Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters

Christina Delimitrou and Christos Kozyrakis
Electrical Engineering Department, Stanford University
{cdel, kozyraki}@stanford.edu

ABSTRACT

Large-scale datacenters (DCs) host tens of thousands of diverse applications each day. However, interference between colocated workloads and the difficulty to match applications to one of the many hardware platforms available can degrade performance, violating the quality of service (QoS) guarantees that many cloud workloads require. While previous work has identified the impact of heterogeneity and interference, existing solutions are computationally intensive, cannot be applied online and do not scale beyond few applications.

We present Paragon, an online and scalable DC scheduler that is heterogeneity and interference-aware. Paragon is derived from robust analytical methods and instead of profiling each 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 in a manner that minimizes interference and maximizes server utilization. Paragon scales to tens of thousands of servers with marginal scheduling overheads in terms of time or state.

We evaluate Paragon with a wide range of workload scenarios, on both small and large-scale systems, including 1,000 servers on EC2. For a 2,500-workload scenario, Paragon enforces performance guarantees for 91% of applications, while significantly improving utilization. In comparison, heterogeneity-oblivious, interference-oblivious and least-loaded schedulers only provide similar guarantees for 14%, 11% and 3% of workloads. The differences are more striking in oversubscribed scenarios where resource efficiency is more critical.

Categories and Subject Descriptors: C.5.1 [Computer System Implementation]: Super (very large) computers; C.1.3 [Processor Architectures]: Heterogeneous (hybrid) systems, D.4.1 [Process Management]: Scheduling

General Terms: Design, Performance

Keywords: Datacenter, cloud computing, heterogeneity, interference, scheduling, QoS

1. INTRODUCTION

An increasing amount of computing is performed in the cloud, primarily due to cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [4]. Large-scale providers such as Amazon EC2 [13], Microsoft Windows Azure [43], Rackspace [33] and Google Compute Engine [17] host tens of thousands of applications on a daily basis. Several companies also organize their IT infrastructure as private clouds, using management systems such as VMware vSphere [41] or Citrix XenServer [47].

The operator of a cloud service must schedule the stream of incoming applications on available servers in a manner that achieves both fast execution (user's goal) and high *resource efficiency* (operator's goal), enabling better scaling at low cost. This scheduling problem is particularly difficult as cloud services must accommodate a diverse set of workloads in terms of resource and performance requirements [4]. Moreover, the operator often has no a priori knowledge of workload characteristics. In this work, we focus on two basic challenges that complicate scheduling in large-scale DCs: *hardware platform heterogeneity* and *workload interference*.

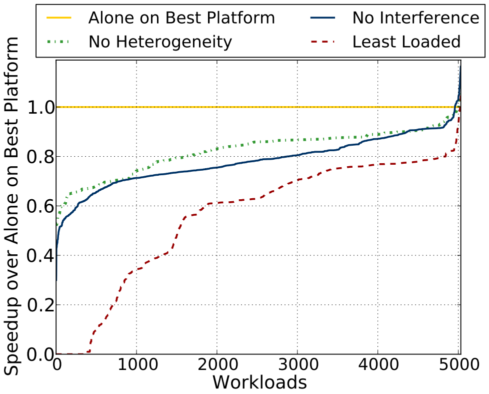


Figure 1: Performance degradation for 5,000 applications on 1,000 EC2 servers with heterogeneity-oblivious, interference-oblivious and least-loaded schedulers compared to ideal scheduling (application runs alone on best platform). Results are ordered from worst to best-performing workload.

Heterogeneity occurs because servers are gradually provisioned and replaced over the typical 15-year lifetime of a DC [4, 19, 23, 27, 31]. At any point in time, a DC may host 3-5 server generations with a few hardware configurations per generation, in terms of the specific speeds and capacities of the processor, memory, storage and networking subsystems. Hence, it is common to have 10 to 40 configurations throughout the DC. Ignoring heterogeneity can lead to significant inefficiencies, as some workloads are sensitive to the hardware configuration. Figure 1 shows that a heterogeneity-oblivious scheduler will slow applications down by 22% on average, with some running nearly 2x slower (see Section 4 for the experimental methodology). This is not only suboptimal from the user's perspective, but also for the DC operator as workloads occupy servers for significantly longer.

Interference is the result of co-scheduling multiple workloads on a single server to increase utilization and achieve better cost efficiency. By co-locating applications a given number of servers can host a larger set of workloads (better scalability). Alternatively, by packing workloads in a small number of servers when the overall load is low, the rest of the servers can be turned off to save energy. The latter is needed because modern servers are not energy-proportional and consume a large fraction of peak power even at low utilization [3, 4, 25, 29]. Co-scheduled applications may interfere negatively even if they run on different processor cores because they share caches, memory channels, storage and networking devices [18, 28, 32]. Figure 1 shows that an interference-oblivious scheduler will slow workloads down by 34% on average, with some running more than 2x slower.

Again, this is undesirable for both users and operators. Finally, a scheduler that is both interference and heterogeneity-oblivious and schedules applications to least-loaded servers is even worse (48% average slowdown), causing some workloads to crash due to resource exhaustion on the server.

Previous work has showcased the potential of heterogeneity and interference-aware scheduling [27, 28]. However, their techniques rely on detailed application characterization and cannot scale to large DCs that receive tens of thousands of potentially unknown workloads every day [9]. Most cloud management systems have some notion of contention or interference-awareness [20, 32, 39, 40, 46]. However, they either use empirical rules for interference management or assume long-running workloads (e.g., online services), whose repeated behavior can be progressively modeled. In this work, we target both heterogeneity and interference and assume no a priori analysis or knowledge of the application. Instead, we leverage information the system *already* has about the large number of applications it has previously seen.

We present *Paragon*, an online and scalable datacenter scheduler that is heterogeneity and interference-aware. The key feature of Paragon is its ability to quickly and accurately classify an unknown application with respect to heterogeneity (which server configurations it will perform best on) and interference (how much interference it will cause to co-scheduled applications and how much interference it can tolerate itself in multiple shared resources). Paragon's classification engine exploits existing data from previously scheduled applications and offline training and requires only a minimal signal about a new workload. Specifically, it is organized as a low-overhead recommendation system similar to the one deployed for the Netflix Challenge [6], but instead of discovering similarities in users' movie preferences, it finds similarities in applications' preferences with respect to heterogeneity and interference. It uses singular value decomposition to perform collaborative filtering and identify similarities between incoming and previously scheduled workloads.

Once an incoming application is classified, a greedy scheduler assigns it to the server that is the best possible match in terms of platform and minimum negative interference between all co-scheduled workloads. Even though the final step is greedy, the high accuracy of classification leads to schedules that satisfy both user requirements (fast execution time) and operator requirements (efficient resource use). Moreover, since classification is based on robust analytical methods and not merely empirical observation, we have strong guarantees on its accuracy and strict bounds on its overheads. Paragon scales to systems with tens of thousands of servers and tens of configurations, running large numbers of previously unknown workloads.

We implemented Paragon and evaluated its efficiency using a wide spectrum of workload scenarios (light, high, and oversubscribed). We used Paragon to schedule applications on a private cluster with 40 servers of 10 different configurations and on 1000 exclusive servers on Amazon EC2 with 14 configurations. We compare Paragon to a heterogeneity-oblivious, an interference-oblivious and a state-of-the-art least-loaded scheduler. For the 1000-server experiments and a scenario with 2500 workloads, Paragon maintains QoS for 91% of workloads (within 5% of their performance running alone on the best server). The heterogeneity-oblivious, interference-oblivious and least-loaded schedulers offer such QoS guarantees for only 14%, 11%, and 3% of applications respectively. The results are more striking in the case of an oversubscribed workload scenario, where efficient resource use is even more critical. Paragon provides QoS guarantees for 52% of workloads and bounds the performance degradation to less than 10% for an additional 33% of workloads. In contrast, the least-loaded scheduler dramatically degrades performance for 99.9% of applications. We also evaluate Paragon on a Windows Azure and a Google Compute Engine cluster and show similar gains. Finally, we validate that Paragon's classification engine achieves the accuracy and bounds predicted by the analytical methods and evaluate various parameters of the system.

The rest of the paper is organized as follows. Section 2 describes the analytical methods that drive Paragon. Section 3 presents the implementation of the scheduler. Section 4 presents the experimental methodology and Section 5 the evaluation of Paragon. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2. FAST & ACCURATE CLASSIFICATION

The key requirement for heterogeneity and interference-aware scheduling is to quickly and accurately classify incoming applications. First, we need to know how fast an application will run on each of the tens of server configurations available. Second, we need to know how much interference it can tolerate from other workloads in each of several shared resources without significant performance loss and how much interference it will generate itself. Our goal is to perform online scheduling for large-scale DCs without any a priori knowledge about incoming applications. Most previous schemes address this issue with detailed but offline application characterization or long-term monitoring and modeling approaches [28, 32, 39]. Instead, Paragon takes a different perspective. Its core idea is that, instead of learning each new workload in detail, the system leverages information it already has about applications it has seen to express the new workload as a combination of known applications. For this purpose we use collaborative filtering techniques that combine a minimal profiling signal about the new application (e.g., a minute's worth of profiling data on two servers) with the large amount of data available from previously scheduled applications. The result is fast and highly accurate classification of incoming applications with respect to both heterogeneity and interference. Within a minute of its arrival, an incoming workload can be scheduled efficiently on a large-scale cluster.

