Caliper: Interference Estimator for Multi-tenant Environments Sharing Architectural Resources

RAM SRIVATSA KANNAN, MICHAEL LAURENZANO, JASON MARS, and LINGJIA TANG, University of Michigan, Ann Arbor, USA
JEONGSEOB AHN, Ajou University, South Korea

We introduce **Caliper**, a technique for accurately estimating performance interference occurring in shared servers. Caliper overcomes the limitations of prior approaches by leveraging a micro-experiment based technique. In contrast to state-of-the-art approaches that focus on periodically pausing co-running applications to estimate slowdown, *Caliper* utilizes a strategic phase-triggered technique to capture interference due to co-location. This enables *Caliper* to orchestrate an accurate and low-overhead interference estimation technique that can be readily deployed in existing production systems. We evaluate *Caliper* for a broad spectrum of workload scenarios, demonstrating its ability to seamlessly support up to 16 applications running simultaneously and outperforming the state-of-the-art approaches.

CCS Concepts: • **Computer Architecture** → **Datacenter Systems Design**; *Datacenter Contention*;

Additional Key Words and Phrases: Datacenter Design, Cache contention, DRAM bandwidth contention, Fairness, muti-core, Interference, Performance, System software metrics

ACM Reference Format:
Ram Srivatsa Kannan, Michael Laurenzano, Jason Mars, Lingjia Tang, and Jeongseob Ahn. 2019. Caliper: Interference Estimator for Multi-tenant Environments Sharing Architectural Resources. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2019), 27 pages. https://doi.org/10.1145/3323090

1 INTRODUCTION

Improving resource utilization in modern multi-core systems has been identified as a critical design goal by large-scale datacenters designers [14]. The motivating factor leading to this trend is the underutilization of multi-core processors due to overprovisioning. Towards realizing this objective, co-locating multiple batch applications on a single server has proven to be beneficial [19, 33, 34, 40, 44, 61–63]. In cloud computing, system virtualization techniques have been instrumental in providing performance isolation while co-locating multiple applications present in under-utilized servers.

[1]New Paper, Not an Extension of a Conference Paper

Authors' addresses: Ram Srivatsa Kannan, ramsri@umich.edu; Michael Laurenzano, mlaurenz@umich.edu; Jason Mars, profmars@umich.edu; Lingjia Tang, lingjia@umich.edu, University of Michigan, Ann Arbor, 2260 Hayward Street, Ann Arbor, MI, 48105, USA, ramsri,mlaurenz,profmars,lingjia@umich.edu; Jeongseob Ahn, Ajou University, Suwon, South Korea, jsahn@ajou.ac.kr(Correspondingauthor).

XXXX-XXXX/2019/1-ART1
https://doi.org/10.1145/3323090

Although the commodity hypervisors such as Xen and KVM have been achieving performance isolation at some level through strict CPU reservations, static partitioning of memory/disk space, and network bandwidth [32, 51], it is inevitable to avoid shared resource contentions, especially at micro-architectural resources. These performance critical resources including last-level cache, memory controller, and main memory bandwidth cause the slowdown of applications in co-located environments. Moreover, the magnitude at which applications are slowed down is highly dependent on the nature of the co-running applications and the availability of shared resources [19, 40].

Under such circumstances, it is essential to have the ability to estimate the slowdown of applications caused due to co-location accurately. Such slowdown estimates could enable resource allocation of shared resources to each application in a slowdown aware manner motivated towards providing strong Quality-of-Service (QoS) guarantees. Also, in Infrastructure-as-a-Service (IaaS) clouds, such a mechanism could be used to bill its customers appropriately based on the amount of slowdown that their applications have been subjected to by the co-running applications [16, 58].

There have been many efforts that try to estimate slowdown of applications at runtime [16, 17, 23, 25, 40, 42, 49, 58, 61, 68]. Prior software approaches [16, 25, 61, 68] utilize an online runtime system that periodically pauses all the applications except one for a short time, thus allowing the running application to monopolize the computing resources on the system during those pause periods. The performance of the running application during such pause periods is used to determine slowdown.

A few other hardware enabled approaches [22, 23, 42, 58] designed to estimate slowdown are based on a methodology that aims at modeling interference bottom up as an aggregate of interference across multiple processor subsystems. However, this may prove to be prohibitively difficult as core counts increase and processor architectures accrue performance improvement mechanisms that are ever larger in number and complexity. These approaches leave several challenges that pose barriers to its adoption:

(1) **Low accuracy:** The most recent state-of-the-art technique addressing this problem [16] **neglects the notion of application phases** and pauses co-running applications periodically at millisecond granularity. This methodology shows estimation errors of up to 40% leaving significant room for improvement in accuracy.
(2) **High overhead:** It has been reported that datacenter providers tolerate no more than 1% to 2% degradation in performance to support dynamic monitoring approaches in production [55]. However, the execution time overhead of the state-of-the-art software interference estimation technique can be as significant as 12% [16].
(3) **Non-reliable (or less scalable):** The accuracy and the overhead of prior approaches [16, 23, 25, 42, 58] deteriorate as the number of co-running applications increases. As the number of cores on modern servers keeps increasing, deploying a technique that inadequately supports current and future levels of multi-tenancy would not be a preferred choice.
(4) **Priori knowledge:** Another class of static techniques requires *a priori* [24, 40] knowledge about all workloads and the profiling for each type of workload. This requirement limits the types of workloads for which such a technique can be applied and, more broadly, the kind of datacenters that can adopt the approach (e.g., public clouds).

In this study, we design a mechanism called *micro-experiments* – short-lived measurements of application performance under different conditions – to accurately estimate the interference experienced by applications due to performance degradation. On top of this mechanism, we introduce **Caliper** to estimate slowdown of an application at runtime with high accuracy and negligible overhead. A micro-experiment is a period during which the performance of an application is abstracted from the interference incurred by co-runners, using which an accurate estimate of its slowdown can be obtained. One of the most crucial challenges while utilizing micro-experiments for estimating the slowdown is to determine when micro-experiments should be performed. We observe that interference does not change significantly within a single application phase. Thus, the problem of identifying when to perform a micro-experiment boils down towards identifying phases of applications at runtime while executing with co-runners. Triggering a micro-experiment on the application at each of its phases once allows the runtime to estimate co-runner interference with negligible overheads accurately.

To enable *Caliper*, one of the most significant challenges is to accurately, efficiently, and continuously detect not only phases within applications but also phases in application's co-runners. In this paper, we design a solution to identify all such phases by leveraging performance monitoring units (PMUs). Since each application has different sensitivities towards architectural resources, we identify the right set of PMU types that can differentiate phase changes across a wide variety of unknown applications. We perform cross-validation on these selected PMU types on a spectrum of application workloads to demonstrate generality. The contributions of this paper are as follows.

- **Phase aware micro-experiments:** We introduce a novel *phase aware interference estimation* technique using *micro-experiments* that is accurate, lightweight and can efficiently support multi-tenancy which can be deployed in clouds or datacenter environments.
- **Resilient phase detection:** We design a novel methodology called PMU scoreboarding that extracts the representative set of performance monitoring units for *detecting phase changes at multi-tenant execution scenarios*. Without a priori knowledge about the workloads, the extracted PMU types are effective in terms of detecting phase changes.
- **Real-world scenarios:** We evaluate our runtime system on real systems for a variety of applications including SPEC CPU2006 [29], NAS Parallel Benchmarks [12], Sirius-Suite [28] and DjiNN&Tonic Suite [27]. Moreover, we evaluate the effectiveness of the proposed runtime as we increase the number of executing applications contexts. In addition to that, we have also evaluated our technique on different microarchitectural subsystems to demonstrate its platform independent nature.

With *Caliper*, we are able to estimate the slowdown at multi-tenant execution scenarios accurately with a mean absolute error of 4% and negligible overhead of less than 1% for a broad spectrum of workload scenarios when executing 16 applications. Compared to state-of-the-art interference estimation techniques [16], our technique shows up to 5× more accuracy with 3× less overhead making it readily deployable in current and future datacenters.

The rest of paper is organized as follows. Section 2 describes the background and discusses the limitations of the prior study. Section 3 introduces the proposed design, Caliper. To achieve the design goals, Section 4 defines phase boundaries, and section 5 presents our technique identifying phase changes in co-location. Section 6 presents the experimental results. Section 7 describes the related work, and Section 8 concludes the paper.

2 BACKGROUND

In this section, we introduce key challenges that are present while co-locating multiple batch applications in multi-core systems. We then illustrate the state-of-the-art techniques that try to address these challenges and their limitations.

2.1 Multi-tenant Execution of Batch Applications

Modern computer systems host a wide range of applications of varying nature. These applications are broadly classified into two types (1) batch applications and (2) user-facing applications. Applications which are of batch type are throughput oriented and not user-facing. This type of application represents today's workloads that execute in datacenters and clouds. Consolidation of such applications to increase the resource utilization of the system is a common trend [9, 11]. On the other hand, another class of applications like memcached and web search is latency critical and hence is required to meet strict Quality of Service (QoS) guarantees. As a result, the consolidation of such latency critical applications with other applications is generally avoided as co-location will affect the latency of these applications significantly [2, 41, 70]. These applications are typically housed in private datacenters or run on dedicated machines that guarantee Service Level Agreements (SLAs).

Although the consolidation of batch applications onto a single server increases the resource utilization, it has a direct impact on individual application performance. State-of-the-art virtualization technologies try to provide performance isolation at some levels. Current hypervisors perform:

(1) Strict CPU reservations by disallowing sharing of CPU cores among different applications [13, 35].
(2) Statically partitioning memory and disk space among different applications [13, 35].
(3) Static partitioning of I/O and network bandwidth proportionally among applications using SR-IOV [32, 51].

