Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads

Ran Xu
Purdue University
xu943@purdue.edu

Subrata Mitra
Adobe Research
subrata.mitra@adobe.com

Jason Rahman
Facebook
jprahman93@gmail.com

Peter Bai
Purdue University
pbai@purdue.edu

Bowen Zhou
LinkedIn
bwzhou@gmail.com

Greg Bronevetsky
Google
bronevet@google.com

Saurabh Bagchi
Purdue University
sbagchi@purdue.edu

ABSTRACT

With the increase in the number cores in modern architectures, the need for co-locating multiple workloads has become crucial for improving the overall compute utilization. However, co-locating multiple workloads on the same server is often avoided to protect the performance of the latency sensitive (LS) workloads from the contentions created by other co-located workloads on the shared resources, such as cache and memory bandwidth.

In this paper, we present Pythia, a co-location manager that can precisely predict the combined contention on shared resources when multiple co-located workloads interfere with an LS workload. Pythia uses a simple linear regression model that can be trained using a small fraction of the large configuration space of all possible co-locations and can still make highly accurate predictions for the combined contentions. Based on those predictions, Pythia judiciously schedules incoming workloads so that cluster utilization is improved without violating the latency threshold of the LS workloads. We demonstrate that Pythia's scheduling can improve cluster utilization by 71% compared to a simple extension of a prior work when the user is ready to sacrifice up to 5% in the QoS metric and achieve cluster utilization of 99% if 10% degradation in QoS is acceptable.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**; *Main memory*; • **Computer systems organization** → *Processors and memory architectures*; Multicore architectures;

KEYWORDS

Resource Utilization, Scheduling, Interference, Performance

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274820

ACM Reference Format:
Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. 2018. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3274808.3274820

1 INTRODUCTION

Large datacenters can host up to several thousands of servers and form the backbone of the compute infrastructure for many large organizations. These large datacenters often cost billions of dollars of capital investment [14] for server procurement, site construction, and operating expenses. Thus, efficient use of all the available computing resources is of paramount importance for minimizing the total cost of ownership (TCO) of datacenters [35].

However, surprisingly, industry-wide datacenter utilization is estimated to be very low [1, 8, 38, 45, 54, 56]. Since, most processors today are equipped with several cores [2], thus one straight forward way to increase the utilization could be to run several applications simultaneously on different cores of the processor where each of those applications themselves can be multi-core. However typically, if one of the workloads is of user-facing or *latency sensitive (LS)* type, this approach is not taken in order to protect them from slowing down. Examples of these LS workloads abound, *e.g.*, web servers, database servers, search, etc. which can often constitute a significant fraction of all workloads that are consistently run in a datacenter. The main reason why such a possible slowdown of the LS workloads is anticipated is because of contentions in the shared resources created by other co-located workloads[1], degrading the quality of service (QoS) metric (e.g., latency).

The other class of workloads in datacenters is often called *batch workloads*. These include a broad swath of non-interactive and non-latency sensitive jobs that perform offline data consolidation, processing, analytics, model training, simulation, batch processing, search index creation, backup/restore and many other maintenance tasks. Even though these tasks are expected to complete within a certain broad time range, they are immune to modest variations in

[1]In the context of this paper, the term *co-location* implies two or more workloads concurrently running on different cores on the same server.

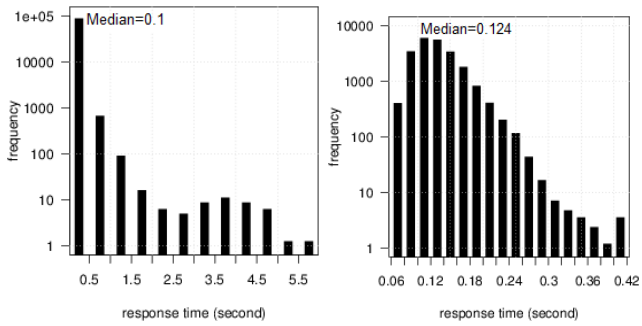
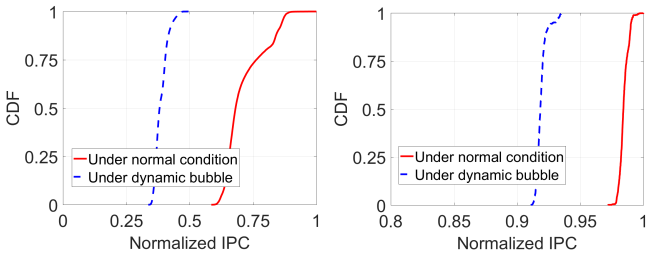


Figure 1: Distribution of response times of an LS workload running on (a) Amazon's EC2 (b) private cloud. The workload has a tail time of 55X (EC2) or 4X (private) of the median due to interference.



(a) MongoDB under low load. ***(b) MongoDB under high load.***

Figure 2: Dynamic bubble to detect the sensitivity characterization severely hurts the performance of the latency sensitive workload (MongoDB).

their completion time. For example, it is acceptable if a task that performs nightly computation of personalized recommendations, completes in less than 6 hours so that the recommendations can be served next morning. Data Center Infrastructure Management that provides a complete inventory of all assets in the datacenter, also mentions such loose deadlines in the order of a day or more [15].

Sources of interference:

Even though each server in a datacenter has several cores [42, 53]), an LS workload can suffer from performance interference if there are other workloads running concurrently on the same server. This happens due to subtle sharing and interference relationships through various shared resources, such as Last-Level-Cache (LLC), memory, and memory bandwidth. For example, LLC, which is shared across cores in the same chip, is a major source of interference as shown by numerous studies [20, 44, 49, 56]. Often workloads put higher pressure on the memory bandwidth either when they are under high load or when working with large datasets because the working-set size does not fit in on-chip caches. Running applications on two separate sockets does not solve the problem entirely as many workloads have memory access patterns that are random and that span across sockets in a NUMA architecture [33, 58]. *Thus, to improve the utilization of the entire cluster, utilization per socket must also be increased through higher degree of co-location.*

Interference or contention through shared resources also affects the application performance in VM-based cloud systems. As shown in Fig. 1, it can be observed that even when applications run in their own VMs in one of the most advanced cloud infrastructures, Amazon EC2, interference can cause the response time of an LS workload (a social network benchmark from CloudSuite [22]) to shoot up 55X from its median value. In a private cloud environment, the same LS workload suffered a 4X slowdown in the response time.

Static vs. dynamic scheduling: To improve the utilization in datacenters, prior research proposed many techniques to intelligently schedule workloads based on predictions about which combinations of workloads can be safely co-located [17, 38, 56, 60]. These previous works can be broadly categorized into static and dynamic approaches. In the static approach [38, 60], performance of the workloads are predicted using various models built offline and the scheduling decision is made based on the static profile. On the other hand, in the dynamic approach [17, 34, 56], the performance of the workloads is profiled online and then scheduling decision is made according to the up-to-date profile. Dynamic techniques can adapt to unpredictable workload variations, but they incur online overhead. For example, when we experimented with Bubble-Flux [56] we found its *dynamic bubble* mechanism that profiles the sensitivity characteristics of the LS workload online, causes severe performance degradation in the LS workload. Fig. 2 shows that QoS of MongoDB[2], a LS application, drops by 44% and 35% when the dynamic bubble runs, respectively under low and high load on a MongoDB server. According to the approach in Bubble-Flux, such perturbation will occur on every candidate server, whenever a batch workload enters the system and needs to be scheduled.

Hence, we posit that static techniques provide a safer and affordable choice for many scenarios while dynamic ones can be deployed to handle unforeseen situations (such as a new workload). Our solution improves the co-location decision at the time of scheduling using precise and lightweight online characterization of the applications.

Dynamic Monitoring and Control: In the dynamic monitoring and control schemes, after a batch workload is scheduled on a server to execute, its performance is actively monitored. If an LS workload is found to be affected by a co-located batch workload, the batch workload may need to be moved to a different server, which can be very costly due to factors like data locality. Hence, throttling of the batch workloads was proposed as a solution by prior works [40, 47, 56], in which the application is forcibly scheduled out for an algorithmically determined period. However, frequent throttling may cause a batch workload to miss even its loose deadline. An accurate workload placement or scheduling approach can not only reduce (or even eliminate) the frequency of such throttling, but also reduce the need for frequent monitoring leading to lower overhead for the runtime.

Profiling burden: The profiling burden (either offline or online) can be high since we are dealing with pairwise (or in our work *n*-way) co-location and hence pairwise interference characterization. Thus, if there are M batch applications and N LS applications, a full profiling will involve $O(M \times N)$ measurements. Therefore, to reduce the profiling burden, some prior works [17, 38, 56, 60] use one, or a few, generic applications as a stand-in for the LS application. We find empirically that this simplification is error-prone. The contention that a particular batch application causes on two LS applications can be quite different as we show in Fig. 4. This is orthogonal to the fact that different LS applications have different

[2]We report Quality of Service (QoS) in terms of the number of instructions per cycle (IPC) [28, 29, 48]. Thus, the lower the IPC, the lower the QoS.

sensitivities to a given amount of contention (which prior works do take into account).

