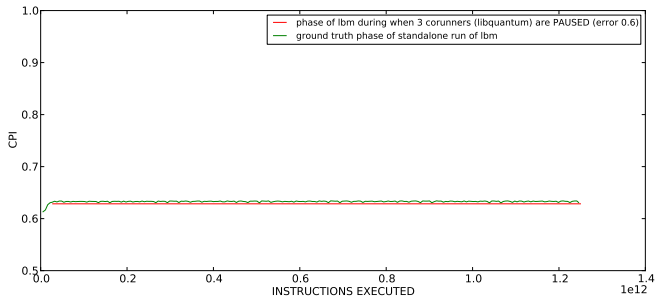

Figure 20: Phase of the CPI of lbm when it is running alone VS phase of estimated CPI of lbm when by pausing libquantum co-runners using Snapshot Shuttering technique

libquantum as co-runners. This is a tricky situation because the pressure on the shared memory subsystem by 4 instances of libquantum will be very high. Due to that estimation of CPI for the lone run of libuquantum becomes comparatively more challenging.

We have also observed that for the applications which are less sensitive co-locations whose degradations are comparatively lesser, prediction becomes much easier. Though this is true for most of the common cases, certain applications like lbm show low error even though they are comparatively contentious applications. On performing phase analysis for lbm, we observe that the phase CPI of lbm is very flat and so is the phase as portrayed by its PMC types and hence prediction becomes much easier for us.

7.2. Phase Level behavior

To first obtain such low error rates, we investigated the phase level behavior of the multiple applications based on a series of performance counter events that show how they were useful in determining phase changes across applications.

Figure 7 depicts the phase of CPI during standalone execution of milc over the phase of LLC stores of milc when milc is co-located with 3 instances of libquantum. Figure 8 depicts the phase of CPI for standalone execution of milc over the phase of L1-dcache-loads of milc when again milc is co-located with 3 instances of libquantum. Figure 9 and 10 provide the same information as Figure 7 and Figure 8 just that the main application that is being observed is mcf instead of milc.

From Figure 7 and Figure 8 we get to know that changes in the phase of lone execution of milc is is being depicted much significantly by the PMC type Last Level Cache Stores (LLC-stores) but, not as much by PMC type L1-d-cache misses. On the other hand, from Figure 9 and Figure 10 we observe that changes in the phase of lone execution of mcf is is being depicted much significantly by the PMC type last L1-d-cache misses but, not as much by PMC type LLC-misses. This made us conclude that compared to a single PMC type, a subset of PMC types will be much more effective in accurately predicting phase changes in an application.

Although we observed the need of more than one PMC type to predict phase changes, a general trend that followed is in our experiments is that whenever there is a phase change with respect to CPI in our applications, most of the micro-architectural events also change. However in some situations, where some applications portrait distinct irregularities, some specific PMC types are able to capture them much more effectively than the others. An analysis of the application characteristics of SPEC lead us to the conclusions that a very few subset of the PMC types are needed so as to capture distinct phase changes within applications. Though requirements for phase analysis for our purpose require methodologies which have negligible overhead, they need not be accurate and fine grained at the basic block level. So the methodology that we employ based on PMCs seem to fit very well in our system.

Another experiment that we felt was interesting for us to conduct was to see the effect on PMC measurements with different co-runners to see if varying co-runners affect PMC measurements and in turn phase prediction. Figure 11,12,13