2.1 Collaborative Filtering Background

Collaborative filtering techniques are frequently used in recommendation systems. We will use one of their most publicized applications, the Netflix Challenge [6], to provide a quick overview of the two analytical methods we rely upon, Singular Value Decomposition (SVD) and PQ-reconstruction (PQ) [34]. In this case, the goal is to provide valid movie recommendations for Netflix users given the ratings they have provided for various other movies.

The input to the analytical framework is a sparse matrix A, the *utility matrix*, with one row per user and one column per movie. The elements of A are the ratings that users have assigned to movies. Each user has rated only a small subset of movies; this is especially true for new users which may only have a handful of ratings or even none. While there are techniques that address the cold start problem, i.e., providing recommendations to a completely fresh user with no ratings, here we focus on users for which the system has some minimal input. If we can estimate the values of the missing ratings in the sparse matrix A, we can make movie recommendations: suggest that users watch the movies for which the recommendation system estimates that they will give high ratings with high confidence.

The first step is to apply singular value decomposition (SVD), a matrix factorization method used for dimensionality reduction and similarity identification. Factoring A produces the decomposition to matrices U, V and Σ.

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} = U \cdot \Sigma \cdot V^T$$

where

$$U_{m\times r} = \begin{pmatrix} u_{11} & \cdots & u_{1r} \\ u_{21} & \cdots & u_{2r} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mr} \end{pmatrix}, V_{n\times r} = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{r1} & v_{r2} & \cdots & v_{rn} \end{pmatrix}$$

$$\Sigma_{r\times r} = \begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_r \end{pmatrix}$$

are the matrices of left and right singular vectors and the diagonal matrix of singular values.

Dimension r is the rank of matrix A and it represents the number of similarity concepts identified by SVD. For instance, one similarity concept may be that certain movies belong to the drama category, while another may be that most users that liked the movie "Lord of the Rings 1" also liked "Lord of the Rings 2". Similarity concepts are represented by singular values (σ_i) in matrix Σ and the confidence in a similarity concept by the magnitude of the corresponding singular value. Singular values in Σ are ordered by decreasing magnitude. Matrix U captures the strength of the correlation between a row of A and a similarity concept. In other words, it expresses how users relate to similarity concepts such as the one about liking drama movies. Matrix V captures the strength of the correlation of a column of A to a similarity concept. In other words, to what extend does a movie fall in the drama category. The complexity of performing SVD on a $m \times n$ matrix is $min(n^2m, m^2n)$. SVD is robust to missing entries and imposes relaxed sparsity constraints to provide accuracy guarantees. Density less than 1% does not reduce the recommendation accuracy [38].

Before we can make accurate score estimations using SVD, we need the full utility matrix A. To recover the missing entries in A, we use PQ-reconstruction. Building from the decomposition of the initial, sparse A matrix we have $Q_{m\times r} = U$ and $P^T_{r\times n} = \Sigma \cdot V^T$. The product of Q and P^T gives matrix R which is an approximation of A with the missing entries. To improve R, we use Stochastic Gradient Descent (SGD), a scalable and lightweight latent factor model [7, 24, 26, 44] that iteratively recreates A:

```
$\forall r_{ui}$, where $r_{ui}$ an element of the reconstructed matrix $R$
    $\epsilon_{ui} = r_{ui} - q_i \cdot {p_u}^T$
    $q_i \leftarrow q_i + \eta(\epsilon_{ui} p_u - \lambda q_i)$
    $p_u \leftarrow p_u + \eta(\epsilon_{ui} q_i - \lambda p_u)$
until $|\epsilon|_{L_2} = \sqrt{\sum_{u,i} |\epsilon_{ui}|^2}$ becomes marginal.
```

In the process above η is the learning rate and λ is the regularization factor. The complexity of PQ is linear with the number of r_{ui} and in practice takes up to a few ms for matrices with $m, n \sim 1,000$. Once the dense utility matrix R is recovered we can make movie recommendations. This involves applying SVD to R to identify which of the reconstructed entries reflect strong similarities that enable making accurate recommendations with high confidence.

2.2 Classification for Heterogeneity

Overview: We use collaborative filtering to identify how well an incoming application will run on the different hardware platforms available. In this case, the rows in matrix A represent applications, the columns server configurations (SC) and the ratings represent normalized application performance on each server configuration.

As part of an offline step, we select a small number of applications, a few tens, and profile them on all different server configurations. We normalize the performance results and fully populate the corresponding rows of A. This only needs to happen once. If a new configuration is added in the DC, we need to profile these applications on it and add a column in A. In the online mode, when a new application arrives, we profile it for a period of 1 minute on any two server configurations, insert it as a new row in matrix A and use the process described in Sec. 2.1 to derive the missing ratings for the other server configurations.

Metric	Applications (%)			
	ST	MT	MP	IO
Selected best SC	86%	86%	83%	89%
Selected SC within 5% of best	91%	90%	89%	92%
Correct SC ranking (best to worst)	67%	62%	59%	43%
90% correct SC ranking	78%	71%	63%	58%
50% correct SC ranking	93%	91%	89%	90%
Training & best SC match	28%	24%	18%	22%

Table 1: Validation metrics for heterogeneity classification.

In this case, Σ represents similarity concepts such as the fact that applications that benefit from SC1 will also benefit from SC3. U captures how an application correlates to the different similarity concepts and V how an SC correlates to them. Collaborative filtering identifies similarities between new and known applications. Two applications can be similar in one characteristic (they both benefit from high clock frequency) but different in others (only one benefits from a large L3 cache). This is especially common when scaling to large application spaces and several hardware configurations. SVD addresses this issue by uncovering hidden similarities and filtering out the ones less likely to have an impact on the application's behavior.

The size of the offline training set is important as a certain number of ratings is necessary to satisfy the sparsity constraints of SVD. However, over that number the accuracy quickly levels off and scales well with the number of applications thereafter (smaller fractions for training sets of larger application spaces). For our experiments we use 20 and 50 offline workloads for a 40 and 1,000-server cluster respectively. Additionally, as more incoming applications are added in A the density of the matrix increases and the recommendation accuracy further improves. Note that online training is performed only on two SCs. This not only reduces the training overhead compared to exhaustive search but since training requires dedicated servers, it also reduces the number of servers necessary for it. In contrast, if we attempted to classify applications through exhaustive profiling, the number of profiling runs would equal the number of SCs (e.g., 40). For a cloud service with high workload arrival rates, this would be infeasible to support, underlining the importance of keeping training overheads low, something that Paragon does.

Classification is very fast. On a production-class Xeon server, this takes 10-30 msec for thousands of applications and tens of SCs. We can perform classification for one application at a time or for small groups of incoming applications (*batching*) if the arrival rate is high without impacting accuracy or speed.

Performance scores: We populate A with normalized scores that represent how well an application performs on a server configuration. We use the following performance metrics based on application type:

(a) *Single-threaded workloads:* We use instructions committed per second (IPS) as the initial performance metric. Using execution time would require running applications to completion in the profiling servers, increasing the training overheads. We have verified that using IPS leads to similar classification accuracy as using full execution time. For multi-programmed workloads we use aggregate IPS.

(b) *Multithreaded workloads:* In the presence of spin-locks or other synchronization schemes that introduce active waiting, aggregate IPS can be deceiving [1, 42]. We address this by periodically polling low-overhead performance counters, to detect changes in the register file (read and writes that would denote regular operations other than spinning) and weight-out of the IPS computation such execution segments. We have verified that scheduling with this ”useful” IPS leads to similar classification accuracy as using full execution time. When workloads are not known, or multiple workload types are present ”useful” IPS is used to drive the scheduling decisions.

The choice of IPS as the base of performance metrics is influenced our current evaluation which focuses on single-node CPU, memory and I/O intensive programs. The same methodology holds for higher-level metrics, such as queries per second (QPS), which cover complex multi-tier workloads as well.

Validation: We evaluate the accuracy of heterogeneity classification on a 40-server cluster with 10 SCs. We use a large set of single-threaded, multi-threaded, multi-programmed and I/O-bound workloads. For details on workloads and server configurations, see Section 4. The offline training set includes 20 applications selected randomly from all workload types. The recommendation system achieves 24% performance improvement for single-threaded, 20% for multi-threaded, 38% for multi-programmed, and 40% for I/O workloads on average, while some applications have a 2x performance difference. Table 1 summarizes key statistics on the classification quality. Our classifier correctly identifies the best SC for 84% of workloads and an SC within 5% of optimal for 90%. The predicted ranking of SCs is exactly correct for 58% and almost correct (single reordering) for 65% of workloads. In almost all cases 50% of SCs are ranked correctly by the classification scheme. Finally, it is important to note that the accuracy does not depend on the two SCs selected for training. The training SC matched the top performing configuration only for 20% of workloads.

We also validate the analytical methods. We compare performance predicted by the recommendation system to performance obtained through experimentation. The deviation is less than 3.8% on average.