Beyond pairwise co-location: One fundamental limitation of all prior work [17, 18, 38, 56, 57, 60] is that they cannot precisely predict the performance impact on the LS workload from *multiple* co-located batch workloads. Previous approaches [17, 18, 60] extended from pairwise co-location decisions (one LS workload and one batch workload) to multi-way co-location decisions (one LS workload and multiple batch workloads) by simply summing the contention caused by each batch workload or stay silent on this topic altogether. We define the term *"degree of co-location"* as the number of batch applications that are concurrently executing on one server with the LS workload. Thus, one LS and one batch application executing concurrently will denote a degree of 1. In Table 1 we show how much the observed contention of multiple co-located batch applications differs from the contention predicted by a simple additive model. The simple additive approach always overestimates the contention (by 16-76%) and the overestimation becomes worse with increasing degree of co-location (Figure 6). Thus, all prior works with such additive model will under-utilize a cluster by ignoring feasible multi-way co-location choices. Typically a datacenter runs a large *variety* of batch applications, which for a reasonably sized datacenter can be of the order of 1000s [17]. Therefore, it is not practical to develop a model for the contention caused by each possible combination (1 through n-way co-location) of these batch applications due to the combinatorial explosion.

Our solution approach: Pythia

In this paper, we present the design and implementation of Pythia[3], a co-location manager that provides precise estimates of combined contention created by multiple co-located batch workloads. It uses such prediction to find efficient co-location opportunities in the entire cluster, thus significantly improving the cluster utilization. For predicting contention from multiple co-located batch workloads, we empirically demonstrate that a weighted linear regression model works well for a wide class of batch workloads and performs far better than a simple additive model. We further show that to learn the parameters of this model, Pythia samples only a small fraction (less than 5%) of the large training space to avoid the high training burden. As Pythia can provide very accurate predictions for the contention induced by multiple co-located batch workloads (Section 5.7), it can be used for initial placement of workloads reducing the likelihood of contention (Section 5.4). Pythia incorporates dynamic monitoring and control schemes, which will be needed, hopefully infrequently, to deal with unexpected changes in application behavior. For specificity, we integrate with BubbleFlux's PiPo mechanism [56] to control the execution intensity of the batch workloads.

One shortcoming of Pythia relative to some prior works, such as Paragon, is that it is ideally suited for recurring workloads. Thus, Pythia has had occasion to perform offline modeling with these workloads, though the load profiles of the workloads can be different. If indeed an unseen workload is encountered, then Pythia has to perform the expensive profiling and modeling operations in the critical path before this workload can be scheduled. This may be an acceptable drawback in practice because data centers see the majority of their workloads as recurring [7, 23].

In summary, our paper makes the following contributions:

(1) We build a contention prediction model for the scenario of multiple co-located workloads. We show that for concurrently running workloads, our prediction model is better than that of prior solutions, which results in over-estimation of the contention and thus misses out on feasible co-locations.
(2) We characterize the contention caused by a batch workload more accurately by leveraging the insight that the contention is specific to the latency sensitive workload. Hence, the simplification for profiling purposes of measuring contention with an abstract, generic stand-in for the latency sensitive workload is inaccurate.
(3) We present the design and implementation of Pythia– a co-location manager that can precisely predict the contention induced by multiple co-located batch workloads running with a latency sensitive workload. Pythia imposes almost negligible online performance overhead and can train models very fast due to its simplicity. Our evaluations show that for a reasonable QoS policy, Pythia can increase server core utilization by 71% compared to an extension of prior work, on a 500 server cluster.
(4) Pythia is able to react to dynamic changes in the application's profile, say due to a change in load or application phase, and still support the co-location decision without degrading the LS workload's QoS below the user-specified threshold.

2 BACKGROUND

In this section, we will discuss at a high level the state-of-the-art approaches to characterize the contention between workloads and how to determine co-location of workloads. We categorize the existing approaches according to how they characterize the different workloads (LS workloads and batch workloads)—whether offline or online. What steps are executed when co-location decisions are made also varies depending on the above dimension.

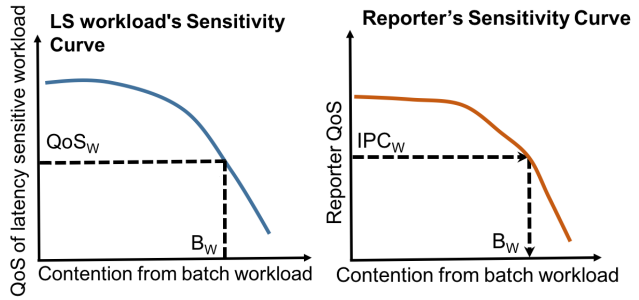
2.1 Offline Profiling for all Workloads

Pairwise sensitivity characterization measures the interference of each batch workload on each LS workload in terms of how much the QoS metric degrades in the LS workload [34]. This approach first measures the QoS metric of each LS workload when it is running alone. Then, it measures how much the QoS metric of each LS workload reduces after co-locating with each batch workload. This offline, pairwise profiling approach needs $(M \times N)$ measurements, where M is the count of batch workloads and N is the count of LS workloads. Finally, a lookup table is used for estimating the impact to an LS workload given any batch workload.

Microbenchmark based contention characterization quantifies the interference from any batch workload by using some microbenchmark (or microbenchmarks) as stand-in for the batch application. The effect of the microbenchmark on the LS workload is then studied. Examples of such microbenchmarks are the bubble application in Bubble-Up [38] and Bubble-Flux [56], Ruler application in SMiTe [60], and several microbenchmarks in Paragon [17].

To give specificity (and because we build on this technique, among others), we describe the bubble microbenchmark in Bubble-Up as a representative example. A bubble application can generate

[3] Pythia was a priestess of the Temple of Apollo at Delphi and was known for her accurate prophecies.



(a) Co-location of batch workload and LS workload *(b) Co-location of batch workload and Reporter*

Figure 3: Basic principle behind predicting QoS in pairwise co-location.

a series of discrete levels of memory contentions to profile LS workloads. We call the level of memory contention "contention score", with typical unit of MB, representing the memory and last level cache usage by a bubble application. By concurrently running the LS workload along with the bubble application, the QoS metric of the LS workload progressively degrades as the memory contention due to the bubble application increases as shown in Fig. 3(a). This curve is called the *"sensitivity curve"* and there is one for each LS workload. This step involves $B \times N$ measurements, where B is the number of discrete memory pressure levels in the microbenchmark.

In the next step, various batch workloads are run individually with each LS workload and the drop in performance of the LS workload is measured. Then, by referring to the sensitivity curve of the LS workload, the *"contention score"* for each batch workload is derived. This process is shown in Fig. 3(a) and this step involves $M \times N$ pairwise measurements. When making a scheduling decision about which batch workload w to co-locate with an LS workload i, the contention score of w toward i is looked up and if that contention score in i's sensitivity curve will *not* cause a degradation in the QoS below the threshold then w and i can be co-located.

Reducing burden of pairwise co-location profiling

To reduce the amount of pairwise contention measurements, which is $M \times N$, Bubble-Up calibrates a *reporter application* as the stand-in for *any* LS workload. Bubble-Up concurrently runs the reporter application along with various batch workloads (one at a time) and measures the drop in performance of the reporter application. By referring to Fig. 3(b), the corresponding contention score of the batch workload is calculated (follow direction of arrows). This approach reduces the burden of offline profiling from $M \times N$ to only M. The two other steps—calculation of the sensitivity curve for each LS workload and determining if co-location is feasible or not at the time of scheduling—stay the same. This approach significantly reduces the number of measurements to characterize contention from $M \times N + B \times N$ to $M + B \times N$. However, it is challenging to come up with a single reporter as a stand-in for any LS workload and we show empirically that this simplification is inaccurate (Sec. 3.1, Fig. 4).

2.2 Online Profiling for LS Workloads

Bubble-Flux [56] argues that due to phase, load, and input changes of an LS workload, the sensitivity curve of the workload may change significantly during runtime. So, it uses a dynamic bubble to perturb the LS workload running on each server and determines the sensitivity curve at runtime. Instead of the $B \times N$ measurements offline, it takes B measurements on each of the S available servers whenever it needs to schedule an incoming batch workload. The offline contention characterization of batch workloads and determining co-location decisions stay the same as in the earlier approach. The biggest challenge of this approach is that determining the sensitivity curve online, prior to scheduling, is heavyweight and is potentially disruptive to the executing applications.

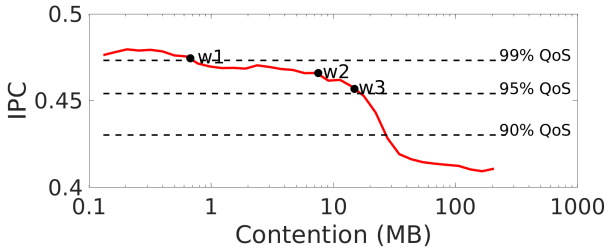
2.3 Online Profiling for all Workloads

Paragon [17] proposes a completely online solution to profile the interference of workloads and does not differentiate between LS and batch workloads. It considers two components—how sensitive a workload is to the interference from other workloads and the interference the workload itself generates. By co-locating a workload and a microbenchmark with steadily increasing contention scores, it determines how sensitive the workload is. Correspondingly, the interference that the workload generates is determined when the QoS of the microbenchmark is reduced below a certain threshold. However, instead of profiling each incoming application in detail, it leverages information the system already has about applications it has previously seen. It uses collaborative filtering techniques to quickly and accurately classify an unknown, incoming workload with respect to heterogeneity and interference in multiple shared resources, by identifying similarities to previously scheduled applications. The classification allows Paragon to greedily schedule applications, even unknown applications, in a manner that maximizes server utilization.

