QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon

CHRISTINA DELIMITROU and CHRISTOS KOZYRAKIS, Stanford University

Large-scale datacenters (DCs) host tens of thousands of diverse applications each day. However, interference between colocated workloads and the difficulty of matching 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 a 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. It does so 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. After the initial application placement, Paragon monitors application behavior and adjusts the scheduling decisions at runtime to avoid performance degradations. Additionally, we design ARQ, a multiclass admission control protocol that constrains application waiting time. ARQ queues applications in separate classes based on the type of resources they need and avoids long queueing delays for easy-to-satisfy workloads in highly-loaded scenarios. Paragon scales to tens of thousands of servers and applications 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**]: Large and Medium ("Mainframe") Computers—*Super (very large) computers*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms: Design, Performance

Additional Key Words and Phrases: Datacenter, cloud computing, resource-efficiency, heterogeneity, interference, scheduling, QoS

ACM Reference Format:
Delimitrou, C. and Kozyrakis, C. 2013. QoS-aware scheduling in heterogeneous datacenters with Paragon. ACM Trans. Comput. Syst. 31, 4, Article 12 (December 2013), 34 pages.
DOI: http://dx.doi.org/10.1145/2556583

This work was partially supported by a Google directed research grant on energy proportional computing. Christina Delimitrou was supported by a Stanford Graduate Fellowship.
Authors' addresses: C. Delimitrou and C. Kozyrakis, Electrical Engineering Department, Stanford University; email address: cdel@stanford.edu.

DOI: http://dx.doi.org/10.1145/2556583

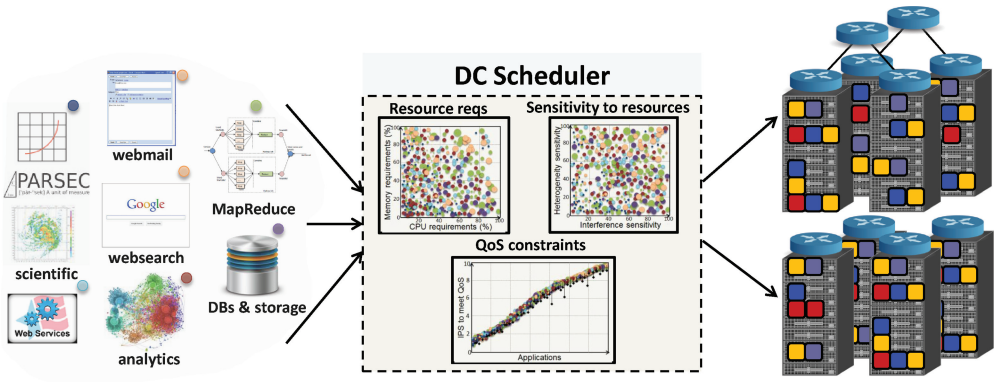


Fig. 1. Overview of scheduling in a datacenter. Applications vary in their resource requirements, their sensitivity to server configuration and interference, and their performance constraints. The scheduler needs to account for all these factors to assign applications to DC servers.

1. INTRODUCTION

An increasing amount of computing is performed in the cloud, primarily due to the flexibility and cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [Barroso and Hoelzle 2009]. Large-scale providers such as Amazon EC2 [Amazon EC2], Microsoft Windows Azure [Windows Azure], Rackspace [Rackspace] and Google Compute Engine [GCE] 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 [VMWare vSphere] or Citrix XenServer [Xenserver].

The operator of a cloud service must schedule the stream of incoming applications on the 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. Figure 1 shows an overview of how scheduling happens in a datacenter. Applications of different types are submitted to the system and the scheduler must decide how to efficiently assign them to servers. This scheduling problem is particularly difficult, as cloud services must accommodate a diverse set of workloads in terms of their resource and performance requirements and the sensitivity of different applications to server configurations and interference from coscheduled workloads [Barroso and Hoelzle 2009; Kozyrakis et al. 2010]. As shown in the figure, even when looking only at memory and CPU requirements, application demands vary widely. The same is the case for their sensitivity to the configuration of hardware platforms and the contention in shared resources, as well as their QoS constraints. Moreover, the operator often has no *a priori* knowledge of workload characteristics. In this work, we focus on two main challenges that complicate scheduling in large-scale DCs: *hardware platform heterogeneity* and *coscheduled workload interference*.

Heterogeneity occurs as servers are gradually provisioned and replaced over the typical 15-year lifetime of a datacenter infrastructure [Barroso and Hoelzle 2009; Hamilton 2010, 2009; Kozyrakis et al. 2010; Mars and Tang 2013; Nathuji et al. 2007]. At any point in time, a DC may host 3 to 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 2(a) shows that a heterogeneity-oblivious scheduler will slow applications down by 22%

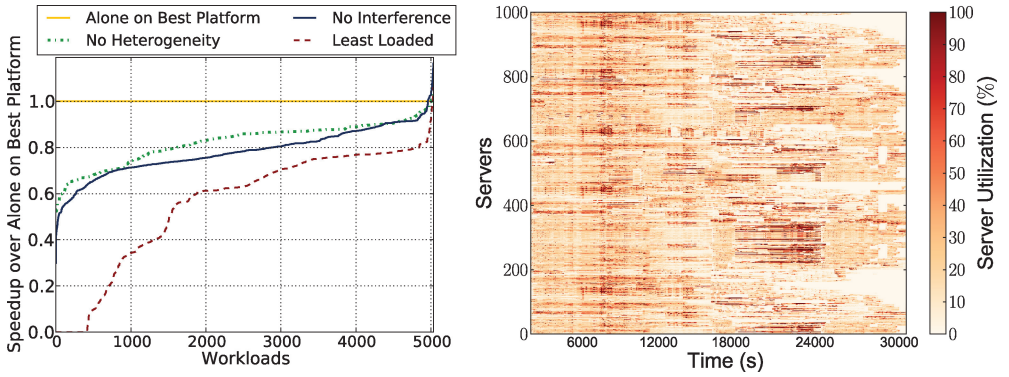


Fig. 2. Performance degradation for 5,000 applications on 1000 EC2 servers with heterogeneity-oblivious, interference-oblivious and baseline least-loaded schedulers compared to ideal scheduling (application runs alone on best platform) (Figure 2(a)). Results are ordered from worst to best-performing workload. Utilization for the 1000-server cluster when using the least-loaded scheduler (Figure 2(b)). Utilization is defined as CPU utilization across the cores of each server.

on average, with some running nearly 2× slower. In this experiment 5000 randomly-selected applications are submitted to a 1000-server EC2 cluster (see Section 5 for details on the 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 coscheduling multiple workloads on a single server to increase utilization and achieve better cost efficiency. By colocating 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 [Barroso 2011; Barroso and Hoelzle 2009; Leverich and Kozyrakis 2010; Meisner et al. 2011]. Coscheduled applications may interfere negatively even if they run on different processor cores because they share caches, memory channels, storage and networking devices [Govindan et al. 2011; Mars et al. 2011; Nathuji et al. 2010]. Figure 2(a) shows that an interference-oblivious scheduler will slow workloads down by 34% on average, with some running more than 2× slower. Again, this is undesirable for both users and operators. Finally, a baseline 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. Figure 2(b) shows the cluster utilization achieved by the least-loaded baseline scheduler. Utilization is calculated as average CPU utilization across a server's cores. For the majority of servers, utilization rarely exceeds 20 to 30%, and in the few cases where utilization is high, performance suffers from either contention or inappropriate server configuration.

Previous work has showcased the potential of heterogeneity and interference-aware scheduling [Mars and Tang 2013, 2011; Delimitrou and Kozyrakis 2013c; Govindan et al. 2011; Nathuji et al. 2007]. Mars et al. [2013, 2011] profile the sensitivity of applications to hardware platforms and to contention to memory resources, and propose techniques that significantly improve utilization with minimal performance degradation. However, techniques that rely on detailed application characterization cannot scale to large DCs that receive tens of thousands of potentially unknown workloads every day [Shen et al. 2011; Barroso 2011; Calder et al. 2011]. Most cloud management systems have some notion of contention or interference-awareness [Hindman

et al. 2011; Nathuji et al. 2010; Vasić et al. 2012; vMotion; Xenserver]. Nonetheless, these schemes 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 of the application. Instead, we leverage information the system *already* has about the large number of applications it has previously seen.