However, applications are still slowed down mainly due to contention at the last-level cache (LLC) and main memory bandwidth. The resource contention at the LLC and main memory bandwidth increases the overall memory access latency, significantly slowing down the execution of different applications. Hence, it becomes critical to identify and gauge the slowdown applications are subjected to when they are housed at multi-tenant execution scenarios. As a major step towards solving this problem, prior approaches try to precisely estimate the amount of slowdown each application which is subjected to in multi-tenant execution scenarios [16, 58].

2.2 Limitations of the State-of-the-art Approach

Broadly, state-of-the-art approaches that try to estimate slowdown are classified into two different categories – **static approaches** that require a priori knowledge about the applications executing and **dynamic approaches** which can perform slowdown estimation for unknown applications. In this section, we enumerate the limitations of the state-of-the-art static and dynamic approaches that try to solve this problem.

2.2.1 Static Approaches. Prior static approaches like Bubble-Up [40] and Cuanta [24] have shown to be effective at generating precise performance predictions at co-located execution scenarios with high accuracy. However, there exist several primary limitations of the work, including requiring a priori knowledge of application behavior and the lack of adaptability to changes in application dynamic behaviors. These limitations restrict the possibility of

	Bubble-Up [40]	POPPA [16]	ASM [58]	FST [23]	Caliper
Low overhead	✓				✓
No additional hardware	✓	✓			✓
No offline profile			✓	✓	✓
Estimation error	7%	45%	20%	30%	4%

Table 1. Comparison between Caliper and other interference estimation techniques

deploying such static approaches for a variety of datacenter infrastructures which encounter unknown applications on a regular basis. (e.g., private datacenters and public clouds)

2.2.2 Dynamic Approaches. Another class of prior works, that does not require a priori knowledge, has attempted to estimate slowdown of applications due to shared cache capacity and/or memory bandwidth interference [16, 58, 68]. The most recent prior work by Breslow et al. [16] is software based that utilizes a technique called *POPPA*. The main motivation behind POPPA towards estimating slowdown is based on modeling interference as a ratio of solo and co-located execution performance. While co-located application performance can be directly measured at runtime, it is challenging to estimate solo performance of an application while running with co-runners simultaneously. Towards obtaining an estimate of solo performance, POPPA periodically pauses all co-running applications for a very short time except one application repeatedly at fixed time intervals as depicted in Fig 1a. The pause periods allow it to monopolize system resources and (briefly) match its solo performance. POPPA has several limitations as it suffers severely from low accuracy and high overheads especially as the number of application contexts increases.

On the other hand, there is a class of literature that has attempted to tackle the problem of estimating slowdown at runtime by utilizing novel hardware to track application interference among individual processor subsystems, which are taken together to model the overall interference of the applications [22, 23, 42, 58]. The most recent work by Subramanian et al. presents Application Slowdown Model (ASM). This work is based on the hypothesis that performance of each application is proportional to the rate at which it accesses the shared cache. Hence, in order to identify the shared cache access rate, it maintains an auxiliary tag store for each application, which tracks the state of the cache in a situation where the application would have been running alone. Every application that is co-located within the system utilizes this specialized hardware periodically in a round robin fashion to collect its corresponding shared cache access rates, which in turn is utilized by ASM to estimate its corresponding slowdown. One of the key limitations of ASM is that it requires additional hardware support precluding it from being used as a solution on existing commodity servers.

The combination of the poor accuracy, overhead, inadequate support for multi-tenancy, deployability, requirement of additional hardware support significantly limits the applicability of the prior approaches. Towards satisfying these shortcomings, we design a technique that can be deployed readily in production-grade datacenters. **Our technique can accurately estimate slowdown in executions scenarios that encounter a wide class of unknown applications**, unlike prior static approaches [24, 40] that require a priori knowledge of the executing applications. Table 1 presents a comparison between Caliper and sever other interference estimation techniques.

Later in Section 6.3, we experimentally evaluate each of these scenarios to illustrate the shortcomings of the prior dynamic approaches [16, 58]. Then, we show how our proposed phase aware interference estimation technique is able to estimate slowdown accurately with negligible overhead even when the number of simultaneously executing applications is up to 16 contexts as existing in modern datacenters.

3 OVERVIEW OF CALIPER

In this section, we describe **Caliper**, a runtime system for estimating interference at multi-tenant execution environments.

Goal. The design goal of Caliper is to accurately estimate the slowdown of an application at runtime. To achieve this, we need to gauge the performance of the application running with co-runners, $Perf_{(co-run)}$, as well as the performance of the application when it is running alone, $Perf_{(solo-run)}$ during runtime. Using these quantities, the slowdown of the applications can be easily estimated by the following Equation 1.

$$slowdown = Perf_{(co-run)}/Perf_{(solo-run)} \tag{1}$$

We have utilized Instructions Per Cycle (IPC) as the metric to quantify performance. $Perf_{(co-run)}$ from equation 1 is the IPC of the application during co-location and is directly measured when the application is running along with the co-runners during runtime. $Perf_{(solo-run)}$ is the solo execution performance of the application. IPC can be measured easily and cheaply on commodity processors. A wide body of prior interference estimation techniques utilizes IPC as their primary metric to quantify performance [16, 25, 58]. For even latency-sensitive applications, a prior study from Google leveraged the CPI (Cycles Per Instructions) metric as a performance indicator [69]. Although the metric may not be highly accurate for some applications, it is used to only guide the performance estimation.

Approach. The primary objective of this study is to be able to precisely estimate $Perf_{(solo-run)}$ even during the presence of co-runners at runtime. To achieve this goal, we introduce a software technique, called *micro-experiment*. **A micro-experiment is a short-lived runtime period for a few milliseconds during which an experiment is run to collect a measurement of interest.** Our runtime system performs micro-experiments by opportunistically pausing the execution of an application's co-runners for a small amount of time so that the resource contention is eliminated temporarily in the system. The result of such a micro-experiment represents an accurate estimate of the application's solo execution performance and this estimation along with $Perf_{(co-run)}$ (direct performance measurement of an application when it is run together with other applications) can be used as a basis to obtain the slowdown at runtime.

Challenges. To keep the cost of the estimation process low, we need to address a key challenge. A recent prior study that periodically pauses co-running applications to estimate the performance degradation has been shown to cause non-negligible overheads [16]. This is due to the following reasons:

(1) Frequent pausing can disturb forward progress of the applications due to the execution stalls.
(2) Pausing an application evicts its entries present in hardware caches, TLBs, BTBs, etc. This exacerbates the performance overhead problem.
(3) As the number of cores in a server increases, more applications (or VMs) can be housed in servers. Under such circumstances, periodically pausing every co-running application will increase the effective time for which individual applications is paused. Hence, a

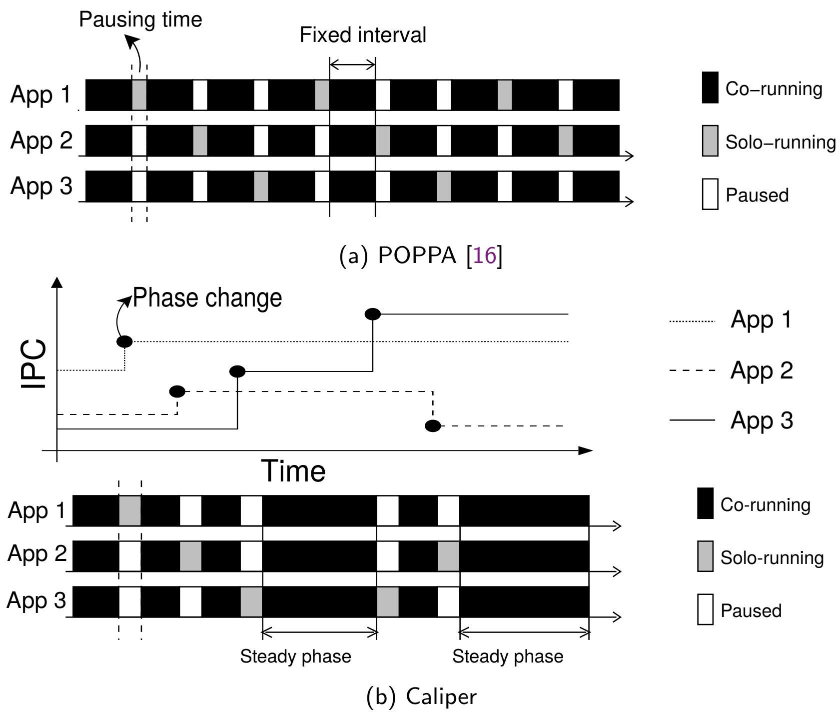


Fig. 1. Interference estimation by *POPPA* [16] vs. *Caliper*

naive technique like periodic pausing becomes an unsuitable solution for operation at scale.

Thus, it is essential to identify when micro-experiments need to be triggered. In this study, we overcome this challenge by utilizing phase boundaries as the triggers for conducting micro-experiments. The key observations that led towards utilizing phase boundaries as triggers are as follows. First, the execution behavior of applications does not drastically change within a single phase. This means that we do not need to estimate slowdown by performing micro-experiments within a steady phase. Second, we observe that the number of phase changes is not large in most applications, as also observed by previous works [21, 26, 56]. The majority of applications have a very few phases spanning over an execution time which range from a few minutes up to half an hour [46]. It gives us an opportunity to opportunistically conduct our micro-experiments technique so that we are able to avoid excessive pauses for the common case where applications have very few phase changes. Keeping these observations in mind we utilize a continuous monitoring system that performs online phase detection identifying phase changes for applications during runtime.