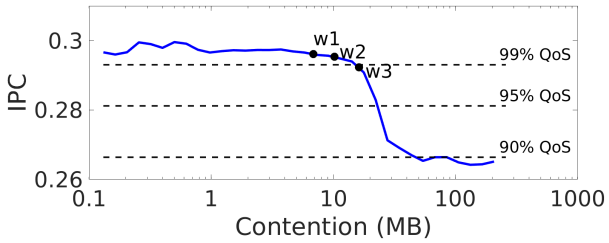
2.4 Interference due to Multiple Concurrent Batch Workloads

Processors of today have several tens of on-chip cores [2] and to improve compute utilization of a cluster all these cores must be kept occupied by applications. Since typically multi-threaded applications do not employ as many number of threads as there are cores in a processor, in a majority of cases multiple applications need to run on the different cores of the same processor. This motivates the need to run multiple co-located batch workloads with a single LS workload on a server.

In Bubble-Up, Bubble-Flux, SMiTe, and Quasar [18], the authors are silent about how they characterize the contention due to multiple batch workloads either at the time of scheduling or later during runtime. Elfen Scheduling [57] works only for 2-way simultaneous multithreading (SMT), where LS workload counts for one and the other batch workload counts for the second. One naïve baseline would be to use brute force and measure the contentions by all possible co-locations of the batch workloads. Suppose there are M batch workloads and maximum allowed degree of co-location is K. Then, this brute force approach would require running $\Theta(M^K)$ possible configurations. For reasonable values of $M = 20$ and $K = 5$ that are typical in a datacenter (M in datacenters can be much higher too



(a) Sensitivity curve of Latency Sensitive workload Redis and contention caused by three different batch workloads Blackscholes, Bodytrack, and Canneal from PARSEC benchmark (respectively, $w1$, $w2$, and $w3$).



(b) Sensitivity curve of Latency Sensitive workload MongoDB and contention caused by three different batch workloads to MongoDB.

Figure 4: Sensitivity characterization of two different LS workloads and contention caused by three batch workloads to each LS workload. The contention caused is different for the two LS workloads, thus disproving the prior approach of using a single stand-in benchmark for any LS workload for the purpose of contention characterization.

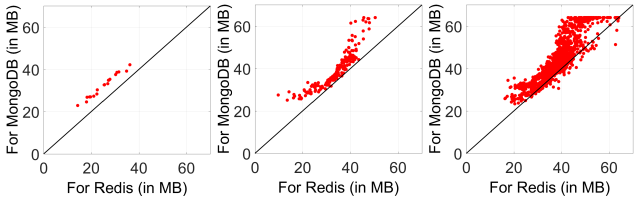
of the order of 1,000 [17]), this leads to an infeasible space of 3.2M configurations. Paragon [17] uses the *sum of interferences* of each individual existing workload on the server to represent the total interference on the incoming workload. Following this strategy, we create a baseline for comparative evaluation called Multi-Bubble-Up that estimates the contention score for each batch workload using the bubble mechanism explained above and then sums up the contention scores for determining the net contention due to multiple co-located workloads. We see empirically that this is highly inaccurate and we shed light on the reason for this in Sec. 3.2.

3 SOLUTION APPROACH

We first describe the two fundamental shortcomings of all prior works that hamper their ability to co-locate multiple applications. Then we present the solution mechanisms in Pythia and end the section with an end-to-end workflow of Pythia.

3.1 Contention Effect is LS-specific

Many prior works [56, 60] use a single generic application as a stand-in for any LS workload to characterize the contention *any* LS workload will face from a batch workload. This is done to reduce the burden of profiling—from $M \times N$ to just M. We examine if it is accurate to make this simplification. We first consider the interference model individually of two different LS workloads, Redis (a key-value store) and MongoDB (a NoSQL database server). In Fig. 4(a), we measure how much contention is caused to Redis due



(a) 1 co-located batch workload *(b) 2 co-located batch workloads* *(c) 3 co-located batch workloads*

Figure 5: Co-located batch workload induce different levels of contention when run with different latency sensitive workloads (e.g., Redis vs MongoDB).

to the three batch workloads, Blackscholes, Bodytrack and Canneal from the PARSEC benchmark suite [11]. As can be seen in the figure, the contention due to the three batch workloads correspond to 0.7 MB, 8.5MB, and 13MB and they cause 1%, 3%, and 5% QoS degradation respectively.

The sensitivity curve for the second LS workload, MongoDB, looks distinctly different (Fig. 4(b)). The contentions due to the three batch workloads are also different, 7MB, 10MB, and 10.5MB, respectively. From this evidence, it appears that one does have to consider each LS workload to understand the effect of contention due to batch workloads, rather than use a single stand-in microbenchmark.

In Fig. 5, we further investigate the difference of memory contention on Redis and MongoDB from 1, 2, and 3 co-located batch workloads. On the X-axis we plot the combined contention created by various co-located batch workloads, drawn from a set of 19 different applications from PARSEC and SPEC benchmark suites, running alongside the LS workload Redis. On the Y-axis we plot the same for MongoDB. If the same set of batch workloads always induces the same contention onto Redis and MongoDB, *i.e.*, the contention is independent of the LS workload, then all the points should fall on the black diagonal line. However, for these two LS workloads, the points are mostly scattered toward the upper half, indicating that batch applications create higher contention for MongoDB than for Redis. *Thus, accurate prediction of combined contention for multiple co-located batch workloads must take into account the specific LS workload that is being considered.* This observation however increases the complexity of the characterization that Pythia has to perform (Section 3.3).

3.2 Extend Existing Models to Predict Multi-way Co-location?

As we have discussed in the previous section, it is desirable to increase the degree of co-location due to increasing core counts on servers. However, prior works are either silent on how they estimate the contention score for multiple batch workload combinations during scheduling [38, 56, 60] or simply choose an additive model [17] (see detailed discussion in Section 2.4). To quantitatively see how the extension may work, we first attempt to directly sum up the contention of each batch workload when singly running with the LS workload and call it Multi-Bubble-Up, which is our extension of Bubble-Up using the additive model used by Paragon. Multi-Bubble-Up profiles the contention score of each batch workload

Table 1: Examples of inaccurate estimates of contention score for 8 combinations of batch workloads. Prefixes Parsec and Spec indicate the corresponding benchmark suit.

Batch workloads	Observed Contention (KB)	Estimated Contention (KB)	Error
SpecBwaves, SpecBzip	11694	14268	22%
ParsecStreamcluster, SpecBwaves, SpecZeusmp	12270	21630	76%
SpecTonto, SpecGcc	9704	14060	44%
ParsecFerret, SpecOmnetpp	5404	6596	22%
SpecBwaves, SpecH264, SpecOmnetpp	12533	17385	38%
SpecGromacs, SpecHmmer, SpecXalancbmk	7044	8226	16%
SpecMiLS, SpecBzip, SpecBzip	9127	11540	26%
ParsecBodytrack, ParsecCanneal, SpecMcf	8403	12895	53%

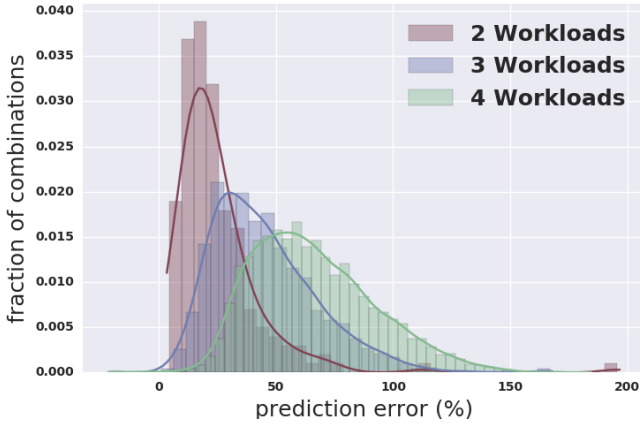


Figure 6: The distribution of prediction error from a direct application of prior work [38], for all combinations of 2, 3, and 4 batch workloads. We observe a mean prediction error of 26% for the 2 batch workload combinations, 45% for the 3 batch workload combinations, and 66% for the 4 batch workload combinations.

with each LS workload offline using the bubble microbenchmark explained in Sec. 2.1. Multi-Bubble-Up then estimates the contention score of multiple co-located batch workloads by summing up the contention score of each individual application in the online phase. To create the ground truth for the actual contention, we regard the co-located batch workloads as a group and actually measure the contention score of this group (using the same bubble approach).

Our empirical finding is that this is rather inaccurate for predicting the contention created by multiple co-located batch workloads. For some randomly sampled combinations of multiple co-located batch workloads, Table 1 shows the actual observed contention versus the estimations predicted by Multi-Bubble-Up. It can be seen that the predictions for the contention by Multi-Bubble-Up are overestimates by 16% to 76%. This leads to unnecessarily conservative choices about which batch workloads to co-locate with the LS workload, resulting in low utilization in the datacenter. Moreover, as we see in Fig. 6, the prediction error for the Multi-Bubble-Up model increases as we increase the number of co-scheduled batch workloads, a concern considering the desire for increased degree of co-location.