We present *Paragon* [Delimitrou and Kozyrakis 2013b], 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 coscheduled 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 in the Netflix Challenge [Bell et al. 2007], 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 coscheduled 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. The system currently focuses on cloud provider scenarios where independent, unknown applications are submitted to the DC. Similar techniques can also be applied for scheduling in systems with long-running applications, such as the clusters running complex multitier services, like Search or Webmail. We defer the study of these scenarios to future work.

Application behavior is rarely constant throughout their execution. Most workloads go through multiple phases that affect their resource requirements and their sensitivity to heterogeneity and interference. Paragon accounts for changes in application behavior and adjusts the scheduling decisions to accommodate them. After the initial application placement, the scheduler monitors application performance and detects possible phase changes. Paragon supports two phase-detection modes: reactive, where rescheduling is evaluated as a result of performance degradation due to a phase change, and preemptive, where rescheduling is evaluated before phases reflect in performance degradation. Both modes rely on lightweight reclassification that detects deviations from the original workload profile.

In such cloud providers, a large fraction of workloads are very short, running batch applications such as short-term analytics. While the scheduling overheads are generally low, they can become noticeable for applications with completion times on the order of a few seconds. For such workloads, Paragon includes an optimization that bypasses the training step of classification and approaches these short jobs as completely unknown input loads (cold-start problem). This significantly reduces the scheduling overheads while still providing decisions of reasonable quality.

Finally, apart from fast execution time, DC users are interested in how fast an application gets scheduled (low waiting time). In the case of oversubscribed scenarios,

applications with few resource requirements can get trapped behind demanding workloads. To avoid head-of-line-blocking with long queueing delays, we design ARQ, a multiclass admission control protocol, that identifies classes of applications and queues workloads based on the type of resources they need. This way nondemanding workloads can get scheduled quickly, while the system still preserves QoS for all submitted applications, by bounding queueing time and diverging workloads to free queues.

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 article is organized as follows. Section 2 describes the analytical methods that drive Paragon. Section 3 presents the scheduler's implementation and Paragon's phase-detection schemes, and Section 4 discusses the multiclass queueing network. Section 5 presents the experimental methodology and Section 6 the evaluation of Paragon. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2. FAST AND 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 [Mars et al. 2011, 2013; Nathuji et al. 2007, 2010; Vasić et al. 2012]. Instead, Paragon takes a different perspective. Its core idea is that, instead of learning each new workload in detail, the system can leverage information it already has about applications it has previously 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 workloads. 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 [Bell et al. 2007], to provide a quick overview of the two analytical methods we rely upon, Singular Value Decomposition (SVD) and PQ-reconstruction (PQ) [Rajaraman and Ullman 2011; Bottou 2010; Kiwiel 2001; Witten et al. 2011]. 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, who may only have a handful of ratings or even none. While there are techniques that address the cold start problem, that is, 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

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

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

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

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 accuracy of the eventual recommendations [Sun et al. 2008].

However, 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 [Bottou 2010; Kiwiel 2001; Lin and Kolcz 2012; Witten et al. 2011] 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 represent 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 Section 2.1 to derive the missing ratings for the other server configurations.

In this case, Σ represents similarity concepts such as the fact that applications that benefit from SC1 will also benefit from SC3, or that applications that benefit from platforms with high memory bandwidth do not benefit from platforms with high storage capacity. U captures how an application correlates to the different similarity concepts and V shows 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,

recommendation 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 30 offline workloads for a 40 and 1000-server cluster, respectively. Additionally, as more incoming applications are added in A, the density of the matrix increases and the recommendation accuracy improves further. 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 to 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 potentially be more accurate—however, it 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 multiprogrammed 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 [Alameldeen and Wood 2006; Wenisch et al. 2006]. 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 has influenced our current evaluation which focuses on single-node CPU, memory and I/O-intensive programs. The same methodology holds with small changes for higher-level metrics, such as queries per second (QPS), which cover complex multitier workloads as well.

Validation. We evaluate the accuracy of the heterogeneity classification on a 40-server cluster with 10 SCs. We use a large set of single-threaded, multithreaded, multi-programmed and I/O-bound workloads. For details on workloads and server configurations, see Section 5. The offline training set includes 20 applications selected randomly from all workload types. Scheduling applications to hardware platforms based on the recommendation system improves performance by 24% for single-threaded, 20% for multithreaded, 38% for multiprogrammed, and 40% for I/O workloads on average, while some applications have a 2× performance difference. Table I summarizes key statistics on 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

Table I. Validation Metrics for Heterogeneity Classification

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%

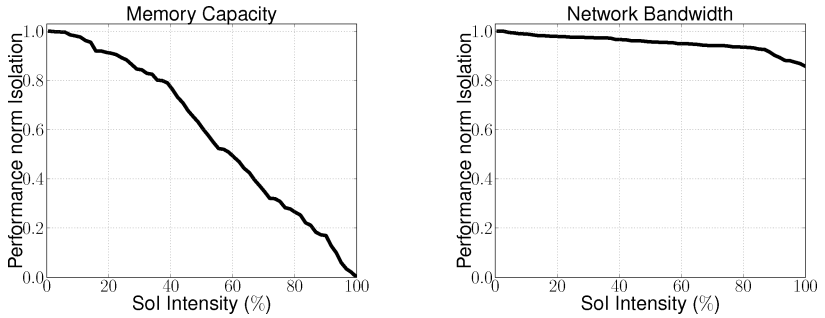


Fig. 3. Sensitivity curves for memory capacity (Figure 3(a)) and network bandwidth (Figure 3(b)) for mcf. As the intensity of the memory capacity SoI increases, mcf's performance degrades. On the other hand, mcf is insensitive to network contention, so its tolerance is close to 100%.

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 for only 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. The intensity of the microbenchmark at that point is recorded as the *sensitivity score* of the workload to the specific type of interference. Figure 3 shows an example of deriving the sensitivity to tolerated interference in the memory capacity and network bandwidth for the *mcf* benchmark of the SPECCPU2006 suite. mcf is very sensitive to memory capacity, with performance quickly degrading as the intensity of the SoI increases. On the other hand, given that it is a single-node workload with no network-related component, it is very tolerant to contention in the network bandwidth. Applications with high tolerance to interference (e.g., sensitivity score over 60%) are easier to coschedule 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

Table II. Overview of Different Contentious Microbenchmarks and Their Corresponding Access Patterns

Microbenchmark (SoI)	Access Pattern
Memory capacity	Random using SSA for ILP
Memory bandwidth	Streaming at increasing rate
LLC capacity (similar for L2)	Random with increasing footprint
LLC bandwidth (similar for L2)	Streaming at increasing rate
L1 i-cache	Random with increasing footprint
L1 d-cache	Random with increasing footprint
Translation Lookahead Buffer (TLB)	Page fetching at increasing rate
Core	Integer, floating point (and vector if apl) instructions
Network bandwidth	Streaming at increasing rate & request size
Storage bandwidth	Streaming at increasing rate & request size

to its performance in isolation. In this case, high sensitivity scores, for example, over 60% correspond to applications that cause a lot of interference in the specific shared resource.

Identifying Sources of Interference (SoI). Coscheduled 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), core and network and storage bandwidth. The same methodology can be expanded to any shared resource. Table II provides an overview of the type and access pattern for each of the ten microbenchmarks. In all cases the intensity of the kernel ranges from minimal to the maximum achieved for a given platform and increases "almost" proportionately with time. The contentious microbenchmarks are designed such that they put most of the pressure in one shared resource at a time. This is easier for resources such as memory bandwidth, where a streaming access pattern can stress bandwidth but not necessarily capacity—but it becomes harder when stressing, for example, the L1 i-cache without saturating the core. We have validated that the effect of the different microbenchmarks does not overlap, or that any overlap is kept minimal. More details on this validation and the design of each of the contentious kernels can be found in Delimitrou and Kozyrakis [2013a].