Fig 1b illustrates how Caliper estimates the slowdown by using micro-experiments. Whenever there is a phase change, we perform a micro-experiment by pausing all the co-running applications giving an opportunity for the un-paused applications to eliminate the resource contention. Then, we are able to measure $Perf_{(solo-run)}$ for the application without the resource contentions. However the most recent work that tries to estimate slowdown during runtime [16], pauses the co-running applications in a periodic fashion as shown in Fig 1a. We have conducted micro-experiments using 75 milliseconds as a pause period. The parameter is empirically determined in our testbed to monopolize architectural resources during that time. Section 6.2 talks in detail about the choice of our pause period. As a result, we can estimate the slowdown with negligible overheads of less than 0.5% for most of the situations. We will discuss the parameter sensitivity in the evaluation section.

While performing micro-experiments, our runtime estimates $Perf_{(solo-run)}$ of an application at every phase boundary. We aggregate the estimation of slowdown at every these individual phases of the application to calculate the slowdown for the entire execution of the application as shown by Equation 2.

$$Perf_{(solo-run)} = \frac{IPC_{(1)} \times T_{(1)} + IPC_{(2)} \times T_{(2)} + ... + IPC_{(n)} \times T_{(n)}}{T_1 + T_2.... + T_n} \tag{2}$$

where, $Perf_{(solo-run)}$ is the estimated IPC of solo execution of an application, $IPC_{(i)}$ is estimated IPC of solo execution of the application during phase i, $T_{(i)}$ is the time for which the application remains in phase i and n is the total number of phases in the application.

4 APPLICATION PHASE BEHAVIORS

In this section, we describe phase behaviors of applications in multi-tenant execution environments. Traditionally, phases can be defined as intervals within the execution of a program with similar behavior [26]. Phase changes typically manifest themselves as observable changes in execution behavior of applications. Although there have been many efforts to detect phase changes of a single application via performance monitoring units (PMUs) [21, 26, 31, 56], it is challenging to precisely identify phase boundaries in multi-tenant environments. This is because the PMU-based measurements of individual applications in multi-tenant environments are affected by the behavior of co-running applications. Prior techniques are unreliable when multiple applications are simultaneously running and hence cannot be directly applicable to our runtime system.

4.1 Two Classes of Phase Changes

As a first step towards detecting phase changes in co-located environments, we taxonomize phases detected by PMUs (e.g., as shifts in an application's CPI) as falling into one of two classes – *endogenous* phase changes that result from an application's innate behavior and *exogenous* phase changes that result from co-running applications. Thus, the goal of our runtime system is to accurately identify endogenous phase changes while minimizing the detection of exogenous phase changes. This is critical as exogenous phase changes are false positives incurring unnecessary micro-experiments. It results in increasing the overhead of our runtime system. In the next subsection, we investigate the causes of exogenous phase changes in further detail.

4.2 Characteristics of Exogenous Phase Changes

To study the characteristics of exogenous phase changes, we observe PMUs when an application is executing along with its co-runners. Through these observations, we identify two critical reasons contributing to exogenous phase changes.

Fluctuation. PMU-based measurements of a single phase are a set of discrete, time series based, numerical quantities that lie between a range possessing minuscule variation as shown in Fig 2 (a). However, in the presence of co-runners, PMU-based measurements belonging to a single phase of the same application fluctuate a lot. In such scenarios, some of the PMU-based measurements lie in the range of a different phase, making it challenging to

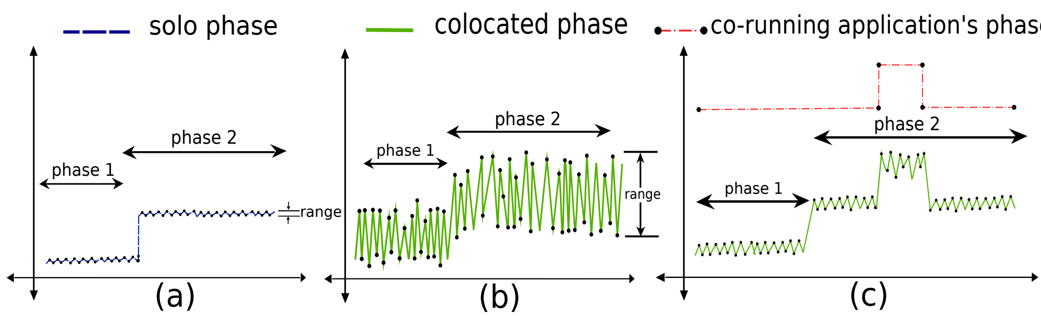


Fig. 2. (a) Solo Execution of application. (b) Fluctuations in PMU type during co-location. (c) Co-phase interference during co-location

determine phase boundaries. Fig 2 (a) represents the execution of an application when it is running alone. Fig 2 (b), represents the execution of an application when it is executing along with a co-runner. From Fig 2 (b), we can clearly see that some PMU measurements from phase 1 lie in the range of the PMU measurements from phase 2 and vice versa. This makes it challenging to identify phase boundaries. We have observed this phenomenon especially with PMU measurements corresponding to micro-architectural entities like last-level cache misses that are shared by multiple cores.

Co-phase interference. Phase changes in one application can cause changes to other co-running applications. We call this phenomenon as co-phase interference. Fig 2 (c), again represents the execution of an application when it is executing along with a different co-runner. From Fig 2 (c), we can clearly see that the change in PMU measurements corresponding to co-phase interference is difficult to be distinguished from endogenous phase changes.

Our goal here is to build a robust **phase aware online runtime system** that detects endogenous phase changes while minimizing the detection of exogenous phase changes. This is because triggering micro-experiments during exogenous phase changes is undesired as they will result in increasing the performance overhead due to pausing of co-runners. In some situations when interference is strong enough, our phase aware online runtime system triggers phase changes even for exogenous phase changes. This could potentially increase the overhead of our system by triggering frequent micro-experiments. However, the occurrence of such events are very infrequent which is evident from the negligible overhead incurred by our system as shown in Section 6.2.

5 IDENTIFYING PHASE CHANGES DURING CO-LOCATION

The primary goal of Caliper's phase detection approach is to detect endogenous phases (true positives) while ignoring exogenous phases (false positives) at runtime. For this purpose, we propose a PMU-based mechanism which identifies the best PMU that can be utilized for phase detection. Identifying the best PMU types is an offline step that is undertaken once. We then utilize the identified PMUs to detect phase changes during runtime. This is an online step that utilizes a continuous monitoring infrastructure.

For the offline step, we first try to identify the representative PMU types which accurately detect every single endogenous phase change while neglecting exogenous phases. In addition to that, the extracted PMU types should be generic. In other words, it should be able to detect endogenous phase changes even for an unknown application whose phase behavior has not been witnessed before. For this purpose, we first assess each PMU type, to detect

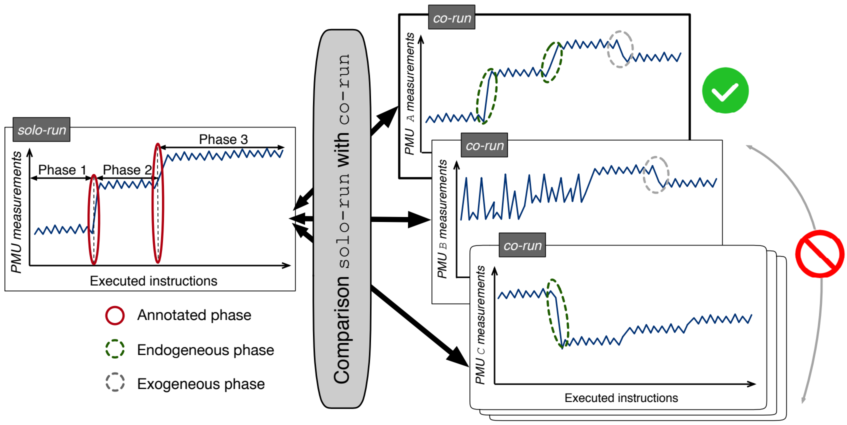


Fig. 3. Comparing phases during co-located execution with phases present in solo execution

phase changes for a training set of applications. We then **cross validate** to examine its ability to detect endogenous phases and ignore exogenous phases for unknown applications. This determines the generality of each PMU type. Based on the ability of each PMU type, we choose the best PMU type.

The initial step in this offline process is to carefully choose our training set of applications to cover a wide range of contentiousness, sensitivity and phase changing attributes [60]. The list of training applications is shown in the first column of Table 2. We use `astar` as our training co-runner which is cross-validated in our evaluation under section 6. The application `astar` from SPEC CPU2006 is known to be both contentious and to have numerous and rapidly changing phases [60], which can train our model to be resistant against both fluctuations as well as co-phase interference. With these pointers, we undertake the following three-step approach to extract the set of PMU types that can be utilized for phase detection.

(1) Comparing PMU measurements during co-run with solo execution. We execute the training set of applications alone to obtain PMU measurements during solo execution. We manually annotate the endogenous phases present in each of the training set of applications.

We then collect PMU measurements for each application present in training set during co-location. By using the PMU measurements during co-location, we verify for each PMU type its ability to detect endogenous phases by comparing the timestamps corresponding to the actual phase changes that happen during solo execution (from the annotated phases during previous step). This process is illustrated in Fig 3 as we observe that the measurements for PMU `A` detect all the two endogenous phases present which are confirmed by the annotated solo execution of the application. However, the measurements for PMU `B` could not detect any endogenous phase changes. It just detects an exogenous phase change which is not desired. With the PMU `C`, it detects only an endogenous phase chase, but misses the other endogenous phase. So, the PMU type `A` is resilient for the application to detect phase changes in multi-tenant environments. We performed above process for 18 different PMU types.