Source of Error in the Naïve Model: We investigate the source of the estimation error in the prior work and identify that the *mutual interference between the co-located batch workloads* is the root cause. At a high level, mutual interference between batch workloads throttles the batch workloads themselves, reducing their IPC and consequently their CPU utilization. This effect further reduces the *combined contention* on shared resources induced by the co-located batch workloads. As the number of concurrently running batch workloads increases, we expect a greater level of mutual interference among the batch workloads. Consequently, in Fig. 6, we see that as the mutual interference increases we see an increasing over-estimation by the naïve sum method.

3.3 Our Model for Multi-way Co-location

To improve the prediction accuracy of combined contention from multiple co-located workloads, Pythia designs a linear regression model. It first weights the individual contention scores of the batch workloads to account for the effect of mutual contention and then does a linear sum. Mathematically, our model has the form:

$$B_S = \sum_{W_i \in S} c_{W_i} B_{W_i}$$

where, S is the set of co-located batch workloads, c_{W_i} is the weight for the batch workload W_i and B_{W_i} is the contention induced by that batch workload when singly co-located with the LS workload. Apparently, Multi-Bubble-Up assumes all contention coeffients c_{W_i} to be 1. This model looks deceptively simple, and it is relative to non-linear models, but it is accurate in predicting combined contention to the extent that is required for making co-location decisions (as we demonstrate empirically in Section 5). The reader should also note that this model is combined with two other dimensions—the specific LS workload (*i.e.*, the contention score of a batch workload will be different for different LS workloads) and the weights depend on the degree of co-location. The second factor means that when the batch workload W_i is co-located with one LS workload and one other batch workload (co-location degree 2), then its weight can be represented as $c^{(2)}_{W_i}$, while for co-location degree 3, it can be different, *i.e.*, $c^{(2)}_{W_i} \neq c^{(3)}_{W_i}$.

Interpretation of the weights: The weights c_{W_i} in our model represent the tendency for the associated batch workload W_i to suffer interference from *other batch workloads* and therefore, exert reduced cache and memory pressure as a result of co-location. Larger weights indicate a *resistance to interference*, while smaller weights represent a *vulnerability to interference* and hence a tendency for that batch workload to exert less memory contention to the LS workload when competing for resources with other batch workloads on the same server.

Calculating the weights offline: The weights (c_{W_i}'s) are derived using linear regression on a training dataset obtained from a small number of training runs, where each run has a combination of batch workloads. The regression technique attempts to calculate the weights so as to minimize the prediction error of the combined contention. One possible concern is the training cost for the model because the search space is large and grows as $O(M^K)$ where M is the count of available batch applications and we are considering co-locations of up to degree K. In Sec. 5, we experimentally demonstrate that training runs consisting of less than 5% of the combination space are enough to predict the combined contention with high accuracy for *any* possible co-location combinations.

To train our model offline, the inputs are the single workload contention scores B_{W_i} for batch workloads W_i, where $i = 1, 2, \ldots, M$,

and the observed contention B_i from multiple co-located batch workloads, where $i = 1, 2, \ldots, k$ and k is the number of sparsely sampled data points. The output of this modeling are the weights c_{W_i}'s. We have to solve the equations separately for each degree of co-location $d = 1, 2, \ldots, K$. Our linear regression model can be set up formally as follows (for clarity of notation, we omit the superscript d that should go with $\xi_{i,j}, c_{W_i}$, and B_i):

$$\begin{pmatrix} \xi_{1,1}B_{W_1} & \xi_{1,2}B_{W_2} & \cdots & \xi_{1,M}B_{W_M} \\ \xi_{2,1}B_{W_1} & \xi_{2,2}B_{W_2} & \cdots & \xi_{2,M}B_{W_M} \\ \vdots & \vdots & \ddots & \vdots \\ \xi_{k,1}B_{W_1} & \xi_{k,2}B_{W_2} & \cdots & \xi_{k,M}B_{W_M} \end{pmatrix} \begin{pmatrix} c_{W_1} \\ c_{W_2} \\ \vdots \\ c_{W_M} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{pmatrix}$$

where $\xi_{i,j}$ represents the number of times that batch workload W_j appears in the i^{th} workload combination[4], such that $\sum_{j=1}^{M} \xi_{i,j} = d$. In practice, most values of $\xi_{i,j}$ are zero because the number of batch workloads M is much larger than the degree of co-location.

For nearly all sample sizes k (rows) and number of batch applications M (columns), this system of linear equations is over-determined. For example, in the three-way co-location scenario and with a sample rate of s, the sample size (*i.e.*, number of rows) is $k = sM^3$. The system is over-determined if

$$sM^3 > M \Rightarrow s > \frac{1}{M^2}$$

As a result, we typically cannot find an exact solution, and instead perform least squares optimization to find the set of coefficients c_{W_i} that minimizes the prediction error on the combined contention scores B_i's.

As we have discussed here, the weight corresponding to a batch workload is calculated as distinct for each unique LS workload. On the other hand, the contention due to a batch workload W_i does *not* differ significantly as you change the exact set of *other co-located batch workloads*, for a given degree of co-location. Thus, the weights c_{W_i} of batch workload W_i does not consider its set of co-located batch workloads. This helps us avoid a combinatorial explosion in modeling the contention created by batch workloads. We experimentally demonstrate this phenomenon in Sec. 5.6 (Fig. 15).

3.4 QoS Metrics

Pythia uses instructions per cycle (IPC) as the QoS metric of the LS workload. The rationale behind using IPC instead of the intuitive latency metric is two folds. First, IPC measurements are more immune to system noise and hence ideal for building predictive models. Second, IPC being measured at the server side is not affected by fluctuations in the network conditions. Our use of IPC is in line with prior works [13, 21, 28, 39, 56, 60], of which [56, 60] use it specifically as a substitute for latency of LS workloads.

We also empirically validate the negative correlation between IPC and read latency in multiple LS workloads. A representative example is shown with MongoDB in Fig. 7. For the allowable QoS degradation of an LS workload under co-location, we use a QoS policy which gives what percentage of IPC compared to the no contention case, the system administrator is willing to tolerate. The higher this percentage is, the higher is the requirement for

[4] Multiple instances of the same batch workload are co-located on different cores.

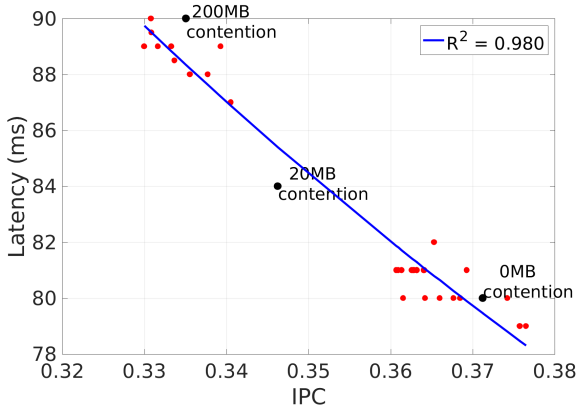


Figure 7: Read latency in MongoDB under different IPCs, showing high negative correlation between the two metrics.

the performance of the LS workloads. We use the commonly used thresholds of 80%, 90%, 95%, and 99% in our evaluation settings.

Temporal variations in the resource usage of the batch workloads may cause QoS violation in the LS workloads and a mean or median value would ignore such tail behaviors. So, in Pythia, we capture the impact of the batch workload on the tail latency of the LS workload by using the 95^{th} percentile of the latency (measured through 5^{th} percentile of the IPC) as the effective contention score. By using the 95^{th} percentile of the latency, Pythia is more conservative about preserving the performance of the LS workload.

3.5 Dynamic Adaptation to LS and Batch Workloads

In our design presentation so far, we have chosen offline profiling of both LS and batch workloads, to avoid degrading the performance of the applications due to online profiling. However, this basic mechanism cannot deal with scenarios where the contention characteristic changes *drastically* at runtime due to phase change behavior in the application or changes in the input to the application. To handle such scenarios, Pythia takes the approach of creating separate profiles for each LS workload corresponding to the different load conditions and different input classes. For example, for MongoDB, we have 3 different load classes, high, medium, and low, and 3 different input classes, corresponding to read-heavy, read-write-balanced, and write-heavy, thus giving a total of 9 profiles. This leads to a constant multiplicative factor increase in the number of LS workloads to consider in our modeling, some domain knowledge about what features to consider for creating the input classes, and a selection of the number of classes to use. In practice however, this task is relatively straightforward. The domain knowledge is at a high level and available to anyone who is going to run the LS workload and cares about its performance. The number of classes can be determined through a sensitivity study and we find empirically that 3 is sufficient for both load and input classes—the sensitivity curves tend to have significant overlap with more than 9 profiles for a given LS workload. At runtime, we inject a *single* bubble and look up from the different sensitivity curves, which

profile of the LS workload to use. Our approach creates less perturbation to the running application than the alternative approach of Bubble-Flux, which creates the entire sensitivity curve of the LS workload at runtime by injecting bubbles of different sizes.