Collaborative Filtering for Interference. We classify applications for interference, tolerated and caused, using the process described in Section 2.1 twice. 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 Section 3.2).

Validation. We evaluated the accuracy of interference classification using the single-threaded and multithreaded workloads and the same systems as for the heterogeneity classification. Table III summarizes some key statistics on the classification quality. Our classifier achieves an average error of 5.3% in estimating both tolerated and

Table III. Validation Metrics for Interference Classification

Metric	Percentage (%)
Average sensitivity error across all SoIs	5.3%
90^{th} percentile error across all SoIs	10.5%
99^{th} percentile error across all SoIs	18.6%
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%

caused interference across all SoIs. For high values of sensitivity, that is, applications that tolerate and cause a lot of interference, the error is even lower (3.4%), while for most applications (both single-threaded and multithreaded) 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 multithreaded 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 [Mars et al. 2011]. 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 the 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, that is, 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 milliseconds, 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.

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

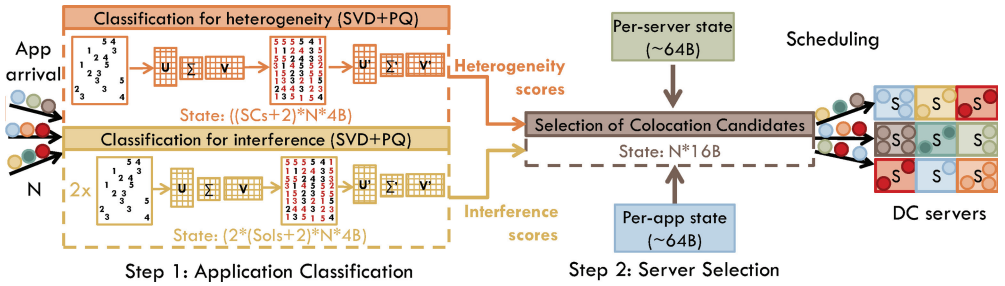


Fig. 4. The components of Paragon and the state maintained by each component. Overall, the state requirements are marginal and scale linearly or logarithmically with the number of applications (N), servers (M), server configurations (SC), and sources of interference (SoI).

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 those 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).

Figure 4 presents an overview of Paragon and its components. The scheduler maintains a per-application and per-server state. The 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 with constant overheads that provides strong guarantees on the quality of candidates as a function of examined servers.

[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.

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 $min(||D_1 + D_2||_{L1})$.

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 nonempty, 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, for example, 10 to 100k servers, the overhead from

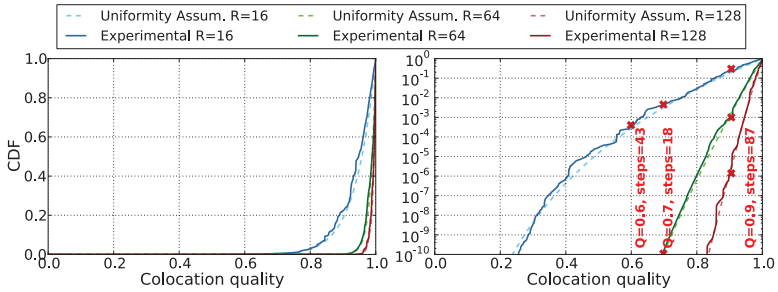


Fig. 5. Colocation quality distribution ($F(x) = x^R$, where $R = 16, 64$ and 128). Figure 5(b) shows the comparison between the greedy algorithm and the statistical scheme for three colocation candidates of $Q = 0.6, 0.7$, and 0.9.

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 [Katz and Lindell 2007] 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 (14 bits). 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. Rehashing b obtains additional IDs of candidate servers. This produces a random subset of the system's servers. After a number of rehashes 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

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

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 $\frac{(-1.0+|min|)}{|max|+|min|} = 0.2/2 = 0.1$.

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 5(a) 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 1000 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, for example, a 5000-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 5(b)). 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 1000-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. Reclassification: Adjusting to Workload Phases

The initial classification in Paragon is performed upon workload arrival, using the information from its 1-minute profiling. Most workloads, however, go through multiple phases throughout their execution. These phases are not captured in the initial profiling and can reflect on changes in resource demands which result in poor performance. Paragon has mechanisms to detect and react to workload phases in order to guarantee QoS throughout a workload's execution.

Section 6 includes a workload scenario where application behavior experiences different phases. The phase-detection mechanisms in Paragon are activated in this case, and take action to address the changes in application behavior.

Workload phases can be detected either upon QoS degradation for a given application (*reactively*), or *preemptively*, before the phase change impacts performance. Paragon supports both detection modes. In reactive mode, the performance of each active application in the cluster is monitored periodically. When performance degradation is detected, either the application has entered a new execution phase, or the classification engine classified it incorrectly. In both cases, the scheduler must take action to avoid further performance degradation. This involves reclassification and potential rescheduling.

Live (in vivo) Classification. Reclassification can happen online on the server where the offending application is running. It follows the same steps as the original classification process. Two microbenchmarks (SoIs) are injected in the system and measure the new interference the application causes and tolerates. Then, the recommender recovers the missing entries and provides the greedy scheduler with the new interference profile. If the scheduler finds an available server that matches the application's requirements better, and the performance benefit amortizes the migration cost, the

workload is migrated to that server.[3] Alternatively, if no suitable server is available, any coscheduled applications are moved to another machine, such that the degraded application runs in isolation. The advantages of *in vivo* reclassification is that migration is only performed when necessary. On the other hand, classification on a server that hosts multiple workloads might be prone to errors due to coscheduled workloads that share resources, such as the memory or cache hierarchy. To minimize classification inaccuracies, the training SoIs in this case correspond to resources that are *only* shared between the offending application and the microbenchmark, for example, L1 caches. This still introduces some minor second-order effects, but we have found these to be negligible in the majority of cases.

Isolated (in vitro) Classification. Alternatively, reclassification can happen in an isolated environment. Upon detection of QoS violation, the workload is migrated to an empty server and reclassified following the original process. The advantage in this case is that classification is not affected by coscheduled applications, and workload performance does not continue to degrade during classification, by remaining in a suboptimal server. The disadvantage is that migration, especially for a stateful workload, may incur significant overheads. Paragon decides between the two types of classification based on the type and state maintained by a given application.

The second phase detection mode is *pre-emptive*. The scheduler periodically samples a few applications, injects microbenchmarks in the corresponding servers and performs live reclassification. If it detects significant deviations from the previous interference profile of a workload, it considers whether migration or rescheduling is beneficial. Preemptive detection prevents QoS violations in the cases where phase changes are not instantaneous.

Reclassification could theoretically hurt application performance, especially when a workload is already under stress. To ensure that no additional degradation occurs due to reclassification, the performance of the reclassified workload is constantly monitored as the intensity of the microbenchmark increases and the latter is immediately removed once the performance of the workload starts to degrade (and before it violates its QoS). This trades-off some accuracy in sensitivity estimation to avoid degradations from an overly aggressive microbenchmark.

In future work, we will consider how resource partitioning techniques can improve live classification accuracy by eliminating interference between coscheduled workloads.

Both in the reactive and pre-emptive modes, workload migration following the detection of a change in behavior can be avoided if the performance degradation is due to a transient spike in load. In that case the application will briefly suffer until the load surge subsides, but can remain in its current assignment, avoiding data movement. This trade-off becomes more complicated for latency-critical applications where even transient drops in performance can violate tail latency guarantees. Currently, Paragon decides only between migration and execution in isolation, based on the state maintained by a given workload.

Table IV shows a short validation of the accuracy of phase detection across the full SPECCPU2006 suite. Applications are scheduled using Paragon on a 6-machine cluster (8 cores, 24GB RAM per machine) that supports VM migration. "Applications with phases" correspond to workloads that experienced degraded performance (lower IPS than when running in isolation) at some point during their execution. In reactive

[3]The exact mechanics of migration depend on the underlying system, for example, process or VM migration. In all experiments there is an underlying mechanism, such as vSphere [VMWare vSphere] that performs the live migration.