2.3 Classification for Interference

Overview: There are two types of interference we are interested in: interference that an application can *tolerate* from pre-existing load on a server and interference the application will *cause* on that load. We detect interference due to contention on shared resources and assign a score to the sensitivity of an application to a type of interference. To derive sensitivity scores we develop several microbenchmarks, each stressing a specific shared resource with tunable intensity. We run an application concurrently with a microbenchmark and progressively tune up its intensity until the application violates its QoS, which is set at 95% of the performance achieved in isolation. Applications with high tolerance to interference (e.g., sensitivity score over 60%) are easier to co-schedule than applications with low tolerance (low sensitivity score). Similarly, we detect the sensitivity of a microbenchmark to the interference the application causes by tuning up its intensity and recording when the performance of the microbenchmark degrades by 5% compared to its performance in isolation. In this case, high sensitivity scores, e.g., over 60% correspond to applications that cause a lot of interference in the specific shared resource.

Identifying sources of interference (SoI): Co-scheduled applications may contend on a large number of shared resources. We identified ten such sources of interference (SoI) and designed a tunable microbenchmark for each one. SoIs span resources such as memory (bandwidth and capacity), cache hierarchy (L1/L2/L3 and TLBs) and network and storage bandwidth. The same methodology can be expanded to any shared resource.

Collaborative filtering for interference: We classify applications for interference tolerated and caused, using twice the process described in Sec. 2.1. The two utility matrices have applications as rows and SoIs as columns. The elements of the matrices are the sensitivity scores of an application to the corresponding microbenchmark (sensitivity to tolerated and caused interference respectively). Similarly to classification for heterogeneity, we profile a few applications offline against all SoIs and insert them as dense rows in the utility matrices. In the online mode, each new application is profiled against two randomly chosen microbenchmarks for one minute and its sensitivity scores are added in a new row in each of the matrices. Then, we use SVD and PQ reconstruction to derive the missing entries and the confidence in each similarity concept. This process performs accurate and fast application classification and provides information to the scheduler on which applications should be assigned to the same server (see Sec. 3.2).

Metric	Percentage (%)
Average sensitivity error across all SoIs	5.3%
Average error for sensitivities $< 30\%$	7.1%
Average error for sensitivities $< 60\%$	5.6%
Average error for sensitivities $> 60\%$	3.4%
Apps with $< 5\%$ error	ST: 65% MT: 58%
Apps with $< 10\%$ error	ST: 81% MT: 63%
Apps with $< 20\%$ error	ST: 90% MT: 89%
SoI with highest error	
for ST: L1 i-cache	15.8%
for MT: LLC capacity	7.8%
Frequency L1 i-cache used as offline SoI	14.6%
Frequency LLC cap used as offline SoI	11.5%
SoI with lowest error	
for ST: network bandwidth	1.8%
for MT: storage bandwidth	0.9%

Table 2: Validation metrics for interference classification.

Validation: We evaluated the accuracy of interference classification using the single-threaded and multi-threaded workloads and the same systems as for the heterogeneity classification. Table 2 summarizes some key statistics on the classification quality. Our classifier, achieves an average error of 5.3% in estimating both tolerated and caused interference across all SoIs. For high values of sensitivity, i.e., applications that tolerate and cause a lot of interference, the error is even lower (3.4%), while for most applications (both single-threaded and multi-threaded) the errors are lower than 5%. The SoIs with the highest errors are the L1 instruction cache for single-threaded workloads and the LLC capacity (L2 or L3) for multi-threaded workloads. The high errors are not a weakness of the classification, since both resources are profiled adequately, but rather of the difficulty to consistently characterize contention in certain shared resources [28]. On the other hand, network and storage bandwidth have the lowest errors, primarily due to the fact that we used CPU and memory intensive workloads for this evaluation.

2.4 Putting It All Together

Overall, Paragon requires two short runs (∼1 minute) on two SCs to classify incoming applications for heterogeneity. Another two short runs against two microbenchmarks on a high-end SC are needed for interference classification. We use a high-end SC to decouple server features from interference analysis. Running for 1 minute provides some signal on the new workload without introducing significant profiling overheads. In Section 3.4 we discuss the issue of workload phases, i.e., transient effects that do not appear in the 1 minute profiling period. Next, we use collaborative filtering to classify the application in terms of heterogeneity and interference, tolerated and caused. This cumulatively requires a few msec even when considering thousands of applications and several tens of SCs or SoIs. The classification for heterogeneity and interference is performed *in parallel*. For the applications we considered, the overall profiling and classification overheads are 1.2% and 0.09% on average.

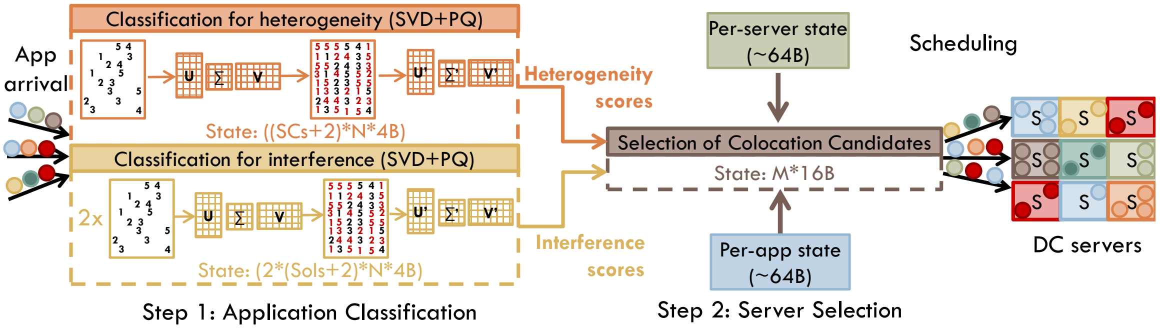


Figure 2: The components of Paragon and the state maintained by each component. Overall, the state requirements are marginal and scale linearly or logarithmically with the number of applications (N), servers (M) and configurations.

Using analytical methods for classification has two benefits; first, we have *strong analytical guarantees* on the quality of the information used for scheduling, instead of relying mainly on empirical observations. The analytical framework provides low and tight error bounds on the accuracy of classification, statistical guarantees on the quality of colocation candidates and detailed characterization of system behavior. Moreover, the scheduler design is workload independent, which means that the analytical or statistical properties the scheme provides hold for any workload. Second, these methods are *computationally efficient*, *scale well* with the number of applications and SCs, do not introduce significant training and decision overheads and enable exact complexity evaluation.

3. PARAGON

3.1 Overview

Once an incoming application is classified with respect to heterogeneity and interference, Paragon schedules it on one of the available servers. The scheduler attempts to assign each workload to the server of the best SC and colocate it with applications so that interference is minimized for workloads running on the same server. The scheduler is online and greedy so we cannot make holistic claims about optimality. Nevertheless, the fact that we start with highly accurate classification helps achieve very efficient schedules. The interference information allows Paragon to pack applications on a subset of servers without significant performance loss[1]. The heterogeneity information allows Paragon to assign to each SC only applications that will benefit from its characteristics. Both these properties lead to faster execution, hence resources are freed as soon as possible, making it easier to schedule future applications (more unloaded servers) and perform power management (more idling servers that can be placed in low-power modes).

Fig. 2 presents an overview of Paragon and its components. The scheduler maintains per-application and per-server state. Per-application state includes information for the heterogeneity and interference classification of every submitted workload. For a DC with 10 SCs and 10 SoIs, we store 64B per application. The per-server state records the IDs of applications running on a server and the cumulative sensitivity to interference (roughly 64B per server). The per-server state needs to be updated as applications are scheduled and, later on, complete. Paragon also needs some storage for the intermediate and final utility matrices and temporary storage for ranking possible candidate servers for an incoming application. Overall, state overheads are marginal and scale logarithmically or linearly with the number of applications (N) and servers (M). In our experiments with thousands of applications and servers, a single server could handle all processing and storage requirements of scheduling[2].

We present two methods for selecting candidate servers; a fast, greedy algorithm that searches for the optimal candidate and a statistical scheme of constant runtime that provides strong guarantees on the quality of candidates as a function of examined servers.

3.2 Greedy Server Selection

In examining candidates, the scheduler considers two factors: first, which assignments minimize negative interference between the new application and existing load and second, which servers have the best SC for this workload. Decisions are made in this order; first identifying servers that do not violate QoS and then selecting the best SC between them. This is based on the observation that interference typically leads to higher performance loss than suboptimal SCs.

The greedy scheduler strives to minimize interference, while also increasing server utilization. The scheduler searches for servers whose load can tolerate the interference caused by the new workload and vice versa, the new workload can tolerate the interference caused by the server load. Specifically it evaluates two metrics, $D_1 = t_{server} - c_{newapp}$ and $D_2 = t_{newapp} - c_{server}$, where t is the sensitivity score for tolerated and c for caused interference for a specific SoI. The cumulative sensitivity of a server to caused interference is the sum of sensitivities of individual applications running on it, while the sensitivity to tolerated interference is the minimum of these values. The optimal candidate is a server for which D_1 and D_2 are exactly zero for all SoIs. This implies that there is no negative impact from interference between new and existing applications and that the server resources are perfectly utilized. In practice, a good selection is one for which D_1 and D_2 are bounded by a positive and small ϵ for all SoIs. Large, positive values for D_1 and D_2 indicate suboptimal resource utilization. Negative D_1 and/or D_2 imply violation of QoS and identify poor candidates that should be avoided.