To deal with dynamism of batch workloads, we find that most are long-lasting and their usage profiles can vary depending on the phase of the application or the size of the input to the application. For example, in Map-Reduce tasks (many batch applications are structured as such), heavy computation of the Map phase is followed by heavy disk IO of the shuffle operation prior to the Reduce phase. We therefore adopt a simple memory pressure monitoring scheme, which can be implemented in a lightweight manner using hardware performance counters. If the monitoring provides hint that the batch workload's usage profile has changed (though there may be false alarms in this), then we re-calibrate the contention score corresponding to the batch workload's current state. If the contention score is determined to cause unacceptable level of contention to the LS workload, then the batch workload is suspended for some length of time. The ratio of suspend time to execution time is higher the larger the contention score is. This mechanism of suspend and resume is identical to the Phase-in-Phase-out (PiPo) mechanism of Bubble-Flux. The performance impact of our mechanism is reduced by the lightweight monitoring that provides hint as to when to apply the more heavyweight mechanism. We show a focused experiment to isolate the effect of our dynamic adaptation mechanism in Section 5.4.

3.6 Scheduling

When a new batch workload arrives and needs to be scheduled by Pythia among all available servers, it uses the *Best Fit* algorithm [9]. According to Best Fit, Pythia places the workload on the server, which will have the smallest amount of headroom left after placing the new workload, while respecting the QoS threshold of the LS workload. To illustrate this, let us assume there are two servers $H1$ and $H2$. $H1$ is already running Redis (the LS workload) along with batch workloads A and B. $H2$ is already running MongoDB (the LS workload) along with batch workload C. Now Pythia has to decide where to place an incoming batch workload D? Let us assume our target is to maintain at least 95% QoS for both Redis and MongoDB which translates to $50MB$ of allowed contention in case of Redis and $30MB$ for MongoDB. Let us say, Pythia uses its models to predict that the combined contention, if A, B, and D are run together, is $47MB$, whereas if C and D are run together the combined contention would be $18MB$. Since this satisfies the QoS criteria for both Redis and MongoDB, both $H1$ and $H2$ are valid placement points. Now according to the Best Fit algorithm, Pythia chooses $H1$ to place the incoming workload D because that would result in the tightest packing of applications in the servers as only $2MB$ of the allowed contention will be left in $H1$ as opposed to $12MB$ of possible slack in case of $H2$. In case, there are no LS workloads present in the server, then the need for protecting the QoS of the LS workloads does not arise. However, we still need to decide where an incoming application should be placed. Pythia could determine the optimum limit for available shared resources for a server by progressively increasing the bubble size of the bubble application and monitoring the total number of swap-ins and swap-outs until the number crosses an acceptable threshold (T). The insight is, if the combined contention score of the batch applications becomes close to this threshold contention score, one or more batch applications might start thrashing — severely degrading its performance and even risk the violation of their deadlines. After that, Pythia uses the same *Best Fit* algorithm to decide the placement of the workload as long as the combined contention does not cross $T - \epsilon$, where ϵ is a small buffer that prevents Pythia from going too close to the thrashing state. If no such co-location is possible, Pythia schedules the incoming workload to an empty server. The scheduling approach avoids unnecessarily waking up idle servers and improves overall utilization. Pythia can also trivially accommodate any other placement algorithm than Best Fit. For example, if the number of servers in the cluster is too large, First Fit algorithm, which would place the job in the first server that can host the workload without violating the QoS, can be used to reduce the placement search time.

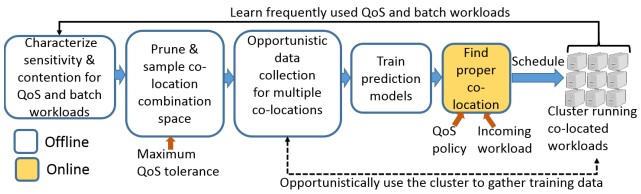


***Figure 8: The complete workflow of* Pythia.**

3.7 Workflow of Pythia

The workflow of Pythia is illustrated in Fig. 8 and comprises of a few major steps, which we describe below. While a majority of Pythia's steps are offline, the final step corresponding to co-location prediction and scheduling is online (marked in yellow). As we have mentioned earlier, Pythia is most effective with recurring workloads, *i.e.* workloads that have been seen before, though their loads may now be different. If the incoming workload that needs to be scheduled is indeed an unknown one, then Pythia must go through the expensive process of profiling it (all the four offline steps 1-4 below) before the new workload can be scheduled.

Step 1: Single Workload and Sensitivity Characterization: The first step performs contention characterization for each batch workload when singly running with a particular LS workload. Such characterization is performed offline for each workload as originally proposed in [38].

Step 2: Pruning and Sampling: The second step reduces the combination space by removing the batch workloads that are far too contentious to have a chance of forming a safe co-location with other batch workloads. To reduce the search space for training, for each LS workload Q, using its sensitivity curve, we determine the maximum contention score θ_Q that Q can tolerate given a QoS policy, and then we discard the batch workloads that create contention higher than θ_Q even when singly co-located with Q.

Step 3: Data Collection to train the model of multiple co-located workloads: From the pruned search space Pythia uses a very small fraction (5% or less in practice) of the combinations of multiple co-located batch workloads to build the prediction model. For these executions, Pythia only collects the IPC values of the LS

workload and measures the combined contention induced by the co-located batch workloads.

Step 4: Model Construction: Here, we build our linear regression prediction model for the contention due to multiple batch workloads. We use the single workload contention scores B_{W_i} for a batch workload W_i from step 1, and the sparsely sampled combined contention of multiple co-located batch workloads from step 3. The output of this modeling step are the weights c_{W_i}'s and these weights are specific to each n in an n-way co-location scenario and to each LS workload.

Step 5: Scheduling: For an incoming workload, Pythia uses its predictive model to select the set of servers on which the workload can be safely co-located without violating the QoS of the LS workload already running on those servers. Then, Pythia uses the *Best Fit* algorithm [9] to place the workload in the server with the lowest amount of available shared resources left.

4 IMPLEMENTATION DETAILS

The various components of Pythia are implemented using a combination of Python and shell scripts and consisting of roughly 5.5 KLOC. For all experiments, the LS and the batch workloads run on different cores, achieved using the Linux command `taskset`. In a NUMA machine, if a process running on a core of one socket accesses memory on the DRAM associated with some other socket, there is a performance impact. So Pythia pins the memory allocation on the same NUMA socket as the workload using the `numactl` command. Pythia measures the IPC of the LS workload at the server side using the Linux tool `perf`. The IPC values are sampled every second and recorded as time series. We found that sampling IPC every second did not have any observable performance impact on the LS workload but sub 100ms sampling rate did have a noticeable performance impact.

Cycle stealing to collect training data

To generate all the training data on our production cluster's resources, we employ a cycle stealing approach whereby we opportunistically execute training runs whenever a server is not running production workload. There is a Master server and multiple Worker servers. The Master server is responsible for generating a list of experiments in which the LS workload and the co-located workloads (either a bubble application, a batch workload, a combination of batch workloads or nothing) are specified. Each Worker requests experiments from the Master, executes the experiments on the nodes managed by itself, returns the completed experiments to the master, and requests for more experiments. The selected combination of batch workloads by Pythia is randomly chosen in the search space of all n-way batch workload combinations. In our environment, we run the experiments in a Moab [4] managed cluster.

5 EVALUATION

5.1 Experimental Setup

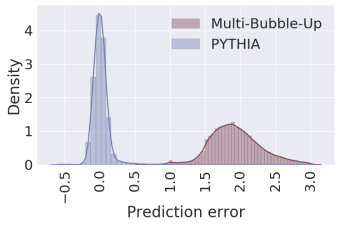
Latency Sensitive Workloads: We use two popular LS applications: Redis [6] and MongoDB [5] for our evaluations and drive these using the Yahoo Cloud Serving Benchmark (YCSB) [16]. Redis [6] is an open source in-memory Key-Value database. It is also often referred as a *data structure server* as it provides access to mutable data structures through a set of commands. MongoDB [5] is an open source, document-oriented NoSQL database. Instead of using traditional table-based relational-database structure, MongoDB uses documents with dynamic schemas similar to JSON objects [3]. MongoDB uses embedded data models to reduce I/O and can support very fast queries.

Batch Workloads: We use a large variety of representative batch workloads drawn from the widely used PARSEC [11] and SPEC2006 [25] benchmark suites. However, after initial profiling for determining the representative contention score, we found out that among the 12 PARSEC, 29 SPEC workloads, only 19 workloads were chosen by Pythia as prospective *co-locatable* batch workload candidates. Rest of the batch workloads had much bigger individual contention scores resulting in degradation of the QoS of the LS applications beyond acceptable limits (80% QoS policy in our case). Of these surviving batch applications, 4 are from the PARSEC suite, while the other 15 workloads are from SPEC. We also use two memory-intensive workloads from the 8 CloudSuite [22] benchmarks – an in-memory analytics workload and a graph analytics workload and separately report specific results of these memory-intensive workloads.