Table IV. Validation of the Phase Detection Mechanism

Reactive detection	
Applications with phases	11
Phases detected	96%
Time until detection	8.5sec
Applications migrated	8
Performance degradation	hmean:3.4% 90^{th} pctl:5.6%
Preemptive detection	
Applications sampled	6 (20%)
Sampled applications with phases	4
Phases detected	3
Applications migrated	2
Performance degradation	hmean:1.8% 90^{th} pctl:3.5%

mode, Paragon detects the majority of phases and adapts application placement quickly to constrain performance losses. In pre-emptive mode, Paragon samples 20% of workloads at a time, and detects upcoming phase changes in 75% of cases where an actual change happened. It also constrains false positives to 5.1%, and therefore avoids unnecessary migrations. The table shows a snapshot from one application sampling.

3.5. Short-Running Jobs

A large fraction of the jobs submitted to public and private clouds are short, batch jobs that complete within a few seconds from their initiation. Introducing a 1-minute training overhead in this case would cause a significant performance penalty for these workloads. Instead, we can skip the training phase and proceed directly to the scheduling decisions. The disadvantage here is that the system now suffers from the "cold-start" problem, which means that no ratings exist for a new "user" (application). This is similar to the event where a new, unknown server configuration is added in the DC. Due to the rarity of that event, the solution in that case is exhaustive profiling of a few applications against the new platform. This solution would not be feasible for the large numbers of such short jobs. Collaborative filtering-based recommender systems often resolve the cold start problem by adopting a hybrid content-based and collaborative filtering approach [Schein et al. 2002], where new users (or items) are assigned ratings automatically, such that the recommender can start finding similarities between them and known users. These automatic ratings can be determined probabilistically or based on popularity, age, or overall quality of the rated item. For example, in the context of a movie recommendation system, entirely unknown users will be recommended some of the most popular, newest, or most highly rated movies, until they provide some ratings of their own as a signal to the collaborative filtering system [Zhang et al. 2010; Weng et al. 2008].

In Paragon, these jobs are scheduled on a randomly-selected server from the machines that have the most available resources and the least caused interference. Note that the information on caused interference is already available from the training phase of the workloads scheduled on those machines. This way the total scheduling overheads of short-running jobs are a few milliseconds, while the jobs are still assigned to appropriate servers. The difference in post-training execution time between selecting a machine using the training runs as input, and with the random selection described above is on average 15% for a set of 100 Hadoop analytics jobs with ideal completion time of 10–30sec. While the difference is negligible for the specific tasks, it becomes considerable for more complex, long-running applications, justifying the need for the training runs in Paragon. Also note that sending the short jobs to the servers with

the least interference is not necessarily the most resource-efficient decision, but it is adopted to preserve the per-application QoS guarantees.

3.6. Discussion

Resource Partitioning and Performance Isolation. Paragon does not explicitly partition resources, therefore it does not enforce strict performance isolation between coscheduled workloads. Instead, it reduces interference by coscheduling applications that do not contend on the same shared resources. Resource partitioning is orthogonal to Paragon. Once interference is reduced, hardware or software isolation techniques can be deployed to eliminate any existing contention. Our EC2 experiments already benefit from existing isolation schemes, for example, the Xen hypervisor deployed in the cluster. We will consider how partitioning interacts with the cluster manager in future work.

Suboptimal Scheduling. A second concern apart from performance isolation 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 rescheduling several active applications. Although rescheduling was not needed for the examined applications, Paragon provides a general methodology to detect such deviations and leverage mechanisms like VM migration to reschedule the suboptimally scheduled workloads.

Latency-Critical Applications and Workload Dependencies. Finally, Paragon does not explicitly consider latency-critical applications or dependencies between application components, for example, a multitier 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 needs to 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, for example, 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, for example, MapReduce runs with high priority and the remaining server capacity is used by instances of a latency-critical or batch 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 [Hindman et al. 2011; Ghodsi et al. 2011], and we will consider how it interacts with Paragon in future work.

4. APPLICATION-AWARE ADMISSION CONTROL

Large cloud providers, such as Amazon EC2 and Windows Azure, typically deploy some admission control protocol. This prevents machine oversubscription, that is, the same core servicing more than one applications, which can induce interference, resulting in serious performance losses.

4.1. Resource Quality-Aware Queueing

We design *Admission Control with Resource Quality Awareness* (ARQ), a QoS-aware admission control protocol that queues and schedules applications based on the quality of resources they need. This solves two problems: First, applications that demand few, easy-to-satisfy resources are not blocked behind demanding workloads. Second, in the case where no suitable servers are available for a given application, the system

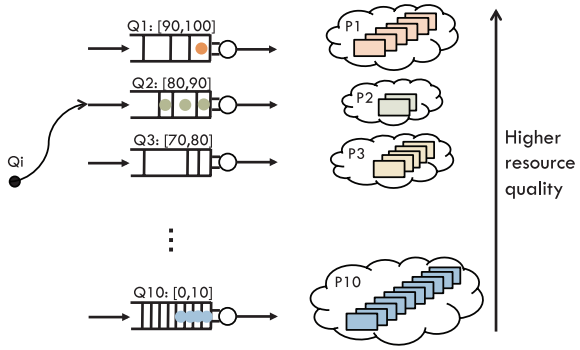


Fig. 6. ARQ design. Each queue corresponds to applications with different resource quality requirements. Servers are also partitioned in pools based on the quality of their resources.

waits for a server of appropriate quality to be freed before it schedules that workload. Alternatively, the application would be directed to the first free server to avoid queueing delays, with the risk of performance losses. In the discussion of ARQ's design we assume single-node applications for simplicity. With some changes the design can hold for multinode workloads as well.

Resource Quality. The resource demands of a workload reflect the load a server should support such that the application can meet its QoS constraints. This is a function of the interference the server can tolerate from the new application, and the interference the new workload can tolerate from applications already running in the same machine. These values are obviously affected by server configuration as well. We use the classification engine in Paragon to derive the sensitivity to tolerated (t_i) and caused (c_i) interference of each server in the cluster. The interference profile for a server is updated upon initiation or completion of an application's execution. Similarly, upon application arrival, Paragon has obtained the interference profile of the incoming workload, as described in Section 2.3. This information guides the scheduling decisions by assigning applications to appropriate servers. Given the interference profile of a server or application, we define *resource quality* as:

$$Q_i = \sum_i c_i, \tag{1}$$

where c_i is summed over all shared resources for which interference is accounted for. Conceptually, higher Q_i reflects applications with high demands (high caused and low tolerated interference) that need high-quality system resources. Low Q_i on the other hand, corresponds to workloads that are insensitive to interference, and can satisfy their QoS even when assigned to servers with poor resource quality, for example, highly-loaded machines, or machines with few cores.

Multiclass Admission Control. We design ARQ as an admission control protocol with multiple classes of "customers" [Bertsimas et al. 2001; Miller 1969], where customers in this case correspond to applications. The class an application belongs to is determined by its Q_i value. Applications with Q_i values that fall in the same range are assigned to the same class. Q_is range from 0 to 100%. We assume ten classes of applications for now. We have performed a sensitivity study with different numbers of queues that justifies this selection. Figure 6 shows an overview of ARQ. Each queue corresponds to applications of a specific class. From top to bottom we move from more to less demanding applications. Upon workload arrival, Paragon determines the class it belongs to based on its interference profile and queues it appropriately. Each class has

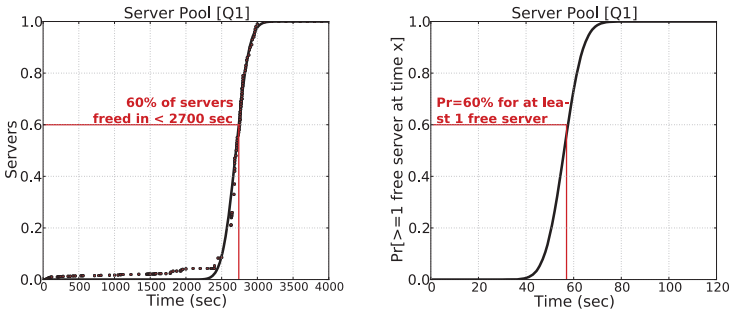


Fig. 7. CDF of server busy times (Figure 7(a)) and CDF of the probability that there will be at least one free server within a specific time window from an application's arrival (Figure 7(b)).

a respective pool of servers of the corresponding resource quality that are most suitable to service the queue's applications. By separating applications based on their resource quality requirements, ARQ avoids bottlenecks where applications that are sensitive to interference block workloads that are not. However, limiting the resources an application can access to the subset of servers of the corresponding pool can also result in performance violations. DC workloads are driven by strict QoS and SLA guarantees and cannot be queued indefinitely waiting for a suitable server to be freed. We address this issue by diverging workloads to queues with better or worse resource qualities.