(2) Obtaining PMU scoreboard. We then quantify the effectiveness of each PMU type that was successful in identifying phase changes during the previous step (1). This quantification helps in selecting the best PMU type that detects every possible phase change

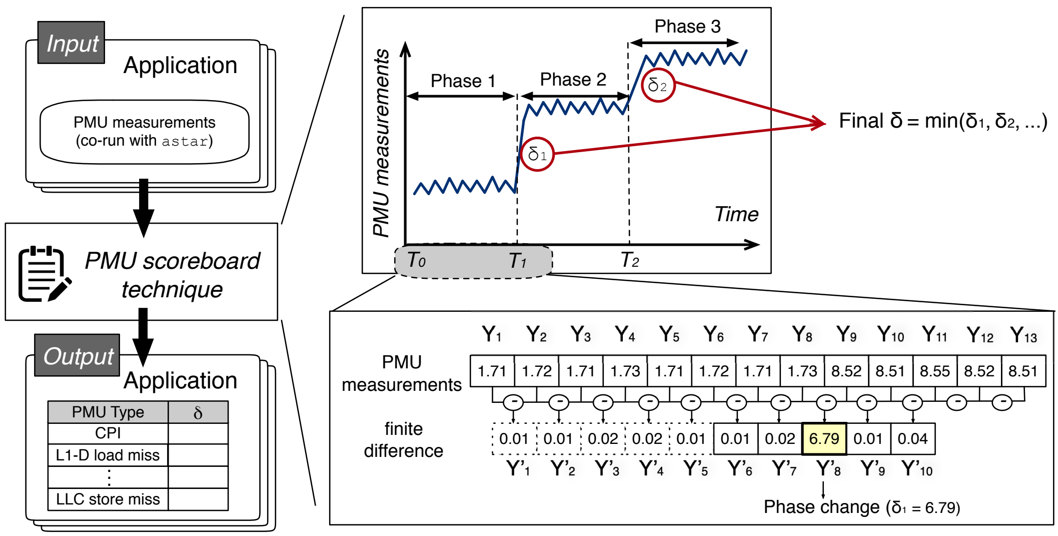


Fig. 4. Overview of PMU scoreboard technique

present in the system. This is done by obtaining the PMU scoreboard which will be discussed in detail at Section 5.1.

(3) Selecting the final set of PMU types. From observing the best PMU type for every single application present in the training set, we obtain a single set of PMU type(s). Those PMU types can be utilized to detect phase changes across a diverse class of application. We describe this step in the Section 5.2.

5.1 Obtaining PMU Scoreboard

The motivation of PMU scoreboarding is to quantify the effectiveness of each PMU type. Using this quantification, we obtain a common set of PMUs that can work effectively towards identifying phase changes. Our PMU scoreboarding quantifies PMU types by gauging how steep change in PMU measurements are at each phase boundary. We use a technique called step detection to quantify steepness at each phase boundary. Fig 4 shows the overall flow for obtaining the PMU scoreboard.

Inputs. Application and training dataset of time series PMU measurements during co-location.

Output. Threshold of separation (δ, described below) quantifying the steepness of a PMU type at phase boundary for an application.

Objective function. To quantify the effectiveness of a PMU type we assess the steepness magnitude expressed by PMU measurements during phase change (higher variation means PMU type distinguishes phase boundaries significantly better).

Methodology. The steepness is obtained by performing the step detection methodology. Step detection scheme is a process of finding abrupt changes in a time series signal. Internally step detection uses a technique called finite difference method for identifying abrupt changes.

5.1.1 Step Detection by Finite Difference Method. The fundamental hypothesis of finite difference method for identifying abrupt changes is based on the fact that the absolute

Workloads	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

difference between subsequent time-series measurements is very high at the exact point where the abrupt changes occur. Phase detection merely translates into Identifying the exact point where that particular abrupt change has happened.

Mathematically, the finite difference of a time series signal is the rate of change in the individual elements. We implement the finite difference method by performing pair wise difference of subsequent elements present in the time series using the following formula :-

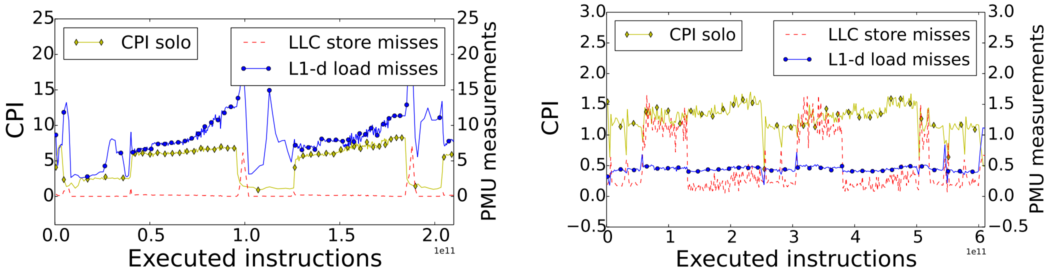
$$Y' = \frac{Y_{j+1} - Y_j}{\Delta T} \qquad\qquad Y'_j = Y_j \ (for\ 1 < j < n-1)$$

where Y_j is the j^{th} points present in the time series, n being the number of points, ΔT being the difference between the number of timestamps for time series values. The result highlights the drastic change by showcasing a high value at the point where phase changes. Figure 4 clearly illustrates this where we can see a sharp increase in the PMU measurement at time T_1 (at the point Y_9). Its corresponding finite differential value is very high at point Y'_8. Hence, the result of finite difference method is a set of differentials similar to the Y' points shown in Figure 4.

The subsequent step is to distinguish Y'_8 from the other Y'_j points. For this purpose we use a moving window approach. We utilize a window size of 5 based on our observations that a single phase lasts at least for 5 seconds [18, 30, 31, 36, 52]. Our continious monitoring infrastructure collects measurements once every second similar to most state of the art approaches [16, 40, 68]. At every point in the moving window we obtain the mean and standard deviation of the current window. If the latest element in the window is three standard deviations below or away the mean, then we conclude that there is an abrupt change at that particular time [1, 37]. Finally, we obtain all such abrupt changes (δs). The lowest numerical value of each such abrupt change obtained by step detection is returned as the threshold of separation δ for a PMU type that is being utilized to perform phase detection for an application. For the example given at Fig 4, the value of δ is the minimum of the value of δ_1 and δ_2. This delta value becomes useful to rank individual PMU types which is explained in the next section.

5.2 Ranking and Selecting PMU Types

In order to choose appropriate PMU types for identifying endogenous phase changes, we rank PMU types for every single application using the δ value (threshold of separation) obtained from the PMU scoreboarding technique. From that, we choose the PMU type that



(a) `mcf` (b) `milc`

Fig. 5. Phase changes triggered by PMU types when running with `astar`. Single PMU type is insufficient to detect phase changes

is capable of detecting endogenous phases across all the applications. In this paper, we have shown the top 3 PMU types in Table 2 for each application that are ranked using the δ value.

However, an observation from our training experiments whose results as depicted in Table 2 shows that no single PMU type can detect phase changes across all the training set of applications. In other words, there can be a situation where an architectural resource that can detect phase changes on an application could fail to detect phase changes completely on a different application. We illustrate this hypothesis based on a real-world example.

Fig 5 shows an example where a single PMU type will not be able to identify phase boundaries across two different applications. Each application requires different PMU types to precisely detect phase changes. In other words, `mcf` requires *L1-d load misses* while `milc` requires *LLC store misses* and vice versa fails. The x-axis indicates the cumulative number of instructions executed as time progresses. The left y-axis featured as yellow (diamond) line shows the CPI of the applications when running alone and the right y-axis and the blue (circle) and red (dashed) line show the selected hardware performance monitors for the application when running with three instances of `astar` as co-runners.

From Fig 5 (a), we find out that the PMU type, *L1-d cache load misses*, can effectively detect phase changes of `mcf` in co-located environments. This is not true for the application `milc` as the same PMU type *L1-d cache load misses* fails to detect phase changes as shown in Fig 5 (b). These results motivate the need for multiple PMU types to capture phase changes across a variety of applications. To achieve this, we undertake an approach where we observe a set of architectural resources (*CPI*, *LLC store miss*, and *L1-D load miss*) in contrast to a single resource. Moreover, to avoid missing endogenous phase changes, we use a conservative approach to trigger a *micro-experiment* even if one of the PMU types out of the three detects a phase change. This is because failing to detect endogenous phase changes will significantly reduce the accuracy in estimating IPC of solo execution. On the other hand, predicting a non-existent phase change causes only negligible overheads when the occurrence of such mispredictions is low. Additionally, the counters *CPI*, *LLC store miss*, and *L1-D load miss* covers charecteristics of applications that are both sensitive and insensitive towards shared cache contention. Hence, these counters prove to be effective in detecting phase changes even for a wide variety of unknown applications irrespective to the nature of their inputs.

Processor	Microarchitecture	Kernel	Hypervisor
Intel Xeon E5-2630 @2.4 GHz	Sandy Bridge-EP	3.8.0	KVM-QEMU v2.0
Intel Xeon E3-1420 @3.7 GHz	Haswell	3.8.0	KVM-QEMU v2.0

Table 3. Experimental platforms

5.3 Using Selected PMU Types for Online Phase Detection

Online phase detection during runtime can be performed using the PMUs that we had identified during the offline step. Whenever an application is execution, our continious monitoring runtime infrastructure monitors each PMU type identified during the offline step. It performs online step detection on each PMU type to detect the presence of any significant variation in the PMU measurements.