We examine candidate servers for an application in the following way. The process is explained for interference tolerated by the server and caused by the new workload (D_1) and is exactly the same for D_2. Given the classification of an application, we start from the resource that is most difficult to satisfy (highest sensitivity score to caused interference). We query the server state and select the server set for which D_1 is non-negative for this SoI. Next, we examine the second SoI in order of decreasing sensitivity scores, filtering out any servers for which D_1 is negative. The process continues until all SoIs have been examined. Then, we take the intersection of candidate server sets for D_1 and D_2. We now consider heterogeneity. From the set of candidates we select servers that correspond to the best SC for the new workload and from their subset we select the server with $min(||D_1 + D_2||_{L1})$.

[1] Packing applications with minimal interference should be a property exhibited by any optimal schedule.

[2] Additional scheduling servers can be used for fault-tolerance.

As we filter out servers, it is possible that at some point the set of candidate servers becomes empty. This implies that there is no single server for which D_1 and D_2 are non-negative for some SoI. In practice this event is extremely unlikely, but is supported for completeness. We handle this case with backtracking. When no candidates exist the algorithm reverts to the previous SoI and relaxes the QoS constraints until the candidate set becomes non empty, before it continues. If still no candidate is found backtracking is extended to more levels. Given M servers, the worst-case complexity of the algorithm is $O(M \cdot SoI^2)$, since theoretically backtracking might extend all the way to the first SoI. In practice, however, we observe that for a 1000-server system, 89% of applications were scheduled without any backtracking. For 8% of these, backtracking led to negative D_1 or D_2 for a single SoI and for 3% for multiple SoIs. Additionally, we bound the runtime of the greedy search using a timeout mechanism, after which the best server from the ones already examined is selected in the way previously described (best SC and minimum interference deviation). In our experiments timeouts occurred in less than 0.1% of applications and resulted in a server within 10% of optimal.

3.3 Statistical Framework for Server Selection

The greedy algorithm selects the best server for an application - or a near-optimal server. However, for very large DCs, e.g., 10-100k servers, the overhead from examining the server state in the first step of the search might become high. Additionally, the results depend on the active workloads and do not allow strict guarantees on the server quality under any scenario. We now present an alternative, statistical framework for server selection in very large DCs based on sampling, which has constant runtime and enables such guarantees.

Instead of examining the entire server state we sample a small number of servers. We use cryptographic hash functions to introduce randomness in the server selection. We hash the scores of tolerated interference of each server using variations of SHA-1 [22] as different hash functions (h_j) for each SoI to increase entropy. The input to a h_j is a sensitivity score for an SoI and the output a hashed value of that score. Outputs have the same precision as inputs (14bits). This process is done once, unless the load of a server changes. When a new application arrives, we obtain candidate servers by hashing its sensitivity scores to caused interference for each SoI. For example, the input to h_1 for SoI 1 is a. The output will be a new number, b which corresponds to server ID u. Re-hashing b obtains additional IDs of candidate servers. This produces a random subset of the system's servers. After a number of re-hashes the algorithm ranks the examined servers and selects the best one. Candidates are ranked by *colocation quality*, which is a metric of how suitable a given server is for a new workload. For candidate i, colocation quality is defined as:

$$Q_i = [sign(\sum^{SoIs} (t-c)_i)]|1 - ||t-c||_1| =$$

$$[sign(\sum_{k=1}^{SoIs} (t(k)-c(k))_i)]|1 - \sum_{k=1}^{SoIs} |t(k)-c(k)|_i|$$

t is the original, unhashed sensitivity to tolerated interference for a server and c the original sensitivity to caused interference for the new workload. The $sign$ in Q_i reflects whether a server preserves (positive) or violates QoS (negative). The L1 norm of $(t-c)$ reflects how closely the server follows the application's requirements and is normalized to its maximum value, 10, which happens when for all ten SoIs $t = 100\%$ and $c = 0$. High and positive Q_i values reflect better candidates, as the deviation between t and c is small for all SoIs. Poor candidates have small Q_i or even negative when they violate QoS in one or more SoIs. Quality is normalized to the range $[0,1]$. For example, for unnormalized qualities in the range $[-1.2, 0.8]$ and a candidate with $Q = -1.0$, the normalized quality will be: $\frac{(-1.0+|min|)}{|max|+|min|} = 0.2/2 = 0.1$.

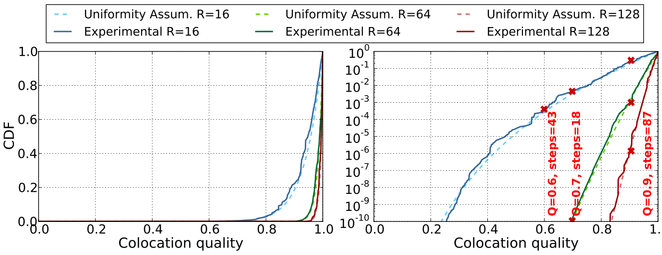


Figure 3: Colocation quality distribution ($F(x) = x^R$, where $R = 16, 64$ and 128). Fig. 3b shows the comparison between the greedy algorithm and the statistical scheme for three colocation candidates of $Q = 0.6, 0.7$ and 0.9.

We now make an assumption on the distribution of quality values, which we verify in practice. Because of the way candidate servers are selected and the independence between initial workloads, Q_i's approximate a uniform distribution, for problems with tens of thousands of servers and applications. Figure 3a shows the CDF of measured quality for 16, 64 and 128 candidates and the corresponding uniform distributions ($F(x) = x^R$, where R the number of candidates examined) in a system with 1,000 servers. In all cases, the assumption of uniformity holds in practice with small deviations. When we exceed 128 candidates (1/8 of the cluster) the distribution starts deviating from uniform. We have observed that for even larger systems, e.g., a 5,000-server Windows Azure cluster, uniform distributions extend to larger numbers of candidates (up to 512) as well. The probability of a candidate having quality a is $Pr(a) = a^R$. For example, for 128 candidates there is a 10^{-6} probability that no candidate will have quality over 0.9.

We now compare the statistical scheme with the greedy algorithm (Figure 3b). While the latter finds a server with quality Q after a random number of steps, the statistical scheme provides strong *guarantees* on the number of candidates required for the same quality. For example, for a candidate with $Q = 0.9$, the greedy algorithm needs 87 steps, but cannot provide ad hoc guarantees on the quality of the result, while the statistical scheme guarantees that for the same requirements, with 64 candidates, there is a 10^{-3} chance that no server has $Q \geq 0.9$. The guarantees become stricter as the distribution gets skewed towards 1 (more candidates). Therefore, although the statistical scheme cannot guarantee optimality, it shows that examining a small number of servers provides strict guarantees on the obtained quality and makes scheduling efficiency workload independent.

In our 1,000-server experiments, the overhead of the greedy algorithm is marginal (less than 0.1% in most cases), while the statistical scheme induces 0.5-2% overheads due to the computation required for hashing. Because at this scale the greedy algorithm is faster, all results in this work are obtained using greedy search. However, for problems of larger scale the statistical scheme can be more efficient.

3.4 Discussion

Workload phases: Application classification in Paragon is performed once for each new workload, using the information from its 1 minute profiling. It is possible that some applications will go through various phases that are not captured during profiling. Hence, the schedule will be suboptimal. We detect such workloads by monitoring their performance scores (e.g., IPS) during execution. If the monitored performance deviates significantly and for long periods of time from the performance predicted by the classification engine, the application may have changed behavior. Upon detection we do one

Server Type	GHz/sockets/cores/ L1(KB)/LLC(MB)/mem(GB)						#
Xeon L5609	1.87	2	8	32/32	12	24 DDR3	1
Xeon X5650	2.67	2	12	32/32	12	24 DDR3	2
Xeon X5670	2.93	2	12	32/32	12	48 DDR3	2
Xeon L5640	2.27	2	12	32/32	12	48 DDR3	1
Xeon MP	3.16	4	4	16/16	1	8 DDR2	5
Xeon E5345	2.33	1	4	32/32	8	32 FB-DIMM	8
Xeon E5335	2.00	1	4	32/32	8	16 FB-DIMM	8
Opteron 240	1.80	2	2	64/64	2	4 DDR2	7
Atom 330	1.60	1	2	32/24	1	4 DDR2	5
Atom D510	1.66	1	2	32/24	1	8 DDR2	1

Table 3: Main characteristics of the servers of the local cluster. The total core count is 178 for 40 servers of 10 different SCs.

of the following. First, we can avoid scheduling a large number of other workloads on the same server as the interference information for this workload is likely incorrect. Second, if there is a migration mechanism available (process or VM migration), we can clone the workload, repeat the classification from its current execution point and evaluate whether re-scheduling to another server is beneficial. Note that migration can involve significant overheads if the application operates on significant amounts of state. Section 5 includes an experiment where workload behavior experiences different phases. We assume that there exists an underlying mechanism, such as vShpere [41], that performs the live migration.

Suboptimal scheduling: A second concern apart from application phases is suboptimal scheduling, either due to the greedy selection algorithm which assigns applications to servers in a per-workload fashion, or due to pathological behavior in application arrival patterns. Suboptimal scheduling can be detected exactly as the problem of workload phases and can potentially be resolved by re-scheduling several active applications. Although re-scheduling was not needed for the examined applications, Paragon provides a general methodology to detect such deviations and leverage mechanisms like VM migration to re-schedule the sub-optimally scheduled workloads.