4.2. Waiting Time vs. Quality of Resources

Diverging an application to a different queue creates a trade-off between the time an application is waiting in a queue and the quality of resources it is allocated. The more time it waits, the higher the chance it will be scheduled to a preferred server. At the same time, as waiting time increases, there is a higher chance the application will violate its QoS requirements. We approach this trade-off as an optimization problem.

Queue Bypassing. When there is no available machine in the server pool of a specific class, queued workloads should be diverged to other queues. There are two possible options for where a workload can be redirected. First, it can be diverged to a higher queue. If the queue directly above the queue the workload was originally placed in is empty, then the workload is assigned to one of that queue's available servers. This hurts utilization, since resources of higher quality than necessary are allocated, but preserves the workload's QoS requirements. In the opposite case, the workload is diverged to a queue of lower resource quality. In this case, performance may degrade, since the application receives resources of lower quality than required. Nonetheless, ARQ guarantees that the application will be assigned to the best available server within the time window dictated by its QoS constraints. Partitioning servers in pools also simplifies the role of the greedy scheduler, which now only has to traverse the corresponding subset of servers, unless the application gets diverged to a different queue.

Free-Server Probability Distributions. ARQ needs to know the likelihood that a server of a specific class will become available within the time an application can be queued for, to decide whether/when it should be diverged to the next queue. We statistically analyze the per-server busy-time periods for each pool to obtain these probability distributions. Busy periods are defined as the per-server intervals from the moment a server is assigned a workload, until that workload completes.

We first use *distribution-fitting* to represent the server busy-time in a closed form using known distributions. Figure 7(a) shows the CDF of server busy-time for the first server pool (highest quality servers) in a 1000 server experiment on EC2. More

details on the methodology can be found in Section 5. We show the experimental data (dots) and the closed form representation, derived from distribution-fitting. In this case, the data is fitted to a curve resembling a normal distribution. The CDF reflects the fraction of servers that are freed within an interval after they have been allocated to an application. For example, 60% of servers in that server pool are freed within 2700 sec from the time an application is scheduled to them.

Using this closed form CDF we easily derive the free-server CDF, which reflects the probability that within a time interval from an application's arrival, at least one server of the corresponding pool will be available. Figure 7(b) shows the free-server probability CDF for the first server pool. The highlighted point shows that there is a 60% probability that within 56 sec from an application's arrival to that queue, there will be at least one free server.

Switching between Queues. ARQ determines the switching point between queues to maximize the probability that a server will be available within a specific time window from an application's arrival. Let's assume for simplicity that the QoS constraint of an application is defined at 95% of the application's optimal performance. This means that the workload can tolerate at most a 5% performance degradation. Given the free-server probability CDFs for each server pool, ARQ solves the following optimization problem for an application a, switching between queues i and j:

$$\begin{aligned} max\ &\{(S_a - wt_i(t)) \cdot Q_i \cdot Pr_i[t], (S_a - wt_j(t)) \cdot Q_j \cdot Pr_j[t]\} \\ s.t.\ &(wt_i(t) + wt_j(t) + P_a) < 0.05 \cdot CT_a, \end{aligned} \tag{2}$$

where $Pr_i[t]$ is the probability that there is a free server in queue i; Q_i is the resource quality of queue i; CT_a is the optimal computation time for application a; P_a is the classification overheads of Paragon; and $S_a = 1.05 \cdot CT_a - P_a$ is the available "slack" that can be used for queueing, before the application violates its QoS constraints. ARQ finds the switching time that maximizes the probability that a server of either queue i or j will become available such that the application will preserve its QoS guarantees. It also promotes waiting longer for a server of the appropriate class rather than switching eagerly to the next queue ($Q_i > Q_j$).

4.3. Stability Analysis

A queueing network is stable if the total expected number of jobs in the network remains bounded as a function of time. Stability of multiclass queueing networks has been studied extensively [Dai 1995, 1996; Gamarnik 2000; Hasenbein 1998] both in the context of deterministic and stochastic arrival processes, using fluid limit models or Lyapunov functions. Here we briefly outline the conditions that guarantee stability in ARQ. Since each future state in the network *depends* only on the current, and not on past states, the system can be modeled as a Markov chain (MC). Suppose λ_i is the external arrival rate for application class i and μ_i is the corresponding service rate. P is the routing matrix, where P_{ij} is the traffic diverged from class i to class j. Then the traffic equation is $\bar{\lambda} = \lambda + P^T\bar{\lambda}$, with the explicit solution $\bar{\lambda} = [I - P^T]^{-1}\lambda$. In this formulation we assume for simplicity that diverged jobs are placed in the tail of the new queue, as opposed to the head. This assumption does not affect the stability conditions of the network. Also, $\rho_S \triangleq \sum_{i \in S} \bar{\lambda_i}/\mu_i$ is the load factor of server S. For Poisson arrival and service processes, the stability of the Markov process is equated with positive (Harris) recurrence for the corresponding queue length process $Q(s) = (Q_i(s), s \geq 0)$. Assuming no jobs are dropped from the network, this reduces to the simple traffic condition: $\rho_S < 1\ \forall S$ to guarantee stability.

Table V. Main Characteristics of Servers of the Local Cluster. (The total core count is 178 for 40 servers of 10 different SCs.)

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

4.4. Additional Policies

ARQ can incorporate additional optimizations to prioritize application scheduling based on, for example, their expected *computation time* or their *priorities*. These optimizations are orthogonal to the principal design of the admission control protocol and may require additional information about the scheduled workloads.

Computation Time. Shortest Job First (SJF) is a well-known algorithm [Tanenbaum 2007] that prioritizes the execution of short over long-running tasks. It improves the system's throughput by completing more tasks in a shorter time while preserving their corresponding QoS requirements. SJF can be implemented in ARQ. Jobs with short expected computation times are scheduled before long-running jobs in each queue. However, given that the QoS requirements of every application must be preserved, SJF is applied with the additional constraint that scheduling of long-running jobs can only be delayed for as long as their QoS guarantees allow.

Priorities. ARQ can incorporate the concept of priorities by scheduling more critical applications first. Although the scheduler attempts to preserve QoS for all submitted workloads, in the event where only some workloads can meet their performance, requirements the scheduler will prioritize the critical over noncritical applications.

5. 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 V. 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 in the cluster.

For the cloud-based clusters we used exclusive (reserved) server instances, that is, 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 1000 instances on Amazon EC2 [Amazon EC2] with 14 different SCs, ranging from small, single-core instances to high-end, quad-socket, multicore machines with hundreds of GBs of memory. All 1000 machines are private, that is, there is no interference in the experiments from external workloads. We also conducted experiments with 500 servers on Windows Azure [Windows Azure] with 8 different SCs and 100 servers on Google Compute Engine [GCE] 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

Table VI. Main Characteristics of Instances of the EC2 Cluster. (The total core count is 7339 for 1000 instances of 14 different SCs.)