Caliper can be summarized as a continious monitoring runtime system that estimates slowdown of an application during runtime accurately.Caliper performs *micro-experiments*, a short-lived experiment to collect a measurement of interest, by opportunistically pausing the execution of co-running applications for a small amount of time so that resource contention can be temporarily eliminated in the system. The result of such a micro-experiment represents an accurate estimate of the solo performance for the application in that small period.

Performing *micro-experiments* frequently causes huge execution overheads. Hence, it is essential to identify when micro-experiments need to be triggered. In this study, we overcome this challenge by utilizing phase boundaries as triggers for conducting micro-experiments. This is because the execution behavior of applications does not drastically change within a single phase. Hence a single micro-experiment for a phase is sufficient to characterize the execution behavior of an application for that phase. Adding to that, the number of phase changes is not many in most applications. We utilize Performance Monitoring Units (PMUs) to detect phase changes during runtime. We perform offline analysis on training data to identify the best PMU types. Our online runtime system uses those PMU types during runtime to detect phase changes.

6 EVALUATION

6.1 Methodology

Infrastructure. We evaluate Caliper on two commodity multicore systems summarized in Table 3. We use Linux KVM as the hypervisor and run applications on virtual machines (VMs) [35] because running virtual machines is a standard way for cloud providers to isolate infrastructure among different customers. Hence our infrastructural setup consists of co-locating multiple virtual machines (VMs) where each VM belongs to a different user.

Each virtual machine has 4GB of main memory and 16GB disk. We use the Ubuntu 12.04 distribution as guest operating systems with Linux kernel 3.11.0. There is no change in the execution characteristics of the applications while executing them using virtualized environments. We take advantage of `perf` tool to collect hardware performance monitors while observing applications.

Applications. To evaluate the effectiveness of our technique, we use the benchmarks from *SPEC CPU 2006* [29] with `ref` inputs, *NPB - NAS Parallel benchmarks* [12]. In addition to that, we execute emerging applications from *SiriusSuite* [28] and *DjiNN&Tonic suite* [27] in batch mode. Sirius suite and DjiNN & Tonic suite contain a class of applications

Benchmarks	Class of applications	AWS use cases [5]
Sirius Suite	Machine learning	NTT Docomo (voice recognition) [6]
DjiNN & Tonic	Deep neural network	PIXNET (facial recognition) [8]
SPEC 2006	General purpose & Scientific	Penn State [7]
NPB	Parallel computing workloads	NASA NEX [11]

Table 4. Benchmark used in evaluation

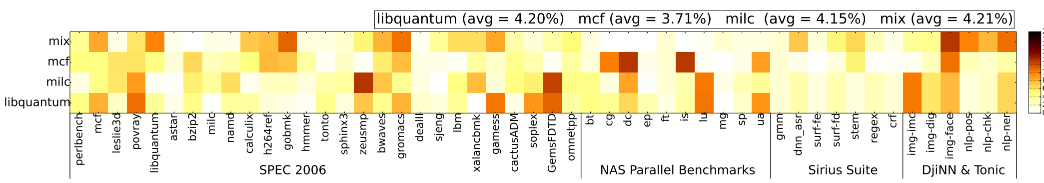


Fig. 6. Accuracy (in percentage error) of SPEC CPU2006, NAS Parallel Benchmarks, Sirius Suite and Djinn&Tonic suite while estimating slowdown when 4 applications are co-located.

which implement state-of-the-art machine learning and computer vision algorithms. It has been a common trend to execute such applications in modern public clouds where multiple applications are oversubscribed in the same server [5–8, 10, 43, 45]. We can clearly see that the benchmark suites that we have utilized to evaluate Caliper are similar to the applications that are being executed in state-of-the-art public cloud computing environments (e.g., Amazon web services [67]). Table 4 enumerates the benchmark suites, application domain and the respective use cases for the applications present in these benchmark suites in a public cloud execution environment like Amazon Web Services (AWS). Also, *SiriusSuite* [28] and *DjiNN&Tonic suite* [27] have stemmed into a startup that builds conversational artificial intelligence systems for banking sector [3].

6.2 Caliper – Accuracy and Overhead

In this section, we evaluate the efficacy of Caliper. We discuss the accuracy in estimating slowdown by Caliper and its overhead experimentally. Accuracy calculated by comparing the estimated slowdown from our runtime system with the actual slowdown, a metric that is consistently followed by existing literature that focuses on estimating slowdown [16, 22, 23, 58].

Fig 6 shows the accuracy when Caliper is trying to estimate slowdown when 4 applications are co-located on a single server. The experimental setup here consists of four broad execution scenarios each based on the type of co-running application that we have taken into consideration represented in the y-axis of Fig 6.

Single vCPU. The single-threaded benchmarks from SPEC CPU 2006, Sirius Suite, Djinn&Tonic are evaluated where for each experiment the observed application executes in a single VM pinned to a single vCPU. The PMU-based measurements are collected from the vCPU at which the application is executing which directly corresponds to the performance of the application.

Multiple vCPU. The multi-threaded benchmarks from NPB are evaluated where for each experiment the observed application executes in a single VM pinned to two vCPUs. Here,

	SPEC		NPB		Sirius	
	Overhead (%)	Phase changes (per min)	Overhead (%)	Phase changes (per min)	Overhead (%)	Phase changes (per min)
main-app	0.65	1.58	0.35	0.38	0.25	0.20
colo-app	0.74	0.91	0.45	0.75	0.44	0.80

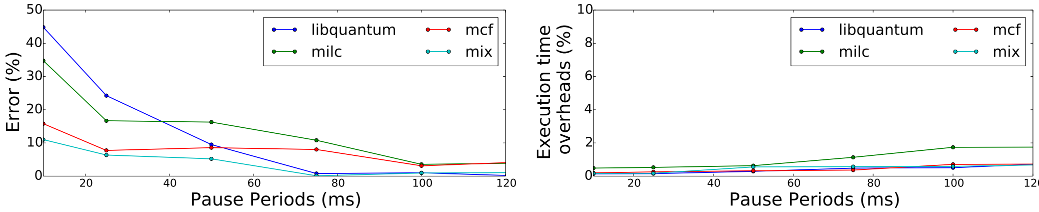
Table 5. Execution time overhead and number of phase changes

the performance of the application is the cumulative values of the PMU-based measurements obtained from each vCPUs at which the application is executing.

Individual cells in Fig 6 present the difference (error) in the estimated slowdown versus the actual slowdown (Light is good and dark is bad). For each experiment, we execute 3 instances of a single type of co-runner `libquantum`, `mcf` and `milc`, simultaneously along with 1 instance of the application on the x-axis. The mix co-runner is a mix of 3 different co-runners, `libquantum`, `mcf`, and `milc`, alongside the applications on the x-axis. We have used `libquantum`, `mcf` and `milc` as co-runners as from our experiments and through prior work [60], we found out that these were the top 3 applications that exhibit significant activities towards shared architectural resources including last-level cache and memory bandwidth. Hence, accurately estimating slowdown during the presence of such co-runners was a big challenge for us[60]. Our experiments to estimate the accuracy of slowdown and runtime overhead takes into account all the 4 applications executing in the system. We run each benchmark three times and take the mean to minimize run to run variability. We check to see if there is any phase change every second owing to the observation that phases are consistent for a few seconds. During every phase change, micro-experiments are performed for 75ms to eliminate resource contention during observation. We obtain the value 75ms empirically by performing a sweep for different quantities optimizing for reduced overhead and increased accuracy. Details will be discussed later.

Accuracy. From Fig 6, we can see that Caliper shows very low error rates across all the applications even when running with multiple instances of cache contentious co-runners like `libquantum`. The average error rate when co-locating with such contentious co-runners is around 4%. We observe that 95% of our applications have errors less than 10% and the worst case error is 12% in our technique, whereas the worst case error of prior techniques is up to 60% (details presented in Section 6.3 of evaluation). We also observe that the error in estimating interference using Caliper remains consistent regardless of the nature of the co-runners. This is indicative of two things (1) accuracy with respect to detecting phases (2) precision of micro-experiments in detecting per phase interference. In the next section, we discuss the importance of having a robust phase detection methodology and its impact on the accuracy of of estimating interference.

Overhead. To enable Caliper on production systems, we have to achieve low overheads so as to minimize the interference to running application on the servers. Table 5 indicates the overhead that is incurred by Caliper while estimating slowdown. We evaluate the overhead at the same experimental setup under which we had evaluated accuracy. From Table 5, we can clearly see that the overhead of the main observed application, as well as the average overhead of the co-running applications, remain less than 1% in most of the cases. On average, the overhead of Caliper's runtime system is around 0.6%. Similarly, we also see that the number of phase changes per minute is also very less. On average, there is a single



(a) Accuracy (b) Overheads

Fig. 7. Accuracy and overheads for Caliper under different pause periods.