Latency-critical applications and workload dependencies: Finally, Paragon does not explicitly consider latency-critical applications or dependencies between application components, e.g., a multi-tier service, such as search or webmail, where tiers communicate and share data. One differentiation in this case comes from the metrics the scheduler must consider. It is possible that the interference classification should use microbenchmarks that aim to degrade the per-query latency as opposed to the workload's throughput. Another differentiation comes from the possible workload scenarios. One scenario can involve a latency-critical application running as the primary process, e.g., memcached, and the remaining server capacity being allocated to best-effort applications, such as analytics or background processes using Paragon. A different scenario is one where a throughput-bound distributed workload, e.g., MapReduce runs with high priority and the remaining server capacity is used by instances of a latency-critical application. Paragon does not currently enforce fine-grain priorities between application components or user requests, or optimize for shared data placement, which might be beneficial for these scenarios. There is related work on this topic [20] and we will consider how it interacts with Paragon in future work.

4. METHODOLOGY

Server systems: We evaluated Paragon on a small local cluster and three major cloud computing services. Our local cluster includes servers of ten different configurations shown in Table 3. We also show how many servers of each type we use. Note that these configurations range from high-end Xeon systems to low-power Atom-based boards. There is a wide range of core counts, clock frequencies and memory capacities and speeds present in the cluster.

For the cloud-based clusters we used exclusive (reserved) server instances, i.e., no other users had access to these servers. We verified that no external scheduling decisions or actions such as auto-scaling or workload migration are performed during the course of the experiments. We used 1,000 servers on Amazon EC2 [13] with 14 different SCs, ranging from small, low-power, dual-core machines to high-end, quad-socket, multi-core servers with hundreds of GBs of memory. All 1,000 machines are private, i.e., there is no interference in the experiments from external workloads. We also conducted experiments with 500 servers on Windows Azure [43] with 8 different SCs and 100 servers on Google Compute Engine [17] with 4 SCs.

Schedulers: We compared Paragon to three alternative schedulers. First, a baseline scheduler that preserves an application's core and memory requirements but ignores both its heterogeneity and interference profiles. In this case, applications are assigned to the *least-loaded* (LL) machine. Second, a *heterogeneity-oblivious* (NH) scheme that uses the interference classification in Paragon to assign applications to servers without visibility in their SCs. Finally, an *interference-oblivious* (NI) scheme that uses the heterogeneity classification in Paragon but has no insight on workload interference. The overheads for the heterogeneity and interference-oblivious schemes are the corresponding classification and server selection overheads.

Workloads: We used 29 single-threaded (ST), 22 multi-threaded (MT) and 350 multi-programmed (MP) workloads and 25 I/O-bound workloads. We use the full SPEC CPU2006 suite and workloads from PARSEC [8] (*blackscholes, bodytrack, facesim, ferret, fluidanimate, raytrace, swaptions, canneal*), SPLASH-2 [45] (*barnes, fft, lu, ocean, radix, water*), BioParallel [21] (*genenet, svm*), Minebench [30] (*semphy, plsa, kmeans*) and SPECjbb (*2, 4 and 8-warehouse instances*). For multiprogrammed workloads, we use 350 mixes of 4 applications, based on the methodology in [36]. The I/O-bound workloads are data mining applications, such as clustering and recommender systems [34], in Hadoop and Matlab running on a single-node. Workload durations range from minutes to hours. For workload scenarios with more than 426 applications we replicated these workloads with equal likelihood (1/4 ST, 1/4 MT, 1/4 MP, 1/4 I/O) and randomized their interleaving.

Workload scenarios: To explore a wide range of behaviors, we used the applications listed above to create multiple workload scenarios. Scenarios vary in the number, type and inter-arrival times of submitted applications. The load is classified based on its relation to available resources; low: the required core count is significantly lower than the available processor resources; high: the required core count approaches the load the system can support but does not surpass it; and oversubscribed: the required core count often exceeds the system's capabilities, i.e., certain machines are oversubscribed.

For the *small-scale* experiments on the local cluster we examine four workload scenarios. First, a *low load* scenario with 178 applications, selected randomly from the pool of workloads, which are submitted with 10 sec inter-arrival times. Second, a *medium load* scenario with 178 applications, randomly selected as before and submitted with inter-arrival times that follow a Gaussian distribution with $\mu = 10$ sec and $\sigma^2 = 1.0$. Third, a *high load* scenario with 178 workloads, each corresponding to a sequence of three applications with varying memory loads. Each application goes through three phases; first medium, then high and again medium memory load. Workloads are submitted with 10 sec intervals. Finally, we examine a scenario, where 178 randomly-chosen applications arrive with 1 sec intervals. Note that the last scenario is an *over-subscribed* one. After a few seconds, there are not enough resources in the system

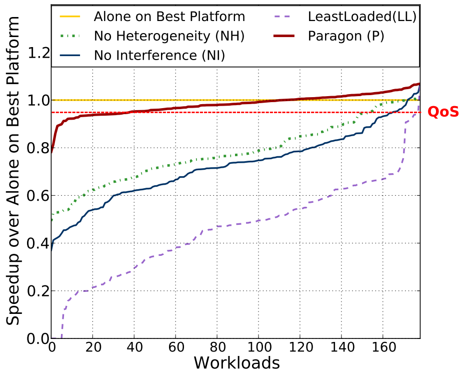


Figure 4: Performance impact from scheduling with Paragon for *medium load*, compared to heterogeneity and/or interference-oblivious schedulers. Application arrival times follow a Gaussian distribution. Applications are ordered from worst to best.

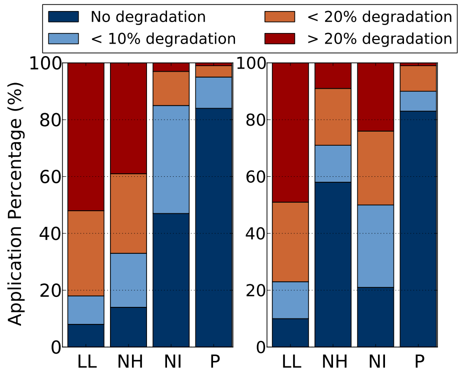


Figure 5: Breakdown of decision quality for heterogeneity (left) and interference (right) for the medium load on the local cluster. Applications are divided based on performance degradation induced by the decisions made by each of the schedulers.

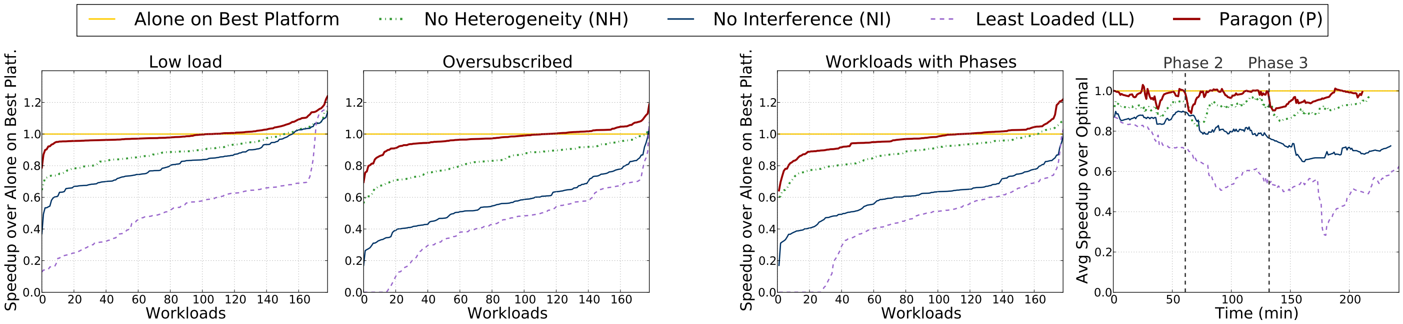


Figure 6: Performance comparison between the four schedulers for three workload scenarios: low, oversubscribed and workloads with phases (Fig. 6(a, b, c)) and performance over time for the scenario where workloads experience phases (Fig. 6d).

to execute all applications concurrently, and subsequent submitted applications are queued.

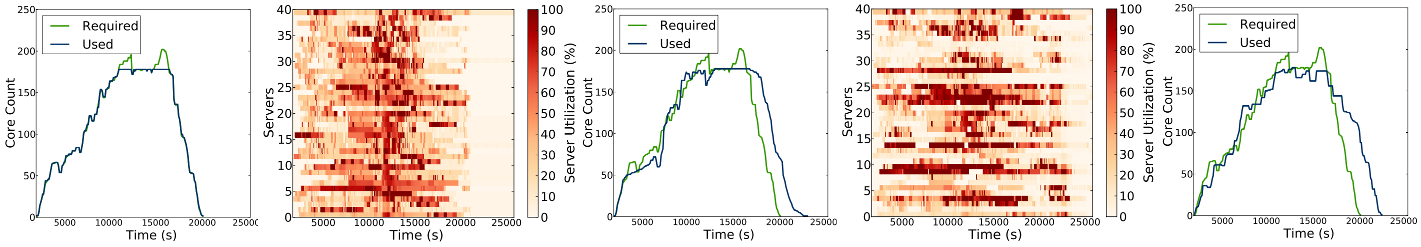
For the *large-scale* experiments on EC2 we examine three workload scenarios; a *low load* scenario with 2,500 randomly-chosen applications submitted with 1 sec intervals, a *high load* scenario with 5,000 applications submitted with 1 sec intervals and an *oversubscribed scenario* where 7,500 workloads are submitted with 1 sec intervals and an additional 1,000 applications arrive in burst (less than 0.1 sec intervals) after the first 3,750 workloads.