Instance Type	Processor Arch	vCPUs	Memory (GB)	#
m1.small	64-bit	1	1.7	78
m1.medium	64-bit	1	3.75	69
m1.large	64-bit	2	7.5	82
m1.xlarge	64-bit	4	15	81
m3.xlarge	64-bit	4	15	82
m3.2xlarge	64-bit	8	30	68
c1.medium	64-bit	2	1.7	67
c1.xlarge	64-bit	8	7	65
m2.xlarge	64-bit	2	17.1	73
m2.2xlarge	64-bit	4	34.2	70
m2.4xlarge	64-bit	8	68.4	58
cr1.8xlarge	64-bit	32	244	62
hi1.4xlarge	64-bit	16	60.5	72
hs1.8xlarge	64-bit	16	117	73

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 multithreaded (MT), and 350 multiprogrammed (MP) workloads and 25 I/O-bound workloads. We use the full SPEC CPU2006 suite and workloads from PARSEC [Bienia et al. 2008] (*blackscholes, bodytrack, facesim, ferret, fluidanimate, raytrace, swaptions, canneal*); SPLASH-2 [Woo et al. 1995] (*barnes, fft, lu, ocean, radix, water*); BioParallel [Jaleel et al. 2006] (*genenet, svm*); Minebench [Narayanan et al. 2006] (*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 Sanchez and Kozyrakis [2011]. The I/O-bound workloads are data mining applications, such as clustering and recommender systems [Rajaraman and Ullman 2011], 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 interarrival 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, that is, 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 interarrival times. Second, a medium-load scenario with 178 applications, randomly selected as before and submitted with interarrival 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 oversubscribed one. After a few seconds, there are not enough resources in the system 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 2500 randomly-chosen applications submitted with 1 sec intervals; a high-load scenario with 5000 applications submitted with 1 sec intervals; and an oversubscribed scenario where 7500 workloads are submitted with 1 sec intervals and an additional 1000 applications arrive in burst (less than 0.1 sec intervals) after the first 3750 workloads.

6. EVALUATION

6.1. Comparison of Schedulers: Small Scale

QoS Guarantees. Figure 8 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. The 79% of workloads maintain their QoS with Paragon, while the heterogeneity-oblivious (NH), interference-oblivious (NI), and least-loaded (LL) schedulers provide similar guarantees for only 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 (95% 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 coscheduled workloads. We expect positive interference to be less prevalent for a more diverse application space.

Scheduling Decision Quality. Figure 9 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

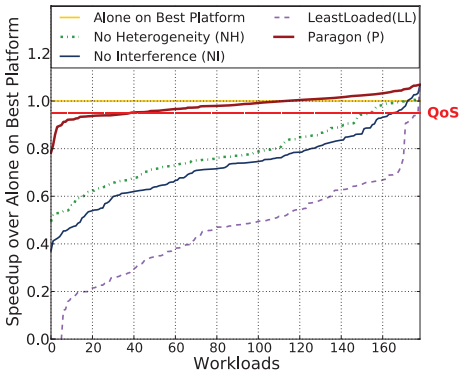


Fig. 8. Performance impact from scheduling with Paragon for *medium-load*, compared to heterogeneity and/or interference-oblivious schedulers. Application arrival times follow a Gaussian distribution. Applications are ordered from worst to best.

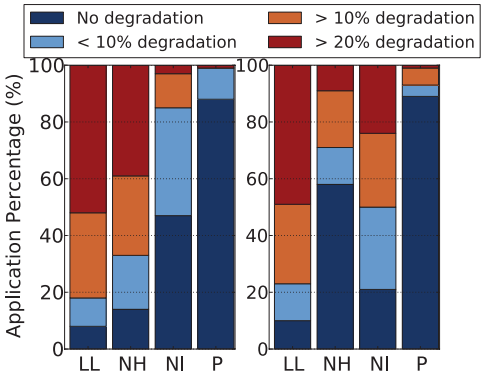


Fig. 9. Breakdown of decision quality for heterogeneity (left) and interference (right) for the medium-load on the local cluster. Applications are divided based on performance degradation induced by the decisions of each scheduler.

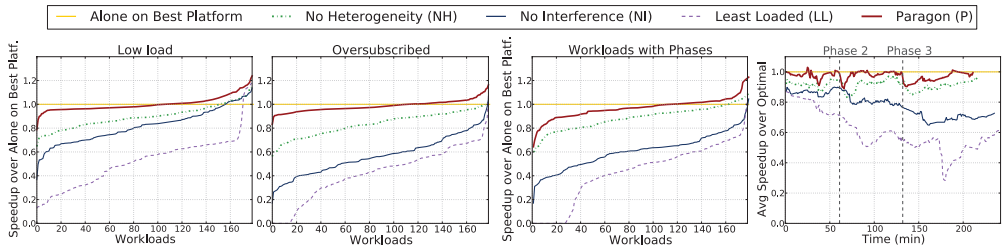
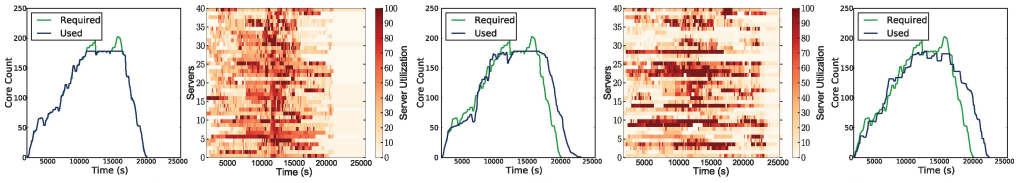


Fig. 10. Performance comparison between the four schedulers for three workload scenarios: low, oversubscribed, and workloads with phases (Figure 10 (a), (b), (c)) and performance over time for the scenario where workloads experience phases (Figure 10(d)).

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 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 88% of applications to their optimal SC.

The right part in Figure 9 shows decision quality with respect to interference. LL behaves the worst for similar reasons, while NI is slightly better than LL because 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 89% 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 in terms of both heterogeneity and interference.

Other Workload Scenarios. Figure 10 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



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

Fig. 11. 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.

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, 82.5% 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, reclassifying the offending workloads and rescheduling them if a more suitable server is available. To verify this, in Figure 10(d) 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 bounces back to values close to 1, as the deviating workloads are rescheduled 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 11(a) shows how Paragon follows the resource requirements for the medium load scenario shown in Figure 8. The green line shows the required core count of active applications based on arrival rate and ideal execution time, and the blue line shows 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 11(b)) and similarly for NH, either overprovisions or oversubscribes servers, resulting in increased execution time; per-application and for the overall scenario. Finally, Figure 11(c) shows the resource allocation for the least-loaded scheduler. There is significant deviation, since the scheduler ignores both

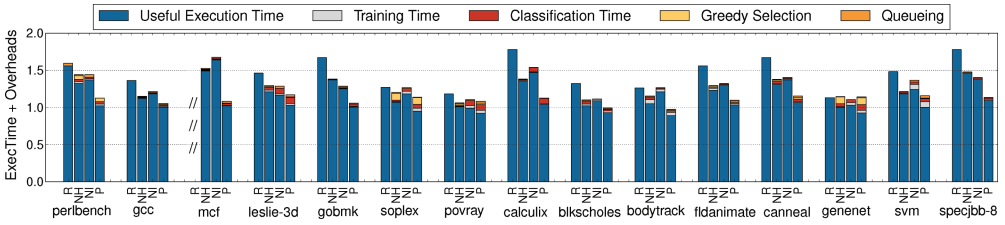


Fig. 12. Execution time breakdown for selected single-threaded and multithreaded applications in the medium load scenario.

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 11 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 middle 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 for both 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, queueing, and server selection using the greedy algorithm. Figure 12 shows the execution time breakdown for selected single-threaded and multithreaded 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 from queueing are less than 1.5% in all cases, while 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.

6.2. Comparison of Schedulers: Large Scale

Performance Impact. Figure 13 shows the performance for the three workload scenarios on the 1000-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

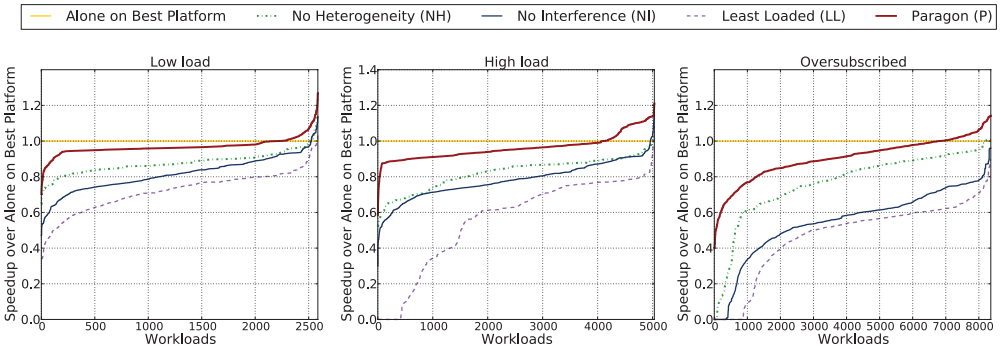


Fig. 13. Performance comparison between the four schedulers, for three workload scenarios on 1,000 EC2 servers.