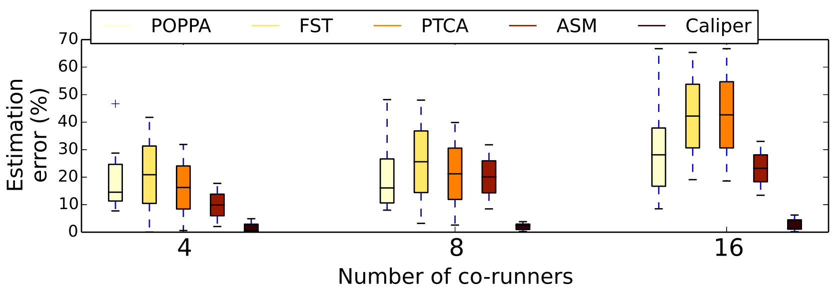


Fig. 8. Estimation error: Caliper vs. state-of-the-art software (POPPA [16]) and hardware (FST [23], PTCA [22], ASM [58]) techniques for estimating interference

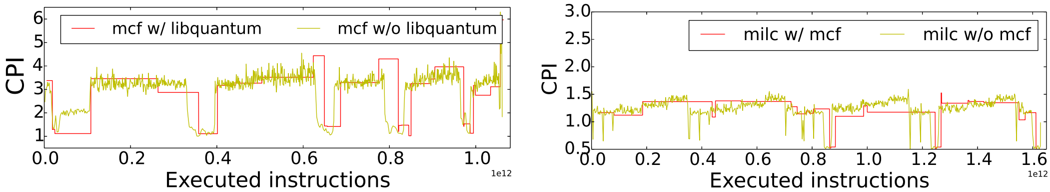
phase change per minute. This indicates that each application is paused for a few hundred milliseconds every minute making the overhead extremely negligible.

Sensitivity of the pause period. Towards obtaining an optimal pause period for operating Caliper, we performed a sensitivity study. The results of this study is shown in Fig 7. From figures 7a and 7a, we clearly observe two trends. First, the accuracy of estimating slowdown increases as the pause periods increase up to 75ms. Then there is no benefit in increasing the pause periods. Hence we have utilized 75ms as an optimal pause period for our mechanism. Second, the overheads do not change drastically as we increase the pause periods. This is because Caliper's frequency at which it pauses the co-runners is too low causing negligible impact in the execution time overheads.

6.3 Comparison with Prior Work

Accuracy. Fig 8 shows the accuracy of Caliper as compared to the accuracy of POPPA [16], FST [23], PTCA [22], and ASM [58] for the benchmarks present in SPEC CPU 2006, NPB, Sirius suite and Djinn&Tonic suite. POPPA works by periodically pausing all co-running applications except one for a very short time at fixed time intervals. The aggregated performance of the applications during the pause periods is the key measure by which slowdown is estimated. Through these experiments, we can see that the estimation error is much lower for Caliper than POPPA [16]. The mean error of POPPA is 13.23%. On the other hand, our technique shows much lower error rates averaging around 3.77%.

It is challenging to estimate the slowdown when running with contentious co-runners as they quickly pollute the shared last-level cache and excessively use the shared memory bandwidth. One of the main reasons for POPPA's poor accuracy is that the pausing time (3.2ms) is too short to capture solo performance of an application. This is because the shared cache would not be warmed up to be containing the entire working set of the application



(a) `mcf` (error 0.51%)

(b) `milc` (error 2.52%)

Fig. 9. Phase level behavior of Caliper for `mcf` and `milc` when running with co-runners, 3 `libquantum` (a) and `mcf` (b), respectively. Micro-experiments are triggered effectively at phase boundaries.

which is to be measured. As a result, the measured application would spend most of its pausing time filling in the shared cache, giving much less time to observe how the application performs when it monopolizes computing resources. On the other hand, Caliper performs micro-experiments which pauses co-runners only when discovering phase boundaries. This enables us to observe the solo execution performance for a longer time without worrying much about the overhead caused due to pausing for additional time. Hence, we are able to achieve high accuracy in estimating slowdown at runtime.

We also observed that the state-of-the-art hardware enabled approaches towards estimating slowdown [22, 23, 58] showed a high error rate. Just like the other software approaches, state-of-the-art hardware enabled approaches utilizes cache access rates of applications during solo execution time to determine slowdown of an application. Cache access rates of applications during solo execution is again obtained by periodically pausing co-running applications in a round robin fashion. Hence the limitations of the prior software approaches hold good for the hardware approaches too. The mean errors of POPPA, FST, PTCA, and ASM are 11.04%, 28.28%, 38.42% and 9.98% respectively. From these results, we were able to see that Caliper can outperform even the state-of-the-art hardware enabled approaches present in the literature.

Multi-tenancy. To evaluate the effectiveness of the state-of-the-art hardware or software based approaches and Caliper towards supporting multiple tenants, we increase the number of executing application contexts to 8 applications and 16 applications. Fig 8 shows the average accuracy of Caliper as compared to POPPA for SPEC CPU2006 and NPB when co-locating with libquantum. We can see that Caliper's accuracy is around 3.95% when co-locating with 16 applications in contrast to POPPA [16], FST [23], PTCA [22] and ASM [58] whose error is around 22%, 40%, 41% and 19% respectively. The low accuracy of the prior techniques is because as the number of co-runners increases, the shared cache becomes much more polluted due to the contention. POPPA even on such situations pauses for the same amount of time which is too less for the shared last-Level cache to warm up so as to exhibit the performance corresponding to solo execution. Hence, its slowdown estimation becomes highly inaccurate. Similarly, hardware techniques perform sampling in a round robin fashion using their proposed specialized hardware whose pressure increases as the number of co-running application increases. However, Caliper utilizes a phase-aware approach that performs micro-experiments at adequate amounts of time during the right time to capture the solo execution characteristics of every phase accurately.

Phase analysis. Now, we try to visualize the effectiveness at which Caliper utilizes its a robust phase detection technique in order to achieve high accuracy and low overhead

	milc (4)	gobmk(1)	hmmer(1)	perbench(1)	astar(7)	namd(1)	pos(1)	chk(1)	calculix(118)	dealII(132)
libquantum	0	0	0	0	2	0	0	0	257	98
mcf	6	0	2	0	3	1	1	1	412	49
milc	5	0	1	0	2	0	0	0	353	33
mix	2	0	1	0	2	0	1	1	251	98

Table 6. Number of false positives incurred in Caliper runtime system. For each benchmark, the number of endogenous phases of the solo-run is represented in parentheses. (E.g., milc (8) means that milc has 8 endogenous phases.) First column contains co-runners.

in estimating slowdown. Toward illustrating this, we analyze the phase level behavior of a selected set of phasy applications to show Caliper's capability towards performing micro-experiments at every single phase change.

Firstly, we select two applications, `mcf` and `milc` to analyze the execution behaviors. These applications possess a significant number of phase changes. As co-runners, we use `libquantum` and `mcf`, respectively. Fig 9 (a) shows the execution behavior of `mcf` with respect to time. In each graph, the yellow line depicts the measured CPI of the application when running alone and the red line shows the CPI estimated by Caliper when the application is running with 3 instances of `libquantum` or `mcf`. We can see that Caliper can effectively trace the phase changes. The closer the red line is to the yellow line, the smaller the error. The error in estimating slowdown is 0.51% over the entire run. For `milc`, Fig 9 (b) presents that our technique can effectively trace all of its phase. The error while estimating slowdown is 2.52%.

Secondly, we evaluate how many false positives are incurred by our technique. Table 6 illustrates the number of falsely detected phase changes by Caliper. The first row shows the benchmarks for which we have evaluated this experiment. We have shown only ten benchmarks in this table in the interest of space constraints. The numbers present in the bracket after the benchmarks show the endogenous phase counts (true positives) when running alone. The first column shows the co-runners along which the benchmarks present in the first row have been evaluated. From our experiments, we observed that the results for most of the benchmarks were similar to `gobmk`, `hmmer`, `namd`, `pos`, `chk`. There was just one phase, and Caliper was able to detect that phase. Additionally, detecting false phases were a rare occurrence consuming negligible overheads. However, we had a few interesting observations for the benchmarks `calculix` and `dealII`. The phases of these applications are very irregular and contain spikes once every few seconds. Each of these situations where spikes occur triggers a phase change resulting in a larger number of false positives. Additionally, another interesting observation from our experiments was that there were more false positives when `mcf` was a co-runner. This is because `mcf` has many phase changes introducing many more false positives due to co-phase interference. However, the frequency at which Caliper's runtime system triggers phase changes is so low that our overhead remains lesser than 1% for most of the time.

Overhead. Fig 10 compares the overhead up to 16 application contexts for Caliper and the state-of-the-art software approach POPPA. We can clearly see that as the number of application context increases, the overhead of Caliper increases negligibly. However, this is not the case for other software approaches. This is due to the fact that, POPPA performs periodic pauses. As more applications are co-located, the effective time for which applications are paused increases as POPPA need to pause every application for the same amount of time for each of the co-runner. However, Caliper pauses applications only during phase changes

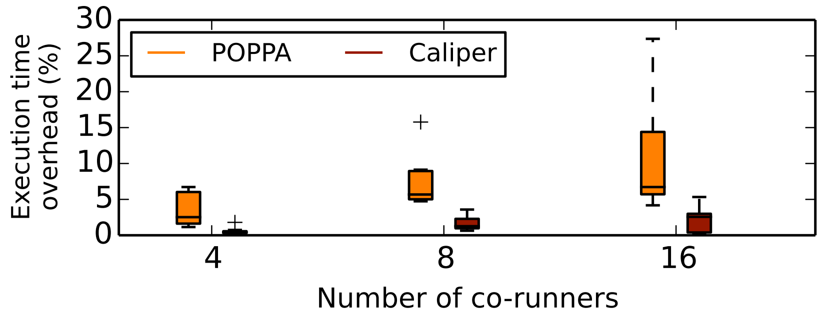


Fig. 10. Overhead: Caliper vs. POPPA

(which are comparatively infrequent). Hence, the overhead incurred by Caliper's runtime system is lesser by an order of magnitude.