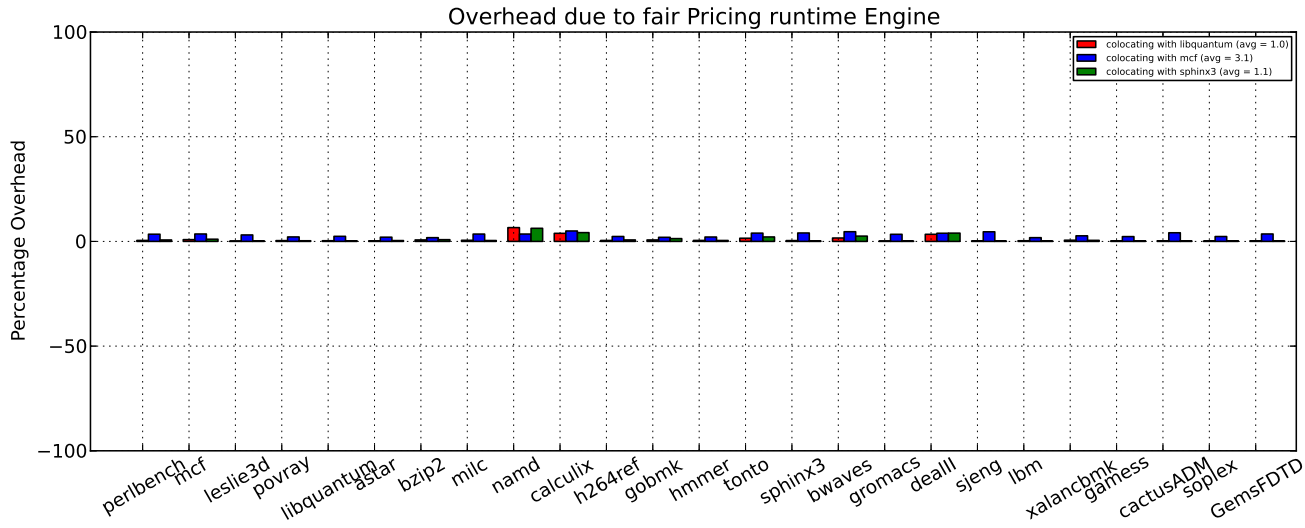

Figure 21: Overhead due to Fair Pricing Runtime Engine when it tries to predict degradation due to co-location

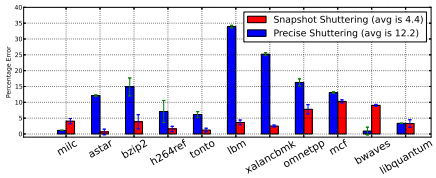

Figure 22: Comaring accuracy between Snapshot Shuttering and precise shuttering technique when libquantum is the co-runner

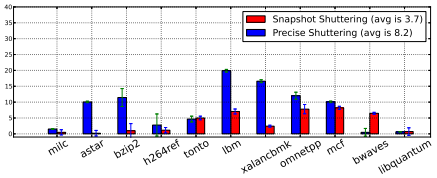

Figure 23: Comaring accuracy between Snapshot Shuttering and precise shuttering technique when mcf is the co-runner

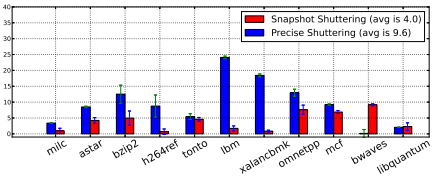

Figure 24: Comaring accuracy between Snapshot Shuttering and precise shuttering technique when lbm is the co-runner

shows the phase of CPI of standalone execution of astar versus the phase of L1-d-cache misses of astar when it is co-running with libquantum, mcf and lbm respectively. From those figures we are able to observe that even when the co-running application changes, the measurements that have been reported by the PMCs have been fairly consistent. Even though there might be slight variations with respect to phase prediction caused due to the run to run variations in the measurement of PMCs, the PMC types that we use are able to capture distinctive phase changes quite consistently.

Figure 14 compares the phases of application astar when it is run alone versus the phase of the application when it co-runners are paused by using smart shuttering technique. We can clearly see in Figure 14 how phase changes corresponding to the lone run of the application is being captured by the PMC event L1-dcache-loads at different points of time. The indication of such events marks the starting of new phase which is represented by flat red lines as shown in Figure 14. The error corresponding to this single run for predicting degradation was 4%.

Figure 15 and Figure 17 shows the comparison of phases of CPI of standalone run of the application mcf with the phase of estimated CPI of standalone run of mcf with the help of smart shuttering technique.

7.3. Overhead due to Fair Pricing Runtime Engine

Figure 21 presents the overhead incurred in predicting degradation due to co-location by the Fair Pricing Runtime Engine. The overhead calculated here is the mean overhead due to phase changes in all the application co-located in the same node. The overhead is minimal. The overhead suffered by applications when they are co-locating with libquantum and sphinx3 are around 1%. This is due to the fact that libquantum have barely few phase changes. On the other hand, though sphinx3 has some phase changes they occur very rarely and hence there is barely any degradation due to co-location.

The overhead when mcf is a co-runner is three times higher than the common case with respect to libquantum and sphinx3. This is probably the worst case scenario that we have faced in our experiments. This is due to the fact that mcf has many distinct phases and whenever there is a phase change being detected for either of the 3 applications running mcf, all the other applications will be paused. This adds to the increase in overhead when mcf in the co-runner.