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 by only 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 ARQ forces certain workloads to wait until resources are made available, the total performance degradation is still bounded and small (only 0.6% of workloads degrade more than 20%), since it coschedules workloads that minimize destructive interference. Without the use of the admission control protocol, performance degradation is higher, with 56% of workloads maintaining their QoS, which is still significantly higher than for the heterogeneity and/or interference-oblivious schemes.

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 to 1000]).

Decision Quality. Figure 14 shows a breakdown in 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 15 shows why this deviation exists. 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

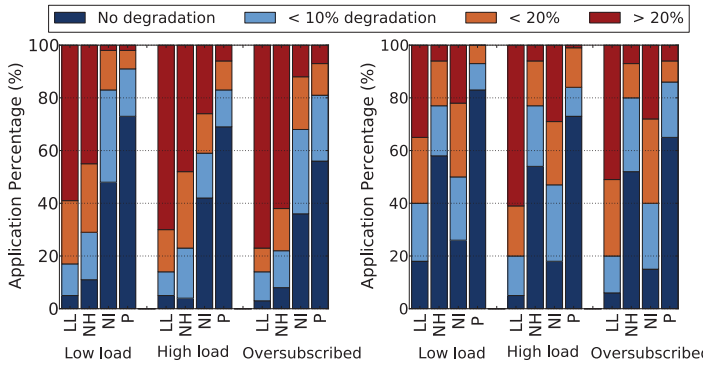


Fig. 14. Breakdown of decision quality in terms of heterogeneity (left) and interference for the three EC2 scenarios.

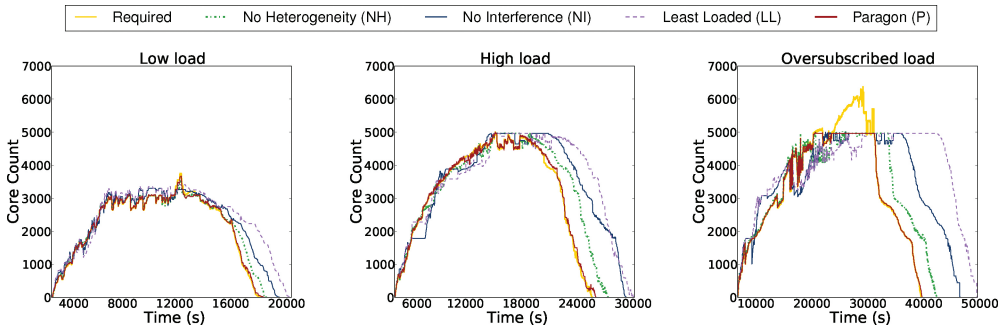


Fig. 15. Comparison of required and performed core allocation between Paragon and the other three schedulers for the three workload scenarios on EC2. The total number of cores in the system is 4960.

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 and 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 2500 and 500 workloads, respectively. For 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

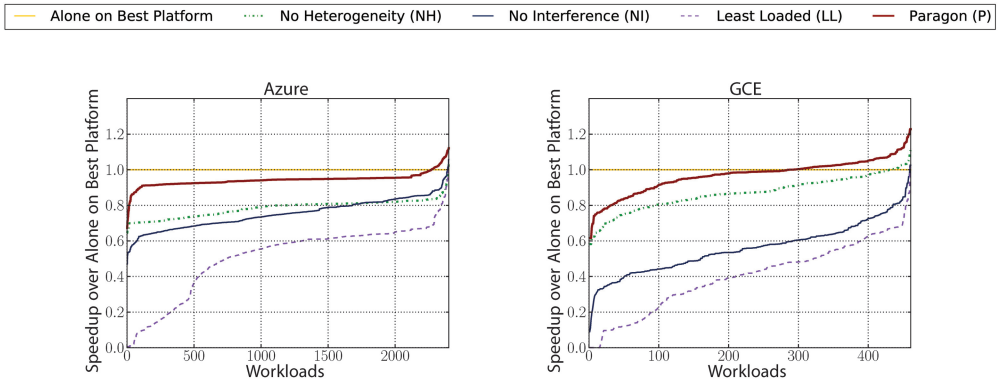


Fig. 16. Performance comparison between the schedulers on the Windows Azure and Google Compute Engine (GCE) clusters.

consistency between experiments, despite the different cluster configurations and underlying hardware, shows the robustness of the analytical methods that drive Paragon.

7. 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 multicore chips.

Datacenter Scheduling. Recent work on DC scheduling has highlighted the importance of platform heterogeneity and workload interference. Mars et al. [2013, 2011] 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 2× due to interference even when considering only two colocated applications. In Mars and Tang [2013], they present a system that uses combinatorial optimization to select the proper server configuration for a given workload. Mars et al. [2011] present a two-step method to characterize the sensitivity of workloads to memory pressure and the stress each application exercises to the memory subsystem. In the same spirit, Yang et al. [2013] apply a dynamic interference sensitivity detection scheme to preserve the performance of batch and latency-critical applications under colocation scenarios. Govindan et al. [2011] also present a scheme to quantify the effects of cache interference between consolidated workloads, although they require access to physical memory addresses. Zhang et al. [2013] use cycles-per-instruction (CPI) as a proxy for interference between workloads and throttle the offending corunners such that the applications return to their expected behavior. Finally, Nathuji et al. [2010] present a control-based resource allocation scheme that mitigates the effects of cache, memory, and hardware prefetching interference between coscheduled 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 [VMWare vSphere], XenServer [Xenserver], or the VM platforms on EC2 [Amazon EC2], and Windows Azure [Windows Azure] 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. VMWare's Distributed Resources Scheduler (DRS) [VMWare-DRS 2012], for example, accounts for CPU and memory requirements when scheduling applications. Recently, DeepDive [Novakovič et al. 2013] proposed a black-box system for management of virtual machines, which accounts for interference while keeping any migrations overheads minimal. 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 (rescheduling).

Resource Management and Rightsizing. There has been significant work on resource allocation in virtualized and nonvirtualized large-scale DCs, including Mesos [Hindman et al. 2011], Rightscale [Rightscale], CloudScale [Shen et al. 2011], Omega [Schwarzkopf et al. 2013]; resource containers [Banga et al. 1999], Dejavu [Vasić et al. 2012]; and the work by Chase et al. [2001]. Mesos performs resource allocation between distributed computing frameworks like Hadoop or Spark [Hindman et al. 2011; Zaharia et al. 2012]. Rightscale automatically scales out three-tier applications to react to changes in the load in Amazon's cloud service [Amazon EC2]. CloudScale identifies application resource requirements using online demand prediction and prediction error handling, without a priori assumptions on application behavior [Shen et al. 2011]. Omega is a shared-state two-level scheduler that exposes the full cluster state to each scheduler of individual frameworks, improving utilization [Schwarzkopf et al. 2013]. Dejavu serves a similar goal by identifying a few workload classes, and, based on them, reuses previous resource allocations to minimize reallocation overheads [Vasić et al. 2012]. Zhu et al. [2009] present a resource management scheme for virtualized DCs that preserves SLAs, and Gmach et al. [2007] provides 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 of long running, three-tier services in future work.

Scheduling for Heterogeneous Multicore 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. [2007] discuss OS-level scheduling for heterogeneous multicores as having the following three objectives: optimal performance, core assignment balance, and response time fairness. Shelepov et al. [2009] present a scheduler that exhibits some of these features and is simple and scalable, while Craeynest et al. [2012] 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 coscheduled workloads should be preserved. Given the increasing number of cores per chip and of coscheduled 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.