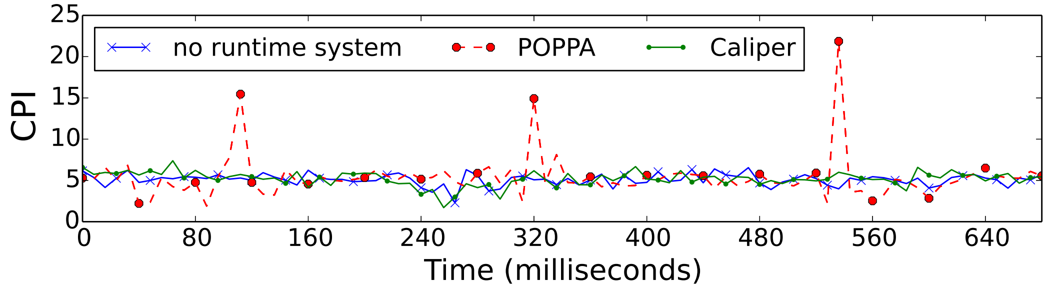
Fig 11 illustrates the reasons behind POPPA's higher execution time overhead. Fig 11 (a) compares the performance of POPPA and Caliper in an environment without any slowdown estimation runtime system. We can clearly see that the increased execution time overhead of POPPA is due to the spikes present in CPI due to frequent pausing of co-runners periodically by POPPA to estimate slowdown. However, Caliper performs micro-experiments rarely (once every phase). Hence, there are no periodic spikes as seen in POPPA. Caliper's execution time overhead also is negligible.

We have experimentally verified the reasons for the increased overhead and is clearly shown in Fig 11 (b), (c), and (d), respectively. As POPPA performs periodic pauses, it incurs addition warmup overheads for the micro-architectural components present in the system. At the end of every pause period, the system refills the micro-architectural components (cache, branch target buffer, TLB etc.) that would have been flushed during its pause period. This gets translated directly into increased execution time overhead.

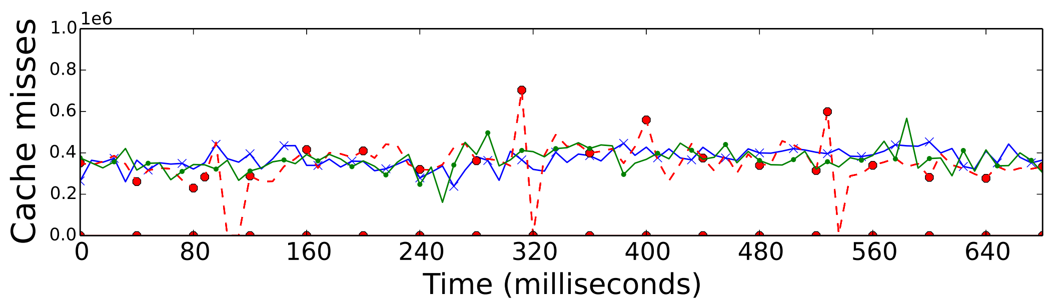
Fig 11 (b), (c), and (d) illustrate the underlying causes for this phenomenon. From Fig 11 (b), we can see that the cache misses increases whenever POPPA pauses co-runners in the system. However, it remains on unaffected for Caliper reasoning out its negligible overhead. Similarly, from Fig 11 (c) and (d), we can see that when POPPA frequently pauses applications, branch misses and TLB (transition look aside buffers) misses increases. This in on similar lines that micro-architectural components like branch target buffer (BTB), TLBs are flushed out frequently during pausing by POPPA. We can see that frequent pauses by POPPA increases the cache misses, TLB misses and branch misses by 11.6%, 5.7% and 7% respectively thereby increasing the runtime overhead of the execution of an application up to 10.5%. However, Caliper's overhead, as well as misses at the micro-architectural structures, remains less than 0.5%.

6.4 Leveraging Caliper for Fair Pricing in Datacenters

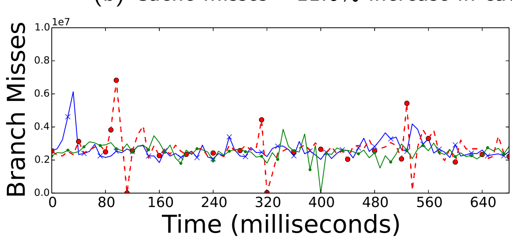
Infrastructure-as-a-service (IaaS) clouds primarily use a pay-as-you-go pricing model that charges users a flat hourly fee for running their applications on shared servers. Customers renting IaaS public clouds now have the capability to choose resource fragments at varying granularity in terms of the number of virtual CPUs, the amount of memory and storage size. Cloud service providers rely on virtualization to isolate resource fragments belonging to each customer. However, in light of significant potential for parallelism, cloud service providers co-locate applications belonging to different users. Since the last-level cache and DRAM



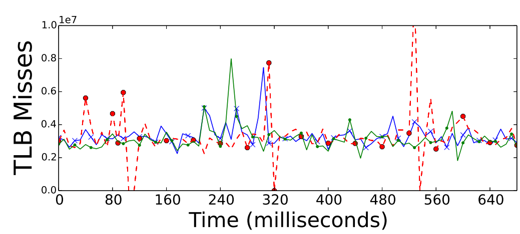
(a) Execution performance - 10.5% increase in execution time due to POPPA's runtime system



(b) Cache misses – 11.6% increase in cache misses due to POPPA's runtime system



(c) Branch misses – 5.7% increase in branch misses due to POPPA's runtime system.



(d) TLB misses – 7% increase in TLB misses due to POPPA's runtime system

Fig. 11. Performance of micro-architectural entities when POPPA's runtime systems are being executed

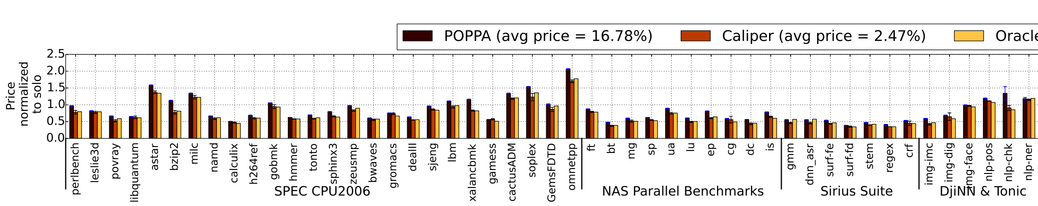


Fig. 12. Comparison of fairness in pricing by *Caliper* with *POPPA*

bandwidth remain shared among applications running within a single server, applications are slowed down as compared to when they run alone on the system. This increased execution time that the application is subjected to reflects directly on the price paid by the users under the pay-as-you-go scheme creating an unfair pricing scenario.

To enable fair pricing in public clouds, it is essential to estimate the performance impact that co-running applications have on an application. Identifying slowdown at runtime would be a very useful information in this regard as it would be an appropriate indicator of influence of co-runners on an application. Hence, such a scheme can be used as a critical substrate upon which any pricing scheme can be built. However, such a scheme is highly dependent on the accuracy at which fairness is estimated. Hence, achieving high accuracy in estimating slowdown becomes critical.

We compare the unfairness that is present while utilizing the hardware enabled approaches for pricing with our approach. We define unfairness as the price by which users are overcharged when they are executing their applications in IaaS public clouds. We use the pricing model proposed by Toosi et.al. [64] and apply the slowdown estimation techniques along with it so as to calculate the resultant price. From Fig 12, we can see that is able to price applications with $5X$ more fairness while pricing users using their slowdown model as compared to the POPPA technique proposed in Breslow et al. [16]

7 RELATED WORK

There have been many prior studies to detect performance interference in a variety aspects of architectural resources. We look first into the hardware enabled approaches and then address the prior work that utilizes system and OS level approaches for detecting interference.

Hardware techniques: There are several approaches that try to estimate slowdown due to contention in shared caches, memory controller and bandwidth. Nesbit et al. employed the network fair queuing model in the memory scheduler to meet the fairness [48]. Mutlu and Moscibroda focused on DRAM specific architectural features such as row buffers and DRAM banks [42]. They utilized memory scheduling techniques to ensure the fairness between multiple threads. Ebrahimi et al. extended the fairness problem in memory subsystems by including shared last level cache and memory bandwidth [23]. This work focused on the source incurring performance interference and proposed throttling mechanism by controlling injection rates of requests to alleviate the contention of shared resources. Suh et al. firstly discussed the cache partitioning scheme to efficiently use the shared resources [59]. Qureshi et al. proposed utility based cache partitioning technique to achieve high performance [53].

Software and systems approaches: There are many efforts introducing software frameworks and proposing the new designs of operating systems [24, 38, 40, 47, 50, 61, 68]. Q-Cloud measures the resource capacity for satisfying QoS in a dedicated server called as a staging server and then performs placement decisions based on choosing the right server that will be profitable to minimize interference [47]. To accurately estimate the performance interferences without profiling on a dedicated server, Bubble-up [40] and Cuanta [24] designed the synthetic workloads to understand the degree of interference when co-locating applications. 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. Their work focused on improving the utilization of shared caches through OS-level page allocation [57]. Zhuravlev et al. extended the CPU scheduler to alleviate the some of the interferences. The goal of this work is to schedule the threads by evenly distributing the load intensity to caches [71]. Blagodurov et al. proposed that the scheduler needs to consider the effects of NUMA [15]. Also, there are numerous prior studies to solve the contention problems such as shared last level cache and NUMA by scheduling virtual machines [4, 39, 54]. Tuncer et al. utilizes a machine learning approach to detect anomalies in HPC systems [65, 66]. They utilize the characteristics of applications that have executed before in the system to model performance anomalies. Zhang et al. identifies contentious application behavior by observing the application performance at runtime using CPI [69] for both latency-sensitive and batch applications running on datacenter. On the one hand, to detect application phases, there are several prior studies, requiring compiler support [18, 20, 52, 56] and utilizing PMU measurements [30, 36]. Isci et al. characterized the two different approaches for performing live runtime phase analysis [31] which motivates us to design Caliper. However, those prior studies could not be directly applicable due to the lack of consideration of multi-tenant environments.