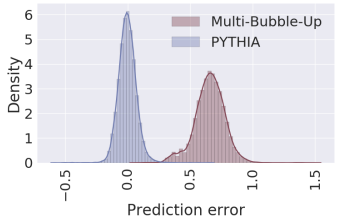
Cluster Setup and Co-location Settings All the experiments were run on a 1,296-node homogeneous production cluster. Each node had two sockets, each with one 2.60 GHz, 64-bit Intel Xeon 8-core E5-2670 processor (total 16 physical cores per node). Each node had 32 GB memory divided between the two sockets. Each processor had a 20 MB L3 cache (LLC) shared between the 8 cores on each socket. Since all workloads are multi-threaded, we used two cores per workload to preserve the general multi-threaded behavior of the workloads. Since, each socket had only 8 cores, the maximum degree of co-location that we use per socket was three co-located batch workloads along with the LS workload (3×2 for batch workloads + 2 cores for the LS workload). We used the second socket of the processor to run lightweight scripts for controlling the experiments and collecting data for model building, thus ensuring negligible perturbation to the system being studied.

5.2 Prediction Accuracy of Pythia

Contention Prediction: We now evaluate if Pythia can precisely predict the combined contention from multiple co-located batch workloads on an LS workload, and compare the performance to Multi-Bubble-Up. We first train our models by randomly sampling a small subset (5%) of the entire space of all combinations of one, two, and three co-located batch workloads chosen from the finally selected 19 batch workloads. Then we predict the combined contention from another combination of co-located workloads (the test set) that was *completely disjoint* from the training set. We plot relative error in the predicted contention error distribution as kernel density estimation (KDE) [50], for both Redis and MongoDB in Fig. 9. It can be observed that Pythia makes highly accurate predictions as the error distributions for both Redis and MongoDB are centered around zero and the shapes (Y-axis is the *density* of the prediction error) are much narrower compared to the error distributions obtained using Multi-Bubble-Up. The median error for Pythia with 5% sampling rate is 6.3% for Redis and -3.4% for MongoDB while that for Multi-Bubble-Up is 185% and 61.5%.

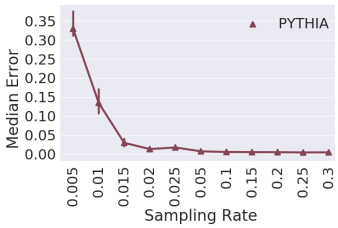


(a) Redis

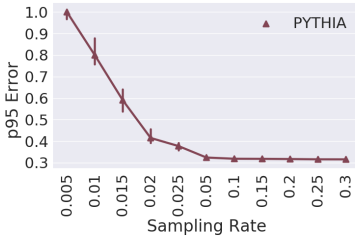


(b) MongoDB

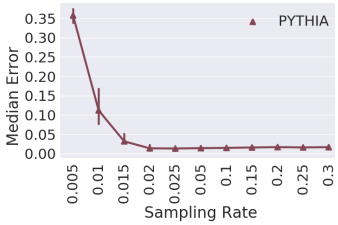
Figure 9: Contention prediction error distribution for Redis and MongoDB with 5% sampling rate



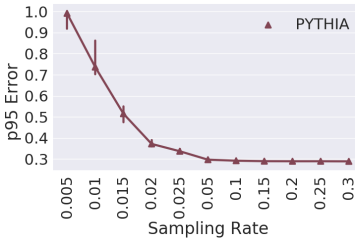
(a) Median error with Redis



(b) 95th percentile error with Redis

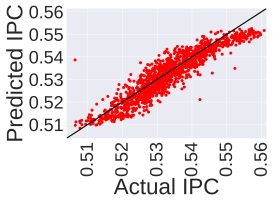


(c) Median error with MongoDB

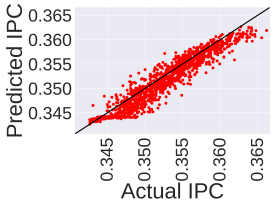


(d) 95th percentile error with MongoDB

Figure 10: Learning curves for predicting contention with Redis and MongoDB



(a) Redis



(b) MongoDB

Figure 11: Prediction accuracy for IPC of Redis and MongoDB with degree of co-location 3

The standard deviation of the error distribution for Pythia is 15.2% for Redis and 10.6% for MongoDB compared to 28.6% and 17.1% for Multi-Bubble-Up. Thus, Pythia's prediction is both more accurate and less variable relative to the prediction from Multi-Bubble-Up. Fig. 9 also supports our earlier hypothesis about the source of inaccuracies in Multi-Bubble-Up. Since for Multi-Bubble-Up most of the prediction errors are positive, this indicates that it systematically overestimates the contention.

In Fig. 11 we show the accuracy of Pythia in predicting the actual IPC of the LS workloads when co-located with three batch workloads corresponding to 1,330 different combinations. We see that the IPCs predicted by Pythia are quite accurate for both MongoDB and Redis as the bulk of the points fall close to the diagonal line with very few outliers. Also, there is no systematic over-prediction or under-prediction for either LS workload.

5.3 Learning Curves

We now evaluate how the prediction accuracy varies as we control the sampling rate to train Pythia's models. Fig. 10 shows how the median and the 95th percentile prediction error for the combined contention drops as we increase the sampling rate for co-location with Redis and MongoDB. This characteristic would help in choosing the optimum sampling rate where high accuracy is desirable but minimizing the cost of training is also important since we use a production cluster for collecting the training data. We perform 50 experiments at each sampling rate and plot the average. For both Redis and MongoDB, it can be observed that even a sampling rate of only 2.5% is good enough to achieve a high median accuracy of prediction. However, if we are interested in the 95^{th} percentile error, we need a higher sampling rate of 5%. Going higher than that does not significantly improve the metric. On the lower extreme, as the sampling rate goes to 1% and below, we see that prediction accuracy dramatically degrades as the model simply does not have sufficient information to make accurate predictions. For sampling rate of 2% and higher, the variation in error is insignificant for Pythia's predictive models.

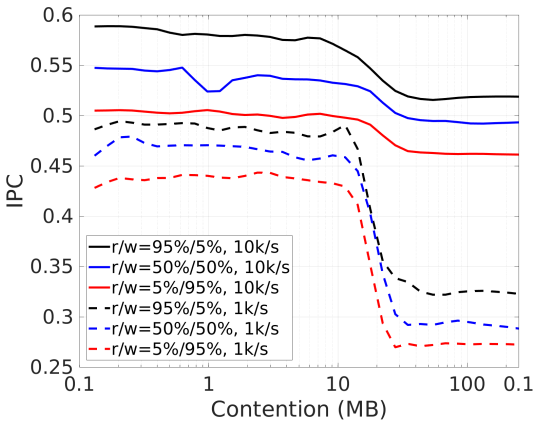


Figure 12: Sensitivity curves of Redis under different read/write ratios and loads.

5.4 Synergy of Pythia with Dynamic Scheduling, Monitoring, and Control

Here we see the quantitative rationale for Pythia's design to handle runtime usage changes in LS workloads. Fig. 12 shows the sensitivity curves of Redis under different read/write ratios (3 levels) and different loads measured in 10^3 operations/s (2 levels). As introduced in Sec. 2.1, each of these sensitivity curves is determined offline by co-locating Redis with a bubble microbenchmark with increasing contention score. It can be seen that the IPC of Redis is different in the different usage cases. At runtime, Pythia uses

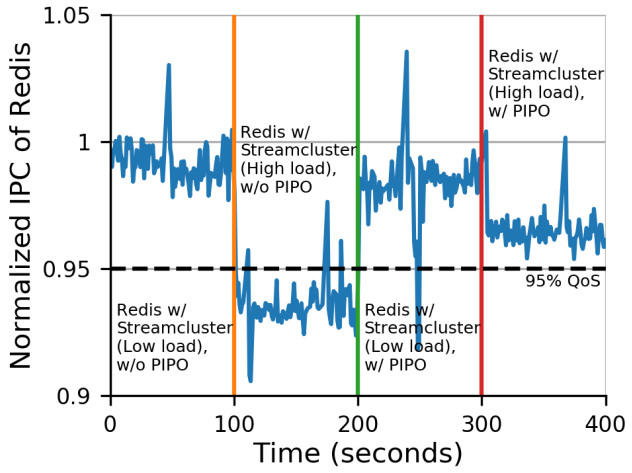
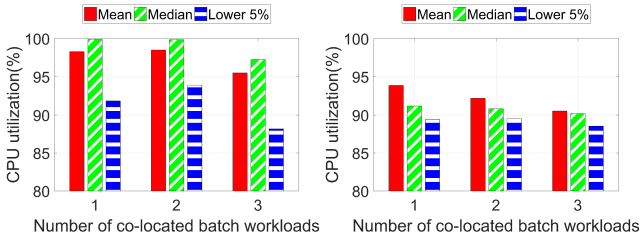


Figure 13: The runtime QoS of Redis when co-locating with a batch workload whose load changes.* Pythia*'s dynamic mechanism allows Redis to meet its QoS.



(a) Redis ***(b) MongoDB***

Figure 14: Increase in the degree of co-location marginally affects the performance of the batch workload.

a bubble with a particular known contention score and measures the corresponding IPC of the LS workload. Then, from the predetermined set of sensitivity curves, it determines the appropriate sensitivity curve to use, which is an accurate reflection of the current usage profile of the LS workload. In this approach, Pythia reduces the online overhead from B measurements of Bubble-Flux to a single measurement, at the cost of the offline profiling and the quantization of the usage profiles.