8. CONCLUSIONS

We have presented Paragon, a scalable scheduler for DCs that is both heterogeneity and interference-aware. Paragon uses 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. Paragon also tracks workload changes and adjusts scheduling decisions at runtime to avoid performance degradations. 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 applications and bounds degradation to less than 10% for an additional 33% out of 8500 applications on a 1000-server cluster. Paragon preserves QoS guarantees while improving server utilization, hence it benefits both the DC operator, who achieves better 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.

ACKNOWLEDGMENTS

We sincerely thank John Ousterhout, Mendel Rosenblum, Byung-Gon Chun, Daniel Sanchez, Jacob Leverich, David Lo, and the anonymous reviewers for their feedback on earlier versions of this manuscript.

REFERENCES

Alameldeen, A. R. and Wood, D. A. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro (Special Issue on Computer Architecture Simulation and Modeling)*.

Amazon EC2. http://aws.amazon.com/ec2/.

Banga, G., Druschel, P., and Mogul, J. C. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*.

Barroso, L. 2011. Warehouse-scale computing: entering the teenage decade. In *Proceedings of ISCA*.

Barroso, L. and Hoelzle, U. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool.

Bell, R. M., Koren, Y., and Volinsky, C. 2007. The BellKor 2008 solution to the Netflix Prize. Tech. rep., AT&T Labs.

Bertsimas, D., Gamarnik, D., and Tsitsiklis, J. N. 2001. Performance of multiclass Markovian queueing networks via piecewise linear Lyapunov functions. *Ann. Appl. Probab. 11*, 1384–1428.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Bottou, L. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*.

Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., ul Haq, M. F., ul Haq, M. I., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. 2011. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*.

Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. 2001. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*.

Craeynest, K. V., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. 2012. Scheduling heterogeneous multicores through performance impact estimation (PIE). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

Dai, J. G. 1995. On positive Harris recurrence of multiclass queueing networks: A unified approach via fluid limit models. *Ann. Appl. Probab. 5*, 49–77.

Dai, J. G. 1996. A fluid-limit model criterion for instability of multiclass queueing networks. *Ann. Appl. Probab. 6*, 751–757.

DELIMITROU, C. AND KOZYRAKIS, C. 2013a. iBench: Quantifying interference for datacenter applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.

DELIMITROU, C. AND KOZYRAKIS, C. 2013b. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

DELIMITROU, C. AND KOZYRAKIS, C. 2013c. The Netflix challenge: Datacenter edition. *IEEE Comput. Archit. Lett.* (June).

FEDOROVA, A., SELTZER, M., AND SMITH, M. D. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

GAMARNIK, D. 2000. On deciding stability of scheduling policies in queuing systems. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. 467–476.

GOOGLE COMPUTE ENGINE GCE. http://cloud.google.com/products/compute-engine.html.

GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

GMACH, D., ROLIA, J., CHERKASOVA, L., AND KEMPER, A. 2007. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 10th IEEE International Symposium on Workload Characterization (IISWC)*.

GOVINDAN, S., LIU, J., KANSAL, A., AND SIVASUBRAMANIAM, A. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*.

HAMILTON, J. 2009. Internet-scale service infrastructure efficiency. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*.

HAMILTON, J. 2010. Cost of power in large-scale data centers. http://perspectives.mvdirona.com.

HASENBEIN, J. J. 1998. Stability, capacity, and scheduling of multiclass queuing networks. Ph.D. dissertation, Georgia Institute of Technology.

HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

JALEEL, A., MATTINA, M., AND JACOB, B. L. 2006. Last level cache (LLC) performance of data mining workloads on a CMP—A case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*.

KATZ, J. AND LINDELL, Y. 2007. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Press.

KIWIEL, K. C. 2001. Convergence and efficiency of subgradient methods for quasiconvex minimization. *Math. Program. (Series A)*, *90*, 1, 1–25.

KOZYRAKIS, C., KANSAL, A., SANKAR, S., AND VAID, K. 2010. Server engineering insights for large-scale online services. IEEE *Micro 30*, 4, 8–19. DOI:http://dx.doi.org/10.1109/MM.2010.73.

LEVERICH, J. AND KOZYRAKIS, C. 2010. On the energy (in)efficiency of Hadoop clusters. *SIGOPS Oper. Syst. Rev. 44*, 1, 61–65.

LIN, J. AND KOLCZ, A. 2012. Large-scale machine learning at Twitter. In *Proceedings of the ACM SIGMOD Conference*.

MARS, J. AND TANG, L. 2013. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

MARS, J., TANG, L., AND HUNDT, R. 2011. Heterogeneity in "homogeneous"; warehouse-scale computers: A performance opportunity. *IEEE Comput. Archit. Lett. 10*, 2, 29–32. DOI:http://dx.doi.org/10.1109/L-CA.2011.14.

MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. 2011. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*.

MILLER, B. L. 1969. A queuing reward system with several customer classes. *Manage. Sci. 16*, 3, 234–245.

NARAYANAN, R., OZISIKYILMAZ, B., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. N. 2006. MineBench: A benchmark suite for data mining workloads. In *Proceedings of the 9th IEEE International Symposium on Workload Characterization (IISWC)*.

NATHUJI, R., ISCI, C., AND GORBATOV, E. 2007. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*.

NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. 2010. Q-Clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*.

Novakovič, D., Vasič, N., Novakovič, S., Kostič, D., and Bianchini, R. 2013. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

Rackspace. Open Cloud. http://www.rackspace.com/.

Rajaraman, A. and Ullman, J. 2011. *Textbook on Mining of Massive Datasets. Rightscale*. https://aws.amazon.com/solution-providers/isv/rightscale.

Sanchez, D. and Kozyrakis, C. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium in Computer Architecture (ISCA-38)*.

Schein, A., Popescul, A., Ungar, L., and Pennock, D. 2002. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*.

Shelepov, D., Alcaide, J. C. S., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S., and Kumar, V. 2009. HASS: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev. 43*, 2.

Shen, Z., Subbiah, S., Gu, X., and Wilkes, J. 2011. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*.

Sun, J., Xie, Y., Zhang, H., and Faloutsos, C. 2008. Less is more: Compact matrix decomposition for large sparse graphs. *J. Stat. Anal. Data Mining 1*, 1.

Tanenbaum, A. S. 2007. *Modern Operating Systems*. 3rd Ed. Peason Education, Inc.

Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R. 2012. Deja vu: accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

vMotion. Migrate VMs with Zero Downtime. http//www.vmware.com/products/vmotion.

VMWare-DRS. 2012. Distributed resource scheduler: design, implementation and lessons learned. *VMware Tech. J. 1*, 1.

VMWare vSphere. http://www.vmware.com/products/vsphere/.

Weng, L.-T., Yue, X., Yuefeng, L., and Nayak, R. 2008. Exploiting item taxonomy for solving cold-start problem in recommendation making. In *Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.

Wenisch, T. F., Wunderlich, R. E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. C. 2006. SimFlex: Statistical sampling of computer system simulation. *IEEE MICRO 26*, 4.

Windows Azure. http://www.windowsazure.com/.

Witten, I. H., Frank, E., and Holmes, G. 2011. *Data Mining: Practical Machine Learning Tools and Techniques.* 3rd Ed.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*.

Xenserver. 6.1. http://www.citrix.com/xenserver/.

Yang, H., Breslow, A., Mars, J., and Tang, L. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. 2012. Spark: Cluster computing with working sets. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., and Wilkes, J. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*.

Zhang, Z.-K., Liu, C., Zhang, Y.-C., and Zhou, T. 2010. Solving the cold-start problem in recommender systems with social tags. arXiv:1004.3732v2.

Zhu, X., Young, D., Watson, B. J., Wang, Z., Rolia, J., Singhal, S., Mckee, B., Hyser, C., Gmach, D., Gardner, R., Christian, T., and Cherkasova, L. 2009. 1000 Islands: An integrated approach to resource management for virtualized datacenters. *J. Cluster Comput. 12*, 1.

Received May 2013; revised September 2013; accepted September 2013