5. EVALUATION

5.1 Comparison of Schedulers: Small Scale

QoS guarantees: Figure 4 summarizes the performance results across the 178 workloads on the 40-server cluster for the *medium load* scenario where application arrivals follow a Gaussian distribution. Applications are ordered in the x-axis from worst to best-performing workload. The y-axis shows the performance (execution time) normalized to the performance of an application when it is running in the best platform in isolation (without interference). Each line corresponds to the performance achieved with a different scheduler. Overall, Paragon (P) outperforms the other schedulers, in terms of preserving QoS (95% of optimal performance), and bounding performance degradation when QoS requirements cannot be met. 78% of workloads maintain their QoS with Paragon, while the heterogeneity-oblivious (NH), interference-oblivious (NI) and least-loaded (LL) schedulers provide similar guarantees only for 23%, 19% and 7% of applications respectively. Even more, for the case of the least-loaded scheduler some applications failed to complete due to memory exhaustion on the server. Similarly, while the performance degradation with Paragon is smooth (94% of workloads have less than 10% degradation), the other three schedulers dramatically degrade performance for most applications, in almost linear fashion with the number of workloads. For this scenario, the heterogeneity and interference-oblivious schedulers perform almost identically, although ignoring interference degrades performance slightly more. This is due to workloads that arrive at the peak of the Gaussian distribution, when the cluster's resources are heavily utilized. For the same workloads, Paragon limits performance degradation to less than 10% in most cases. This figure also shows that few workloads experience speedups compared to their execution in isolation. This is a result of cache effects or instruction prefetching between similar co-scheduled workloads. We expect positive interference to be less prevalent for a more diverse application space.

Scheduling decision quality: Figure 5 explains why Paragon achieves better performance. Each bar represents a percentage of applications based on the performance degradation they experience due to the quality of decisions of each of the four schedulers in terms of platform selection (left) and impact from interference. Blue bars reflect good and red bars poor scheduling decisions. In terms of platform decisions, the least-loaded scheduler (LL) maps applications to servers with no heterogeneity considerations, thus it significantly degrades performance for most applications. The heterogeneity-oblivious (NH) scheduler assigns many workloads to suboptimal SCs, although fewer than LL, as it often steers workloads to high-end SCs that tend to tolerate more interference. However, as these servers become saturated, applications that would benefit from them are scheduled suboptimally and NH ends up making poor quality assignments afterwards. On the other hand, the schedulers that account for heterogeneity explicitly (interference-oblivious (NI) and Paragon (P)) have much better decision quality. NI induces no degradation to 47% of workloads and less than 10% for an additional 38%. The reason why NI does



(a) Paragon (b) No Interference (c) Least-Loaded

Figure 7: Comparison of activity and utilization between Paragon, the interference-oblivious and the least-loaded scheduler. Plots show the required and allocated core count at each moment. We also show heat maps of server utilization over time for Paragon and the interference-oblivious scheme.

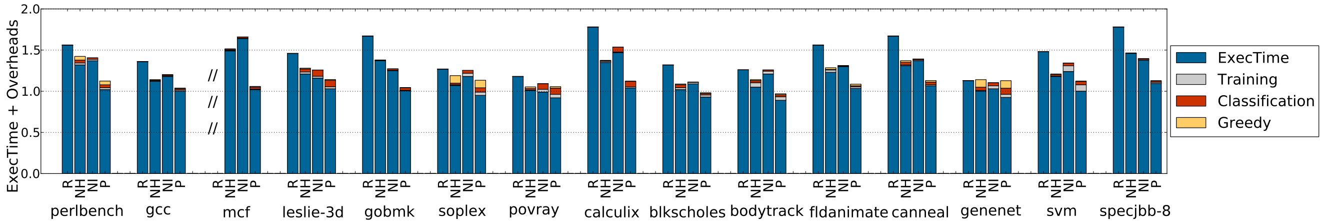


Figure 8: Execution time breakdown for selected single-threaded and multithreaded applications in the medium load scenario.

not behave better in terms of platform selection is that it has no input on interference, therefore it assigns most workloads to the best SCs. As these machines become saturated, destructive interference increases and performance degrades, although, unlike NH, which selects a random SC next, NI selects the SC that is ranked second for a workload. Finally, Paragon outperforms the other schedulers and assigns 84% of applications to their optimal SC.

The right part in Figure 5 shows decision quality with respect to interference. LL behaves the worst for similar reasons, while NI is slightly better than LL since it assigns more applications to high-end SCs, that are more likely to tolerate interference. NH outperforms NI as expected, since NI ignores interference altogether. Paragon assigns 83% of applications to servers that induce no negative interference. Considering both graphs establishes why Paragon significantly outperforms the other schedulers, as it has better decision quality both in terms of heterogeneity and interference.

Other workload scenarios: Figure 6 compares Paragon to the three schedulers for the other three scenarios; low load, oversubscribed, and workloads with phases. For low load, performance degradation is small for all schedulers, although LL degrades performance by 46% on average. Since the cluster can easily accommodate the load of most workloads, classifying incoming applications has a smaller performance impact. Nevertheless, Paragon outperforms the other three schedulers and achieves 99% of optimal performance on average. It also improves resource efficiency during low load by completing the schedule faster. For the oversubscribed scenario, Paragon guarantees QoS for the largest workload fraction, 75% and bounds degradation to less than 10% for 99% of workloads. In this case, accounting for interference is much more critical than accounting for heterogeneity as the system's resources are fully utilized.

Finally, for the case where workloads experience phases, we want to validate two expectations. First, Paragon should outperform the other schedulers, since it accounts for heterogeneity and interference (66% of workloads preserve their QoS). Second, Paragon should adapt to the changes in workload behavior, by detecting deviations from the expected IPS, re-classifying the offending workloads and re-scheduling them if a more suitable server is available. To verify this, in Fig. 6d we show the average performance for each scheduler over time. The points where workloads start changing phases are denoted with vertical lines. First, at phase change, Paragon induces much less degradation than the other schedulers, because applications are assigned to appropriate servers to begin with. Second, Paragon recovers much faster and better from the phase change. Performance rebounces to values close to 1 as the deviating workloads are re-scheduled to appropriate servers, while the other schedulers achieve progressively worse average performance.

Resource allocation: Ideally, the scheduler should closely follow application resource requirements (cores, cache capacity, memory bandwidth, etc.) and provide them with the minimum number of servers. This improves performance (applications execute as fast as possible without interference) and reduces overprovisioning (number of servers used, periods for which they are active). The latter particularly extends to the DC operator, as it reduces both capital and operational expenses. A smaller number of servers needs to be purchased to support a certain load (capital savings). During low load, many servers can be turned off to save energy (operational savings).

Figure 7a shows how Paragon follows the resource requirements for the medium load scenario shown in Figure 4. The green line shows the required core count of active applications based on arrival rate and ideal execution time and the blue line the allocated core count by Paragon. Because the scheduler tracks application behavior in terms of heterogeneity and interference it is able to follow their requirements with minimal deviation (less than 3.5%), excluding periods when the system is oversubscribed and the required cores exceed the total number of cores in the system. In comparison, NI (Figure 7b) and similarly for NH, either overprovisions or oversubscribes servers, resulting in increased execution time; per-application and for the overall scenario. Finally, Figure 7c shows the resource allocation for the least-loaded scheduler. There is significant deviation, since the scheduler ignores both heterogeneity and interference. All cores are used but in a suboptimal manner. Hence, execution times are increased for individual workloads and the overall scenario. Total execution time increases by 28%, but more importantly per-application time degrades, which is harmful both for users and DC operators.

Server utilization: In Figure 7 we also plot heat maps of the server utilization over time for Paragon and the interference-oblivious scheduler. Server utilization is defined as average CPU utilization across the cores of a server. For Paragon, utilization is high in the mid-

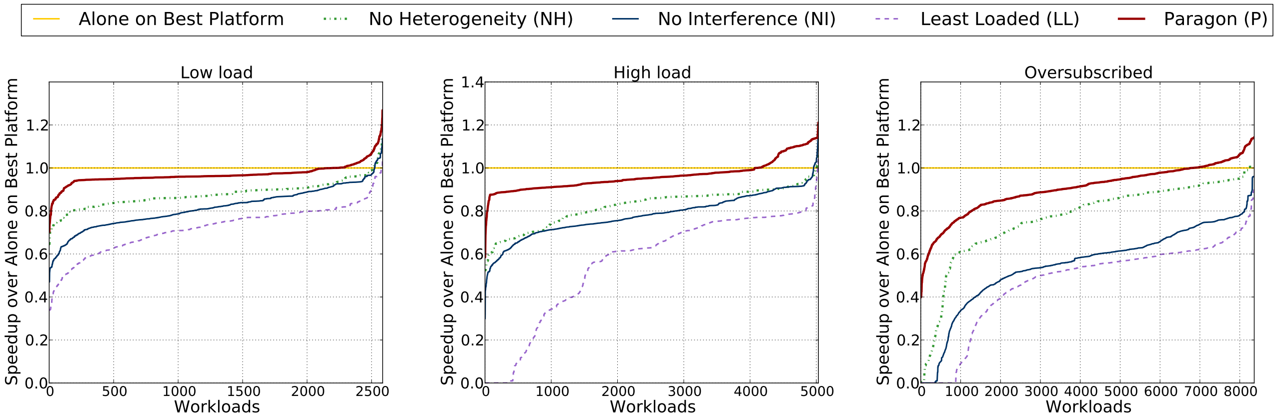


Figure 9: Performance comparison between the four schedulers, for three workload scenarios on 1,000 EC2 servers.