Here we evaluate the role a dynamic mechanism (Bubble-Flux's PiPo for specificity) that is coupled with Pythia, plays in shielding the LS workload from uncertainties in the intensity of a co-located batch workload. In Fig. 13, we co-locate a batch workload from the PARSEC benchmark, Streamcluster, with Redis, the LS workload. We observe that through the load changes in Streamcluster, during low load, the QoS of Redis stays well above the user-specified QoS threshold (which for this experiment is defined as 95%). However, when the Streamcluster load increases (the second phase between 100 and 200 s), the QoS threshold is violated most of the time. However, Pythia, using the PiPo mechanism can control the proportion of execution allowed to the batch workload. Thus it can control the contention that the batch workload can cause, so that even when the batch workload is running at high load, we can still preserve the QoS of the LS workload. Thus, we see that in the 4th phase, the QoS of Redis stays above the threshold line.

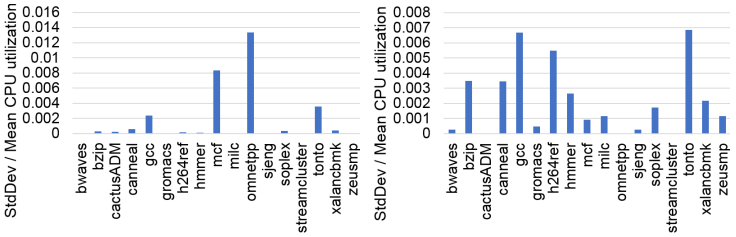
5.5 Performance Analysis of Pythia

Pythia's modeling phase is composed of the training data collection and the model construction, both of which happen offline. For the first part, Pythia's training data collection is opportunistic as it tries to utilize idle nodes. Hence it is not possible to quantify the total amount of time to gather training data for Pythia's models since there is not a continuous period of time. We found that each experiment (corresponding to an execution of an LS workload and a certain number of co-located batch workloads) ran for 195 sec on average. Moreover, since a sampling rate of 5% gives accurate enough results, our modeling and data collection time are significantly shorter. For the second offline part, model construction is very efficient and takes less 40 millisecond for various sampling rates and co-location combinations (21 ms for 2.5% sampling rate and 35 ms for 30%). This is a consequence of using a simple linear model with constant coefficients, which despite its simplicity provides accurate predictions. We also estimate that this modeling step would take approximately 1.48 sec and 12.07 sec if 100 and 200 different batch workloads are used (instead of the 19 that we used), respectively. Thus Pythia's modeling step uses just 9.1μ s per sample and can handle unto 2.17M co-location combinations/s at a sampling rate of 5%. For the online overhead, recall that the actual prediction for the combined contention from multiple workloads is just a dot-product between the individual contentions of the batch workloads and the corresponding coefficients from the trained model. Hence the prediction in Pythia is extremely fast and each prediction took *less than 1 millisecond* in our experiments.

5.6 Impact of Co-location on Batch Workloads

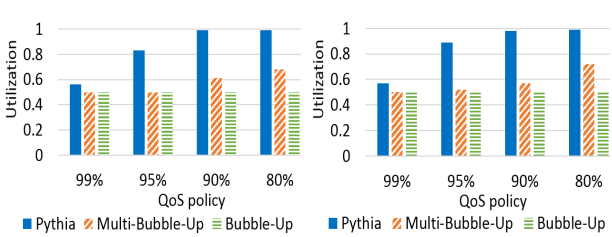
An obvious question at this point is, as a result of these co-locations, how much do these batch workloads suffer. If they suffer significantly so as to miss even their loose time deadlines, then the question of co-locating multiple batch applications will become moot and we can then rely on prior work. We see from Fig. 14 that for 3 co-located batch workloads along with the LS workload, the batch workloads suffer only less than 10% degradation in their individual performance without affecting the QoS of the LS workload. Since batch workloads are inherently tolerant to small performance degradation, multiple co-location is thus feasible.

Recall, for each LS workload, our model assigns a single contention coefficient capturing the impact due to mutual interference on each of the batch workloads. The question that arises is why such a single coefficient per batch and LS workload combination is good enough, rather than a more complex model where the coefficient depends also on the exact *set* of co-located batch workloads. Proof of this can be seen in Fig. 14, which shows the mean, median, and lower 5% CPU utilization of various co-location combinations of the 19 batch workloads and for three different degrees of co-location for both Redis and MongoDB. Here we measure the effect of mutual interference through the variations in the CPU utilization because when a batch workload suffers from mutual interference, its CPU utilization also drops. These batch workloads are primarily CPU-bound and so we measured the CPU utilization of cores used



(a) Redis *(b) MongoDB*

Figure 15: Coefficient of variation in the performance of a batch workload due to mutual interference from other co-located batch workloads. For 17 of the 19 batch applications, the variation is very small. Bodytrack and Ferret are outliers (not shown here).



(a) Utilization with MongoDB *(b) Utilization with Redis*

Figure 16: Cluster utilization for various QoS policies.

by the batch workloads. It can be observed that as we increase the degree of co-location, the CPU utilization of the batch workloads decreases but only by very small amounts. For three co-located batch workloads, the median CPU utilization drops by only 1.4% for Redis and 3% for MongoDB compared to a single co-located batch workload. Further proof for this comes from Fig. 15, where we plot the *coefficient of variation (CV)* of the CPU utilization (which is standard deviation over mean) for a batch workload B_i when it was co-located with any other batch workload and the LS workload. The magnitude of this variation (on the Y-axis) is extremely small for 17 out of the 19 batch workloads—maximum of 0.7% for MongoDB and 1.3% for Redis. This observation highlights that for the majority of the cases (17 out of 19), a single coefficient per batch workload is adequate and there is no need to have different coefficients for each specific batch workload combination, which would have significantly complicated the training process.

In contrast to the general trend, the CV measures for the two outliers Bodytrack and Ferret are much higher—11% for Bodytrack and 3% for Ferret w.r.t. Redis and 44% for Bodytrack and 40% for Ferret w.r.t. MongoDB. This indicates that their offered contention depends exactly on which other batch workloads are co-located with them. Digging deeper, we find that the CPU utilization of Bodytrack (also Ferret) differs depending on how many other batch applications it is co-located with. We found that Bodytrack, which is a computer vision application for tracking human body through a sequence of images, is very sensitive to available memory bandwidth [10]. Thus under memory contention from the co-located workloads its CPU utilization drops with increase in the degree of co-location because of the reduced available memory bandwidth. Ferret is a content-based image similarity search application. It keeps a database of feature vectors of images in memory to find the images most similar to a given query image and its working-set size is unbounded [12]. Thus, with increase in the degree of co-location, the CPU utilization of Ferret drops due to both cache and memory-bandwidth interference. For such volatile workloads, we can create a more fine-grained model that has coefficients corresponding to the exact combination of *other* batch workloads co-located with the volatile workload.

Co-location with Memory-Intensive Applications:

Since some batch applications are memory-intensive we experiment with two such applications—a graph analytics application and an in-memory analytics application from CloudSuite [22]. We show the result for instances of these batch applications running with Redis and MongoDB in Table 2. We see that Pythia can safely co-locate up to 2 instances of the graph analytics application with Redis and up to 3 instances with MongoDB, given a QoS threshold of 90%. The corresponding numbers are 2 and 2 for the in-memory analytics application.

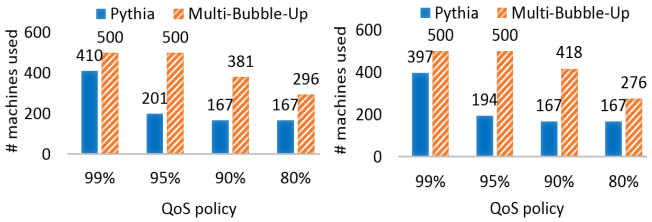
Table 2: Normalized IPC of latency-sensitive workloads when co-located with one or more instances of a memory-intensive batch application.

	Graph analytics			In-memory analytics		
# Co-located application instances	1	2	3	1	2	3
Redis	0.9521	0.9200	0.7706	0.9523	0.9355	0.6061
MongoDB	0.9770	0.9418	0.9117	0.9806	0.9860	0.6854

5.7 Improvement in Cluster Utilization

We now evaluate if Pythia can improve the utilization of a cluster and compare it with Bubble-Up and Multi-Bubble-Up. The utilization is measured as the number of cores on which some application process is running divided by the total number of available cores. A higher value is better. We start with a cluster of 500 servers where we restrict the scheduler to use a single socket with 8 cores for co-location (*i.e.*, 500 × 8=40,000 available cores in total), leaving aside the other socket for the experimental book-keeping. Experiments were initialized by placing an LS workload on each server, using two cores each. A randomly selected incoming batch workload (from the set of 19 batch workloads) is scheduled by Pythia. We run this experiment for a total of 500 arrivals of batch workloads. The QoS policy for the LS workload is specified as x%, which means that the minimum acceptable QoS is x% of the QoS when the workload is run without interference. So the higher the value of x is, the stricter is the QoS requirement. Pythia would co-locate the incoming batch workload on one of the servers if the estimated combined contention does not violate the QoS policy. Otherwise, the workload would be scheduled on a separate server. Since we use two cores for the LS workload and two for each of the batch workloads, the maximum degree of co-location possible is three. These experiments were ran separately with Redis and MongoDB as the LS workload and for different QoS policies.