We also observe that applications like namd,calculix and dealII which exhibit highly irregular phases show a slightly more overhead than the other typical applications. On an average around 60% of the applications when co-located along with libquantum and sphinx have overhead of less that 0.5% overhead. The reason why it happens is because these applications have a comparatively flat phase of execution. Hence when the applications start executing initially, ground truth is obtained once and this is used as the estimated ground throughout the execution period of the application. The estimated ground truth not only has low error in prediction, it also incurs a negligible overhead in the order of milliseconds when compared to the time for which the application runs.

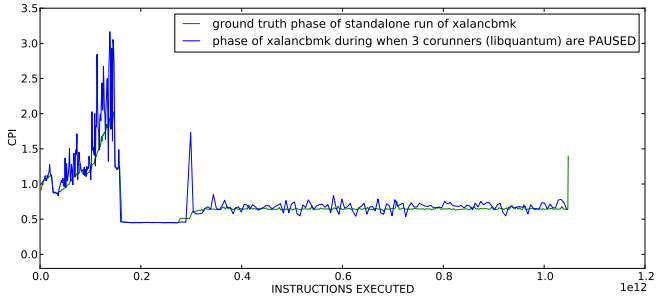

Figure 25: Phase of the CPI of xalancbmk when it is running alone VS phase of xalancbmk when trying to estimate ground truth CPI by pausing by precise shuttering technique when there are co-runners running along

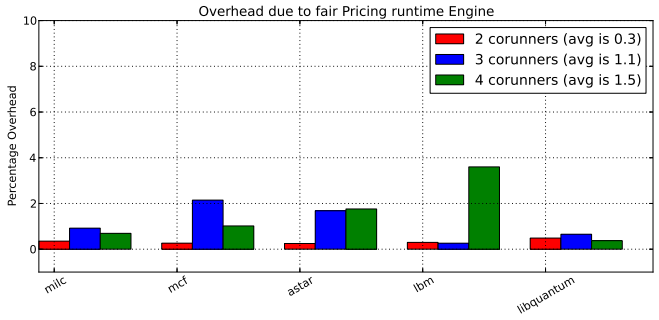

Figure 27: Overhead due to precise shuttering mechanism which is used by [16] to predict degradation due to co-location when libquantum is the co-runner

7.4. Comparison with shuttering

Figure 22, 23, 24 shows the comparison when we use Fair Pricing runtime engine with smart shuttering as compared to the prior work which uses precise shuttering. We can clearly see that for almost all of the benchmarks Fair Pricing Runtime Engine does much better than precise shuttering. The only exceptional case is bwaves. The reason is due to the fact that the phase of bwaves even when observed at a comparatively coarse granular intervals of around one sample points at every one second, contains a lot of spikes. The binary classification technique that we use disregards these points of noise while predicting phase changes. Though such points do not attribute to phase change, they manipulate towards the final average CPI of the application and discarding them will result in a prediction of degradation which is more than that of what it is actually is which results this unusual variability. Even at such a corner case, our runtime system is able to predict degradation within 10 % error.

Figure 25 and Figure 26 compares the phases of application xalancbmk when it is run alone versus the phase of the application when it is paused by using precise shuttering technique and when it is paused by using smart shuttering technique respective in the presence of co-runners. We can clearly see that in Figure 25, the phase of the pause periods of the application when paused by smart shuttering matches the ground truth execution phase of lone application much more than the phase the phase of the pause periods of the application when paused by precise shuttering.

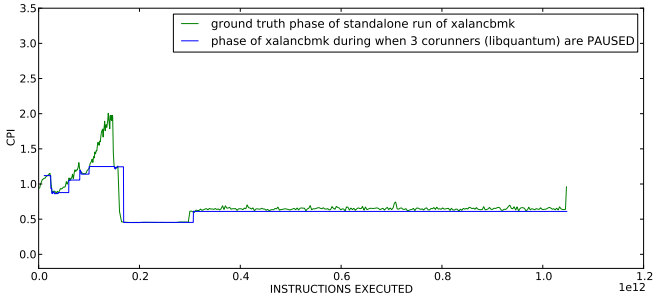

Figure 26: Phase of the CPI of xalancbmk when it is running alone VS phase of xalancbmk when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are co-runners running along