8 CONCLUSIONS

In this paper, we introduce to estimate slowdown of applications that are co-located in multi-core systems that contend for shared cache and main memory. Caliper accurately estimates slowdowns using phase aware micro-experiment approach which utilizes system software tools like Performance Monitoring Units. We demonstrate the superiority of Caliper as compared to the state-of-the-art hardware enabled approaches. We conclude finally by illustrating the scenarios at which estimating interference at runtime would be useful by evaluating its applicability in one such scenario.

9 ACKNOWLEDGEMENTS

This work was sponsored by the National Science Foundation (NSF) under grants IIS-VEC1539011 and NSF CAREER SHF-1553485. Jeongseob Ahn was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1C1C1005166). We also thank Lavanya Subramanian for her insightful suggestions that helped improve this work.

REFERENCES

[1] [n. d.]. Oracle outlier. https://docs.oracle.com/cd/E40248_01/epm.1112/cb_statistical/frameset.htm?ch07s02s10s01.html

[2] [n. d.]. Private VS Public clouds which one to choose. http://www.logicworks.net/blog/2015/03/difference-private-public-hybrid-cloud-comparison/. Accessed: 2015.

[3] 2018. https://clinc.com/

[4] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-Ri Choi, and Jaehyuk Huh. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 19–19.

[5] Amazon. [n. d.]. Amazon web services User Case Studies. https://aws.amazon.com/solutions/case-studies/. Accessed: 2015-08-12.

[6] Amazon. [n. d.]. AWS Case Study: NTT Docomo. http://aws.amazon.com/solutions/case-studies/ntt-docomo/. Accessed: 2015-08-12.

[7] Amazon. [n. d.]. AWS Case Study: Penn State. http://aws.amazon.com/solutions/case-studies/penn-state/. Accessed: 2015-08-12.

[8] Amazon. [n. d.]. AWS Case Study: PIXNET. http://aws.amazon.com/solutions/case-studies/pixnet/. Accessed: 2015-08-12.

[9] Amazon Inc. [n. d.]. Amazon Elastic Compute Cloud(EC2).

[10] S. Avireddy, V. Perumal, N. Gowraj, R. S. Kannan, P. Thinakaran, S. Ganapthi, J. R. Gunasekaran, and S. Prabhu. 2012. Random4: An Application Specific Randomized Encryption Algorithm to Prevent SQL Injection. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. 1327–1333. https://doi.org/10.1109/TrustCom.2012.232

[11] AWS Case Study. [n. d.]. NASA/JPL's Desert Research and Training Studies: https://aws.amazon.com/solutions/case-studies/nasa-jpl/.

[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the*

1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91). ACM, New York, NY, USA, 158–165.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 164–177.

[14] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.

[15] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 1–1.

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

[17] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 17)*.

[18] Ghislain Landry Tsafack Chetsa, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf, and Georges da Costa. 2013. A User Friendly Phase Detection Methodology for HPC Systems' Analysis. In *Proceedings of the 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing (GREENCOM-ITHINGS-CPSCOM '13)*. IEEE Computer Society, Washington, DC, USA, 118–125.

[19] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 77–88.

[20] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. *SIGARCH Comput. Archit. News* 30, 2 (May 2002), 233–244.

[21] Ashutosh S. Dhodapkar and James E. Smith. 2003. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 217–.

[22] Kristof Du Bois, Stijn Eyerman, and Lieven Eeckhout. 2013. Per-thread Cycle Accounting in Multicore Processors. *ACM Trans. Archit. Code Optim.* 9, 4, Article 29 (Jan. 2013), 22 pages.

[23] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. 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)*. ACM, New York, NY, USA, 335–346.

[24] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. 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)*. ACM, New York, NY, USA, Article 22, 14 pages.

[25] A. Gupta, J. Sampson, and M.B. Taylor. 2014. Quality Time: A simple online technique for quantifying multicore execution efficiency. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. 169–179.

[26] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. *J. Instruction-Level Parallelism* 7 (2005).

[27] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. 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)*. ACM, New York, NY, USA, 27–40.

[28] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. 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)*. ACM, New York, NY, USA, 223–238.

[29] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[30] C. Isci, G. Contreras, and M. Martonosi. 2006. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 359–370.

[31] Canturk Isci and Margaret Martonosi. 2006. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques.. In *HPCA*. 121–132.

[32] Jithin Jose, Mingzhe Li, Xiaoyi Lu, Krishna Chaitanya Kandalla, Mark Daniel Arnold, and Dhabaleswar K Panda. 2013. SR-IOV support for virtualization on infiniband clusters: Early experience. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 385–392.

[33] R. S. Kannan, A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. 2018. Proctor: Detecting and Investigating Interference in Shared Datacenters. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 18)*.

[34] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 34.

[35] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, Vol. 1. Ottawa, Ontario, Canada, 225–230. http://linux-security.cn/ebooks/ols2007/OLS2007-Proceedings-V1.pdf

[36] J. Lau, S. Schoenmackers, and B. Calder. 2005. Transition phase classification and prediction. In *11th International Symposium on High-Performance Computer Architecture*. 278–289.

[37] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 49, 4 (2013), 764–766.

[38] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2014. Going Vertical in Memory Management: Handling Multiplicity by Multi-policy. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 169–180.

[39] Ming Liu and Tao Li. 2014. 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)*. IEEE Press, Piscataway, NJ, USA, 325–336.

[40] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. 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)*. ACM, New York, NY, USA, 248–259.

[41] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. 2011. Cloud Computing - The Business Perspective. *Decis. Support Syst.* 51, 1 (April 2011), 176–189.

[42] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 146–160.

[43] V. Nagarajan, R. Hariharan, V. Srinivasan, R. S. Kannan, P. Thinakaran, V. Sankaran, B. Vasudevan, R. Mukundrajan, N. C. Nachiappan, A. Sridharan, K. P. Saravanan, V. Adhinarayanan, and V. V. Sankaranarayanan. 2012. SCOC IP Cores for Custom Built Supercomputing Nodes. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 12)*.

[44] V. Nagarajan, K. Lakshminarasimhan, A. Sridhar, P. Thinakaran, R. Hariharan, V. Srinivasan, R. S. Kannan, and A. Sridharan. 2013. Performance and energy efficient cache system design: Simultaneous execution of multiple applications on heterogeneous cores. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 13)*.

[45] V. Nagarajan, V. Srinivasan, R. Kannan, P. Thinakaran, R. Hariharan, B. Vasudevan, N. C. Nachiappan, K. P. Saravanan, A. Sridharan, V. Sankaran, V. Adhinarayanan, V. S. Vignesh, and R. Mukundrajan. 2012. Compilation Accelerator on Silicon. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 12)*.

[46] Arun A Nair and Lizy K John. 2008. Simulation points for SPEC CPU 2006. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE, 397–403.

[47] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 237–250.

[48] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 208–222.

[49] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 219–230.

[50] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2013. 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)*. ACM, New York, NY, USA, 181–192.

[51] NIC Partitioning and SR-IOV. [n. d.]. NIC Partitioning and SR-IOV, Technology brief by cavium.

[52] E. Perelman, M. Polito, J. . Bouguet, J. Sampson, B. Calder, and C. Dulong. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 10 pp.–.

[53] Moinuddin K. Qureshi and Yale N. Patt. 2006. 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)*. IEEE Computer Society, Washington, DC, USA, 423–432.

[54] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng zhong Xu. 2013. Optimizing virtual machine scheduling in NUMA multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. 306–317.

[55] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (July 2010), 65–79.

[56] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*. ACM, New York, NY, USA, 336–349.

[57] Livio Soares, David Tam, and Michael Stumm. 2008. 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)*. IEEE Computer Society, Washington, DC, USA, 258–269.

[58] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, , and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. IEEE Computer Society, 13.

[59] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2002. 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)*. IEEE Computer Society, Washington, DC, USA, 117–.

[60] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2011. 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)*. ACM, New York, NY, USA, 12–21.

[61] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. 2013. 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)*. ACM, New York, NY, USA, 89–100.

[62] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2017. Phoenix: a constraint-aware scheduler for heterogeneous datacenters. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS 17)*.

[63] Prashanth Thinakaran, Jashwant Raj, Bikash Sharma, Mahmut T Kandemir, and Chita R Das. 2018. The Curious Case of Container Orchestration and Scheduling in GPU-based Datacenters. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 18)*.

[64] A. N. Toosi, R. N. Calheiros, R. K. Thulasiram, and R. Buyya. 2011. Resource Provisioning Policies to Increase IaaS Provider's Profit in a Federated Cloud Environment. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. 279–287.

[65] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. Coskun. [n. d.]. Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning. *IEEE Transactions on Parallel and Distributed Systems* ([n. d.]), 1.

[66] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2017. Diagnosing Performance Variations in HPC Applications Using Machine Learning. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 355–373.

[67] Wikipedia. [n. d.]. Amazon Elastic Compute Cloud. https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud. Accessed: 2015-08-10.

[68] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. 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)*. ACM, New York, NY, USA, 607–618.

[69] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 379–391.

[70] Yilei Zhang, Zibin Zheng, and M.R. Lyu. 2011. Exploring Latent Features for Memory-Based QoS Prediction in Cloud Computing. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. 1–10.

[71] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. 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)*. ACM, New York, NY, USA, 129–142.