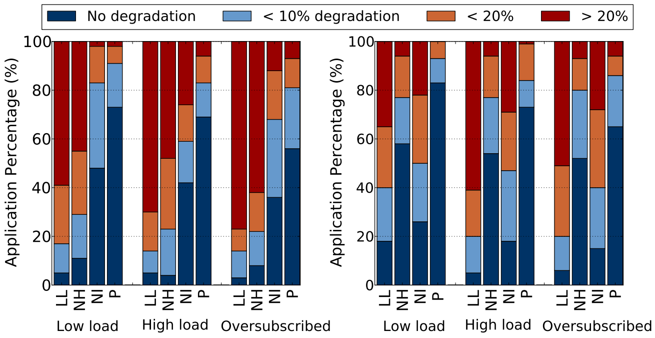


Figure 10: Breakdown of decision quality in terms of heterogeneity (left) and interference for the three EC2 scenarios.

dle of the scenario when many applications are active (47% higher than without colocation), and returns to zero when the scenario finishes. In this case, resource usage improves without performance degradation due to interference. On the other hand, NI keeps server utilization high in some servers and underutilizes others, while violating per-application QoS and extending the scenario's execution time. This is undesirable both for the user who gets lower performance and for the DC operator, since the high utilization in certain servers does not translate to faster execution time, adhering scalability to servicing more workloads.

Scheduling overheads: Finally, we evaluate the total scheduling overheads for the various schemes. These include the overheads of offline training, classification and server selection using the greedy algorithm. Figure 8 shows the execution time breakdown for selected single-threaded and multi-threaded applications. These applications are representative of workloads submitted throughout the execution of the medium load scenario. All bars are normalized to the execution time of the application in isolation in the best SC. Training and classification for heterogeneity and interference are performed in parallel so there is a single bar for each, for every workload. There is no bar for the least-loaded scheduler for *mcf*, since it was one of the benchmarks that did not terminate successfully. Paragon achieves lower execution times for the majority of applications and close to optimal. The overheads of the recommendation system are low; 1.2% for training and 0.09% for classification. The overheads of the greedy algorithm are less than 0.1% in most cases with the exceptions of *soplex* and *genenet* that required extensive backtracking which was handled with a timeout. Overall, Paragon performs accurate classification and efficient scheduling within 1 minute of the application's arrival, which is marginal for most workloads.

5.2 Comparison of Schedulers: Large scale

Performance impact: Figure 9 shows the performance for the three workload scenarios on the 1,000-server EC2 cluster. Similar to the results on the local cluster, the *low load* scenario, in general, does not create significant performance challenges. Nevertheless, Paragon outperforms the other three schemes, it maintains QoS for 91% of workloads and achieves on average 0.96 of the performance of a workload running in isolation in the best SC. When moving to the case of *high load*, the difference between schedulers becomes more obvious. While the heterogeneity and interference-oblivious schemes degrade performance by an average of 22% and 34% and violate QoS for 96% and 97% of workload respectively, Paragon degrades performance only by 4% and *guarantees QoS* for 61% of workloads. The least-loaded scheduler degrades performance by 48% on average, while some applications do not terminate (crash). The differences in performance are larger for workloads submitted when the system is heavily loaded and becomes oversubscribed. Although, we simply queue applications in FIFO order until resources become available, Paragon bounds performance degradation (only 0.6% of workloads degrade more than 20%), since it co-schedules workloads that minimize destructive interference. We plan to incorporate a better admission control protocol in the scheduler in future work.

Finally, for the oversubscribed case, NH, NI and LL dramatically degrade performance for most workloads, while the number of applications that do not terminate successfully increases to 10.4%. Paragon, on the other hand, provides strict QoS guarantees for 52% of workloads, while the other schedulers provide similar guarantees only for 5%, 1% and 0.09% of workloads respectively. Additionally, Paragon limits degradation to less than 10% for an additional 33% of applications and maintains performance degradation moderate (no cliffs in performance such as for NH in applications [1-1000]).

Decision quality: Figure 10 shows a breakdown of the decision quality of the different schedulers for heterogeneity (left) and interference (right) across the three experiments. LL induces more than 20% performance degradation to most applications, both in terms of heterogeneity and interference. NH has low decision quality in terms of platform selection, while NI causes performance degradation by colocating unsuitable applications. The errors increase as we move to scenarios of higher load. Paragon decides optimally for 65% of applications for heterogeneity and 75% for interference on average, significantly higher than the other schedulers. It also constrains decisions that lead to larger than 20% degradation due to interference to less than 8% of workloads. The results are consistent with the findings for the small-scale experiments.

Resource allocation: Figure 11 shows why this deviation exists.

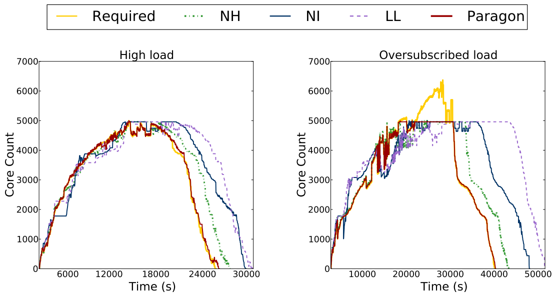


Figure 11: Comparison of required and performed core allocation between Paragon and the other three schedulers for the three workload scenarios on EC2. The total number of cores in the system is 4960.

We omit the graph for low load where deviations are small and show the high and oversubscribed scenarios. The yellow line represents the required core count based on the applications running at a snapshot of the system, while the other four lines show the allocated core count by each of the schedulers. Since Paragon optimizes for increased utilization within QoS constraints, it follows the application requirements closely. It only deviates when the required core count exceeds the resources available in the system. NH has mediocre accuracy, while NI and LL either significantly overprovision the number of allocated cores, or oversubscribe certain servers. There are two important points in these graphs: first, as the load increases the difference in execution time exceeds the optimal one, which Paragon approximates with minimal deviation. Second, for higher loads, the errors in core allocation increase dramatically for the other three schedulers, while for Paragon the average deviation remains constant, excluding the part where the system is oversubscribed.

Windows Azure & Google Compute Engine: We validate our results on a 500-server Azure and a 100-server Compute Engine (GCE) cluster. We run a scenario with 2,500 and 500 workloads respectively. Due to space reasons we omit the performance figures for these experiments, however, in both cases the results are consistent with what was noted for EC2. In Azure, Paragon achieves 94.3% of the performance in isolation and maintains QoS for 61% of workloads, while the other three schedulers provide the same guarantees for 1%, 2% and 0.7% of workloads. Additionally, this was the only time where NI outperformed NH, most likely due to the wide variation between SCs which increases the importance of accounting for heterogeneity. In the GCE cluster, which has only 4 SCs, workloads exhibit mediocre benefits from heterogeneity-aware scheduling (7% over random), while the majority of gains comes from accounting for interference. Overall, Paragon achieves 96.8% of optimal performance and NH 90%. The consistency between experiments, despite the different cluster configurations and underlying hardware, shows the robustness of the analytical methods that drive Paragon.

6. RELATED WORK

We discuss work relevant to Paragon in the areas of DC scheduling, VM management and workload rightsizing. We also present related work from scheduling for heterogeneous multi-core chips.

Datacenter scheduling: Recent work on DC scheduling has highlighted the importance of platform heterogeneity and workload interference. Mars et al. [27, 28] showed that the performance of Google workloads can vary by up to 40% due to heterogeneity even when considering only two SCs and up to 2x due to interference even when considering only two co-located applications. In [27], they present an offline scheme that used combinatorial optimization to select the proper SC for each workload. In [28], they present an offline, two-step method to characterize the sensitivity of workloads to memory pressure and the stress each application exercises to the memory subsystem. Govindan et al. [18] also present a scheme to quantify the effects of cache interference between consolidated workloads, although they require access to physical memory addresses. Finally, Nathuji et al. [32] present a control-based resource allocation scheme that mitigates the effects of cache, memory and hardware prefetching interference of co-scheduled workloads. In Paragon, we extend the concepts of heterogeneity and interference-aware DC scheduling in several ways. We provide an online, highly-accurate and low-overhead methodology that classifies applications for both heterogeneity and interference across multiple resources. We also show that our classification engine allows for efficient, online scheduling without using computationally intensive techniques which require exhaustive search between colocation candidates.

VM management: VM management systems such as vSphere [41], XenServer [47] or the VM platforms on EC2 [13] and Windows Azure [43] can schedule diverse workloads submitted by a large number of users on the available servers. In general, these platforms account for application resource requirements which they learn over time by monitoring workload execution. Paragon can complement such systems by making efficient scheduling decisions based on heterogeneity and interference and detecting when an application should be considered for migration (re-scheduling).