Fig. 16 shows that Pythia significantly improves the utilization compared to Bubble-Up and Multi-Bubble-Up. Utilization with Bubble-Up always remains 0.5 because it can handle at most one batch workload along with the LS workload and both together



(a) # Servers for MongoDB *(b) # Servers for Redis*

Figure 17: Number of servers used for various QoS policies.

occupy 4 cores out of 8 available. When the QoS policy is very strict such as 99%, both Pythia and Multi-Bubble-Up find it difficult to identify a suitable server for co-location because the combined contention has to be small. Hence utilization does not increase much, but still Pythia outperforms Multi-Bubble-Up. For a more relaxed QoS policy, such as 90%, Pythia can more aggressively co-locate workloads and achieves near perfect utilization of 99% for both Redis and MongoDB. In contrast, Multi-Bubble-Up is only able to achieve a utilization of 72% and 68%, respectively for Redis and MongoDB. For a 95% QoS policy, Pythia achieves a gain of 71% in utilization over Multi-Bubble-Up.

Fig. 17 shows the corresponding number of servers used to schedule all the 500 incoming batch workloads. Ability to pack jobs on lesser number of servers translates to energy savings as empty servers can be put into an idle or a low-power state. Since we allow at most three co-located batch workloads in a server, the minimum number of servers required for co-location is 500/3∼167. Note that for the 80% QoS policy, Pythia exactly uses this minimum number of required servers, while Multi-Bubble-Up uses 65% more servers. Pythia used the Best Fit algorithm for finding the best co-location spot but using the Worst Fit (use the server where there will be the greatest headroom left) or the First Fit (use the first server with available space) algorithm showed similar results.

6 DISCUSSION

Co-location scenario: We define the co-location scenario as one LS workload and multiple batch workloads. In general, our contention characterization can adapt to the co-location scenario with multiple LS workloads by considering one as LS workload and the other as batch workloads and vice-versa. However, the scheduling algorithm will be more complicated when an additional LS workload is scheduled and its service-level objectives (SLO) should also be guaranteed.

Heterogeneous Clusters: Pythia can handle non-homogeneous clusters by training its model separately for each type of processor architecture in the cluster. This does linearly increase the profiling and training cost in proportion to the number of architectures. Such training cost can be reduced by modeling the contention score as a function of the architectural parameters, such as, core count, cache-size, TLB size, and memory bandwidth. Thus, a bubble-size or other model parameter measured in one type of processor can be translated to another type of processor architecture.

Contentions Along Network and Disk I/O: Pythia focuses on contentions in memory subsystems for which no proper isolation mechanism is yet available on commercial processors. However, our approach can be extended to include network and disk resources by designing bubbles in those resource dimensions. Pythia's regression-based combined contention prediction model is generic and should be effective for any type of contention as long as the individual contentions from each batch workload is effectively captured.

7 RELATED WORK

Isolation mechanisms: A large body of work focuses on isolation techniques for CPU [24, 31, 59], cache [26, 30, 46, 55], memory controllers [19, 43], and network [27, 51]. Heracles [34] is a feedback-based dynamic controller that combines best isolation mechanisms along different dimensions and uses special hardware features (such as Intel's Cache Allocation Technology) to meet latency targets for the LS workloads. However, many of these techniques are still not available in the off the shelf machines that is used in most of datacenters. Novakovic *et al.* [44], Maji *et al.* [36, 37], and Leverich *et al.* [32] showed that performance degradation due to interference is still a reality, especially due to contention in the LLC [52]. Pythia is orthogonal to these works as in the presence of imperfect isolation, it helps to find the best workload placement.

Performance prediction and control: Several prior works focused on predicting application performance [17, 18, 32, 38, 41, 56, 59, 59]. Paragon [17] predicts workload performance on various target architectures and server configurations through collaborative filtering. Quasar [18] extends that idea to predict performance even in scale-out and scale-up architectures. CPI^2 [59] uses cycle per instruction (CPI) to monitor and detect degraded performance. DeepDive [44] detects interference between co-located VMs and mitigates the problem by migrating the VM to a new host and thus can be very costly. IC^2 [37] shows how to detect and mitigate interference in clouds through application reconfiguration. Both are reactive systems as opposed to our predictive scheduling model.

8 CONCLUSION

Co-location of multiple workloads on the same server can greatly improve cluster utilization. However, co-location increases the risk of violating the QoS guarantee for the latency sensitive workloads due to contention on the shared resources. Here we present Pythia, a co-location manager that can achieve better scheduling by accurately predicting the severity of contention from multiple applications and its effect on the latency-sensitive workload. It handles changes in the application characteristic through limited use of dynamic scheduling, monitoring, and control.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1513197 and CNS-1527262 and gift funding from Adobe Research and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2010. Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space. http://www.gartner.com/newsroom/id/1472714.

[2] 2018. Intel Xeon Platinum Processor. https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/platinum-processors.html
[3] 2018. JSON (Java Script Object Notation). https://www.json.org/
[4] 2018. Moab Cloud HPC Suite. https://www.adaptivecomputing.com/moab-hpc-basic-edition/
[5] 2018. MongoDB. https://www.mongodb.com/
[6] 2018. Redis. https://redis.io/
[7] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 1–14.
[8] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.
[9] Carter Bays. 1977. A Comparison of Next-fit, First-fit, and Best-fit. *Commun. ACM* (1977).
[10] Major Bhadauria, Vincent M Weaver, and Sally A McKee. 2009. Understanding PARSEC performance on contemporary CMPs. In *IISWC*.
[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*.
[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*.
[13] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*.
[14] Richard E Brown, Richard Brown, Eric Masanet, Bruce Nordman, Bill Tschudi, Arman Shehabi, John Stanley, Jonathan Koomey, Dale Sartor, Peter Chan, et al. 2007. *Report to congress on server and data center energy efficiency: Public law 109-431*. Technical Report. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US).
[15] Dave Cole. 2012. Data center infrastructure management. *Data Center Knowledge* (2012).
[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*.
[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*.
[19] 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 *ASPLOS*.
[20] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ISCA*.
[21] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*.
[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*.
[23] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems (Eurosys)*. ACM, 99–112.
[24] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. 2006. Enforcing performance isolation across virtual machines in Xen. In *Middleware*.
[25] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (2006).
[26] Lisa R Hsu, Steven K Reinhardt, Ravishankar Iyer, and Srihari Makineni. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*.
[27] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical network performance isolation at the edge. In *NSDI*.
[28] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. 2012. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 51.
[29] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 158–169.
[30] Harshad Kasture and Daniel Sanchez. 2014. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS*.
[31] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. 2009. Task-aware virtual machine scheduling for I/O performance. In *VEE*.
[32] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *EuroSys*.
[33] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. 2013. NUMA-aware algorithms: the case of data shuffling.. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*. 1–10.
[34] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *ISCA*.
[35] Luiz Andre Barroso and Urs HÃűlzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (2 ed.). Morgan & Claypool Publishers.
[36] Amiya K Maji, Subrata Mitra, and Saurabh Bagchi. 2015. Ice: An integrated configuration engine for interference mitigation in cloud services. In *ICAC*.
[37] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating Interference in Cloud Services by Middleware Reconfiguration. In *Middleware*.
[38] 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 *MICRO*.
[39] Jason Mars, Lingjia Tang, and Mary Lou Soffa. 2011. Directly characterizing cross core interference through contention synthesis. In *HiPEAC*.
[40] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2010. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 257–265.
[41] Subrata Mitra, Greg Bronevetsky, Suhas Javagal, and Saurabh Bagchi. 2015. Dealing with the unknown: Resilience to prediction errors. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 331–342.
[42] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Matthias S Müller. 2011. Memory performance and SPEC OpenMP scalability on quad-socket x86_64 systems. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 170–181.
[43] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*.
[44] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *ATC*.
[45] National Resources Defense Council (NRDC). 2014. Data Center Efficiency Assessment. https://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf.
[46] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*.
[47] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. 2014. Stay-away, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference*. ACM, 301–312.
[48] 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 (2010), 65–79.
[49] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA*.
[50] Simon J Sheather and Michael C Jones. 1991. A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society. Series B (Methodological)* (1991), 683–690.
[51] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. 2010. Seawall: performance isolation for cloud datacenter networks. In *HotCloud*.
[52] 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 *MICRO*.
[53] Christian Terboven, Dirk Schmidl, Henry Jin, Thomas Reichstein, et al. 2008. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?* ACM, 377–384.
[54] A. Vasan, Anand Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. 2010. Worth their watts? - an empirical study of datacenter servers. In *HPCA*.
[55] Yuejian Xie and Gabriel H Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*.
[56] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *International Symposium on Computer Architecture (ISCA)*. 607–618.
[57] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading.. In *USENIX Annual Technical Conference*. 309–322.

[58] Ce Zhang and Christopher Ré. 2014. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1283–1294.
[59] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *EuroSys*.
[60] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *MICRO*.