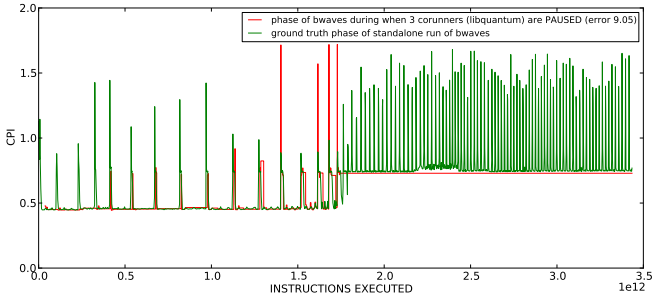

Figure 28: Phase of the CPI of bwaves when it is running alone VS phase of bwaves when trying to estimate ground truth CPI by pausing by Snapshot Shuttering technique when there are co-runners running along

Figure 27 shows the overhead associated with predicting degradation due to co-location by precise shuttering technique. We can see that as the number of co-runner increases, the overhead of the prior technique increases drastically across all type of co-runners. This is due to the fact that frequent shuttering would constantly thrash the shared caches to fill up data corresponding to the application that was shuttered most recently. During longer though less frequent shuttering this scenario can be much avoided. Moreover when it comes to applications with flat phases, snapshot shuttering avoids pausing the co-runners for most of the time and hence overhead turns out to be much lesser.

8. Conclusion

This work presents Fair Pricing Runtime Engine, a novel approach to predict degradation due to co-location and to price users accordingly. The Fair Pricing Runtime Engine is has negligible performance overhead, operates without any special hardware or programmer support, and can accurately predict degradation due to co-location. These combination of features gives our Runtime the capability to predict real time applications with high precision and to price the users running them accordingly.

Using this mechanism we are able to predict the degradation due to co-location of applications with a mean absolute error of about 4% within an overhead of around 1%. This situation we feel would put users to a situation to use Reserved/On Demand Instances which can prevent over provisioning of resources and improve energy efficiency.

References

[1] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[2] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary-Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[3] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, March 2010.

[4] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra 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, pages 129–142, New York, NY, USA, 2010. ACM.

[5] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 38(1):129–142, March 2010.

[6] Sriram Govindan, Jie Liu, Aman Kansal, and Anand 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, pages 22:1–22:14, New York, NY, USA, 2011. ACM.

[7] Amazon Inc. Elastic Compute Cloud, November 2008.

[8] Ki-Woong Park, Jaesun Han, JaeWoong Chung, and Kyu Ho Park. Themis: A mutually verifiable billing system for the cloud computing environment. *Services Computing, IEEE Transactions on*, 6(3):300–313, July 2013.

[9] E. Elmroth, F.G. Marquez, D. Henriksson, and D.P. Ferrera. Accounting and billing for federated cloud infrastructures. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pages 268–275, Aug 2009.

[10] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[11] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou 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)*, ASPLOS '13, pages 89–100, New York, NY, USA, 2013. ACM. Acceptance Rate: 23

[12] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM. Acceptance Rate: 19

[13] Stavros Harizopoulos, Mehul A. Shah, Justin Meza, and Parthasarathy Ranganathan. Energy efficiency: The new holy grail of data management systems research. *CoRR*, abs/0909.1784, 2009.

[14] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 155–166, Feb 2013.

[15] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou 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, pages 248–259, New York, NY, USA, 2011. ACM. Acceptance Rate: 21

[16] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling fair pricing on high performance computer systems with node sharing. *Scientific Programming*, 22(2):59–74, 2014.

[17] C. Isci and M. Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 121–132, Feb 2006.

[18] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: duration predictions and applications to dvfs. *Micro, IEEE*, 25(5):39–51, Sept 2005.

[19] E. Duesterwald, J. Torrellas, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 220–231, Sept 2003.

[20] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[21] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Qualitytime: A simple online technique for quantifying multicore execution efficiency. In *ISPASS*, 2014.

[22] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 317–328, Washington, DC, USA, 2012. IEEE Computer Society.

[23] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 104–115, Nov 2011.

[24] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 245–257, New York, NY, USA, 2000. ACM.

[25] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 191–202, March 2005.

[26] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. Accurate off-line phase classification for hw/sw co-designed processors. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[27] Qumranet. Kvm: Kernel-based virtualization machine. Technical report, Qumranet, 2007.

[28] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[29] Arnaldo Carvalho de Melo. The new linux?perf?tools.

[30] Lingjia Tang, Jason Mars, and Mary Lou 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 (co-located with PLDI 2011)*, pages 12–21, New York, NY, USA, 2011. ACM.