Resource management and rightsizing: There has been significant work on resource allocation in virtualized and non-virtualized large-scale DCs, including Mesos [20], Rightscale [35], resource containers [2], Dejavu [39] and the work by Chase et al. [10]. Mesos performs resource allocation between distributed computing frameworks like Hadoop or Spark [20]. Rightscale automatically scales out 3-tier applications to react to changes in the load in Amazon's cloud service [35]. Dejavu serves a similar goal by identifying a few workload classes and based on them, reuses previous resource allocations to minimize reallocation overheads [39]. Zhu et al. [48] present a resource management scheme for virtualized DCs that preserves SLAs and Gmach et al. [16] a resource allocation scheme for DC applications that relies on the ability to predict their behavior a priori. In general, Paragon is complementary to resource allocation and rightsizing systems. Once such a system determines the amount of resources needed by an application (e.g., number of servers, memory capacity, etc.), Paragon can classify and schedule it on the proper hardware platform in a way that minimizes interference. Currently, Paragon focuses on online scheduling of previously unknown workloads. We will consider how to integrate Paragon with a rightsizing system for scheduling long running, 3-tier services in future work.

Scheduling for heterogeneous multi-core chips: Finally, scheduling in heterogeneous CMPs shares some concepts and challenges with scheduling in heterogeneous DCs, therefore some of the ideas in Paragon can be applied in heterogeneous CMP scheduling as well. Fedorova et al. [14] discuss OS level scheduling for heterogeneous multi-cores as having the following three objectives: optimal performance, core assignment balance and response time fairness. Shelepov et al. [37] present a scheduler that exhibits some of these features and is simple and scalable, while Craeynest et al. [11] use performance statistics to estimate which workload-to-core mapping is likely to provide the best performance. DC scheduling also has similar requirements as applications should observe their QoS, resource allocation should follow application requirements closely and fairness between co-scheduled workloads should be preserved. Given the increasing number of cores per chip and co-scheduled tasks, techniques such as those used for the classification engine of Paragon can be applicable when deciding how to schedule applications to heterogeneous cores as well.

7. CONCLUSIONS

We have presented Paragon, a scalable scheduler for DCs that is both heterogeneity and interference-aware. Paragon is derived from validated analytical methods, such as collaborative filtering to quickly and accurately classify incoming applications with respect to platform heterogeneity and workload interference. Classification uses minimal information about the new application and relies mostly on information from previously scheduled workloads. The output of classification is used by a greedy scheduler to assign workloads to servers in a manner that maximizes application performance and optimizes resource usage. We have evaluated Paragon with both small and large-scale systems. Even for very demanding scenarios, where heterogeneity and interference-agnostic schedulers degrade performance for up to 99.9% of workloads, Paragon maintains QoS guarantees for 52% of the applications and bounds degradation to less than 10% for an additional 33% out of 8500 applications on a 1,000-server cluster. Paragon preserves QoS guarantees while improving server utilization, hence it benefits both the DC operator, who achieves perfect resource use and the user, who gets the best performance. In future work we will consider how to couple Paragon with VM management and rightsizing systems for large-scale datacenters.

ACKNOWLEDGEMENTS

We sincerely thank our shepherd, Byung-Gon Chun for his useful comments and feedback. We also thank John Ousterhout, Mendel Rosenblum, Daniel Sanchez, David Lo, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was partially supported by a Google directed research grant on energy proportional computing. Christina Delimitrou was supported by a Stanford Graduate Fellowship.

8. REFERENCES

[1] A. Alameldeen, D. Wood. “IPC Considered Harmful for Multiprocessor Workloads”. *In IEEE Micro, July/Aug. 2006.*

[2] G. Banga, P. Druschel, J. Mogul. “Resource containers: a new facility for resource management in server systems”. *In Proc. of OSDI, 1999.*

[3] L. Barroso. "Warehouse-Scale Computing: Entering the Teenage Decade". *ISCA Keynote, SJ, June 2011.*

[4] L. A. Barroso, U. Holzle.“The Datacenter as a Computer". *Synthesis Series on Computer Architecture, May 2009.*

[5] L. A. Barroso and U. Holzle.“The Case for Energy- Proportional Computing". *Computer, 40(12):33–37, 2007.*

[6] R. M. Bell. Y. Koren, C. Volinsky.“The BellKor 2008 Solution to the Netflix Prize". Technical report, AT&T Labs, Oct 2007.

[7] L. Bottou. “Large-Scale Machine Learning with Stochastic Gradient Descent”. *In Proc. of COMPSTAT 2010.*

[8] C. Bienia, et al. “The PARSEC benchmark suite: Characterization and architectural implications”. *In Proc. of PACT, 2008.*

[9] B. Calder, et al. “Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency”. *In Proc. of SOSP, 2011.*

[10] J. Chase, D. Anderson, et al. “Managing Energy and Server Resources in Hosting Centers". *In SIGOPS, 35(5):103–116, 2001.*

[11] K. Craeynest, et al. “Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)”. *In Proc. of ISCA, 2012.*

[12] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. *In Proc. of OSDI, SF, 2004.*

[13] Amazon Elastic Compute Cloud.`http://aws.amazon.com/ec2/`

[14] A. Fedorova, D. Vengerov, D. Doucette. “Operating System on Heterogeneous Core Systems”. *In Proc. of OSHMA, 2007.*

[15] S. Ghemawat, H. Gobioff, S.-T Leung . “The Google File System”. *In Proc. of SOSP, NY, 2003.*

[16] D. Gmach, J. Rolia, et al. “Workload Analysis and Demand Prediction of Enterprise Data Center Applications”. *In Proc. of IISWC, 2007.*

[17] Google Compute Engine. `cloud.google.com/compute`

[18] S. Govindan, et al. “Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. *In Proc. of SOCC, 2011.*

[19] J.R. Hamilton. “Cost of Power in Large-Scale Data Centers”. `http://perspectives.mvdirona.com`

[20] B. Hindman, et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. *In Proc. of NSDI, 2011.*

[21] A. Jaleel, M. Mattina, B. Jacob. “Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP - A Case Study of Parallel Bioinformatics Workloads”. *In Proc. of 12^{th} HPCA, TX, 2006.*

[22] J. Katz and Y. Lindell. “Introduction to Modern Cryptography”. *Chapman & Hall/CRC Press, 2007.*

[23] C. Kozyrakis, A. Kansal, et al. “Server Engineering Insights for Large-Scale Online Services”. *In IEEE Micro, vol.30, no.4, July 2010.*

[24] K. Kiwiel. “Convergence and efficiency of subgradient methods for quasiconvex minimization”. *Math. Programming, Springer, 2001.*

[25] J. Leverich, C. Kozyrakis. “On the Energy (In)Efficiency of Hadoop Clusters”. *In Proc. of HotPower, October 2009.*

[26] J. Lin, A. Kolcz. “Large-Scale Machine Learning at Twitter”. *In Proc. of SIGMOD, Scottsdale, 2012.*

[27] J. Mars, L. Tang, R. Hundt. “Heterogeneity in ”Homogeneous” Warehouse-Scale Computers: A Performance Opportunity”. In IEEE CAL, July-December 2011.

[28] J. Mars, L. Tang, et al. “Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations”. *In Proc. of MICRO-44, Brazil, December 2011*

[29] D. Meisner, C. Sadler, L. A. Barroso, et al. “Power Management of On-line Data-Intensive Services”. *In Proc. of ISCA, SJ, June 2011.*

[30] R. Narayanan, B. Ozisikyilmaz, et al. “MineBench: A Bench- mark Suite for DataMining Workloads”. *In Proc. of IISWC, 2006.*

[31] R. Nathuji, C. Isci, E. Gorbatov. “Exploiting platform heterogeneity for power efficient data centers”. *In Proc. of ICAC, 2007.*

[32] R. Nathuji, et al. “Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds”. *In Proc. of EuroSys, 2010.*

[33] Rackspace. `http://www.rackspace.com/`

[34] A. Rajaraman and J. Ullman. “Textbook on Mining of Massive Datasets”, 2011.

[35] Amazon EC2: Rightscale. `https://aws.amazon.com/solution-providers/isv/rightscale`

[36] D. Sanchez, C. Kozyrakis. “Vantage: Scalable and Efficient Fine-Grain Cache Partitioning”. *In Proc. of ISCA, SJ, 2011.*

[37] D. Shelepov, J. Saez, et al. “HASS: A Scheduler for Heterogeneous Multicore Systems”. *In OSP, vol. 43, 2009.*

[38] J. Sun, Y. Xie, H. Zhang, C. Faloutsos. “Less is More: Compact Matrix Decomposition for Large Sparse Graphs”. *In Proc. of SDM, 2007.*

[39] N. Vasic, et al. “DejaVu: Accelerating Resource Allocation in Virtualized Environments”. *In Proc. of ASPLOS, London, 2012.*

[40] vMotion™. `http://www.vmware.com/products/vmotion`

[41] VMWare vSphere. `http://www.vmware.com/products/vsphere/`

[42] T. Wenisch, et al. “SimFlex: Statistical Sampling of Computer System Simulation”. *In IEEE MICRO, vol. 26, no. 4, Jul-Aug 2006.*

[43] Windows Azure. `http://www.windowsazure.com/`

[44] I. Witten, E. Frank et al. “Data Mining: Practical Machine Learning Tools and Techniques”. M. Kaufmann, 3rd Edition.

[45] S. Woo, M. Ohara, et al. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. *In Proc. of the 22nd ISCA, 1995.*

[46] Xen Hypervisor 4.0. `http://www.xen.org/`

[47] XenServer. `http://www.citrix.com/products/xenserver/overview.html`

[48] X. Zhu, et al. “1000 Islands: An Integrated Approach to Resource Management for Vurtualized Datacenters”. *In Cluster Computing Journal, 2009.*