sPARE: Partial Replication for Multi-tier Applications in the Cloud

Robert Birke, *Senior Member, IEEE,* Juan F. Pérez, *Member, IEEE,* Zhan Qiu, *Member, IEEE*
Mathias Björkqvist, *Member, IEEE* and Lydia Y. Chen, *Senior Member, IEEE*

Abstract—Offering consistent low latency remains a key challenge for distributed applications, especially when deployed on the cloud where virtual machines (VMs) suffer from capacity variability caused by co-located tenants. Replicating redundant requests was shown to be an effective mechanism to defend application performance from high capacity variability. While the prior art centers on single-tier systems, it still remains an open question how to design replication strategies for distributed multi-tier systems. In this paper, we design a first of its kind PArtial REplication system, sPARE, that replicates and dispatches read-only workloads for distributed multi-tier web applications. The two key components of sPARE are (i) the variability-aware replicator that coordinates the replication levels on all tiers via an iterative searching algorithm, and (ii) the replication-aware arbiter that uses a novel token-based arbitration algorithm (TAD) to dispatch requests in each tier. We evaluate sPARE on web serving and searching applications, i.e., MediaWiki and Solr, the former deployed on our private cloud and the latter on Amazon EC2. Our results based on various interference patterns and traffic loads show that sPARE is able to improve the tail latency of MediaWiki and Solr by a factor of almost 2.7x and 2.9x, respectively.

Index Terms—Cloud, replication, tail latency, models,load balancing

1 Introduction

Performance variability is considered one of the major pitfalls in the cloud computing paradigm [1, 2], because the virtualization technology does not guarantee performance isolation [3]. Applications hosted in the cloud are thus subject to interference from unknown neighboring workloads. The more distributed an application is, the higher the probability that certain components experience interference and capacity drops. Examples include modern web applications with standard multi-tier architectures [4], where each server[1] can exhibit time-varying capacity. An effective, yet expensive, solution to combat performance variability in the cloud and to fulfill service level agreements (SLAs), defined by the tail latency, is to scale out the provisioned resources in the unit of virtual machines (VMs) [5, 6]. Nonetheless, the amount of resources required to fulfill SLAs in a cloud environment with highly varying VM capacities can grow rapidly [7, 8], and thus hinder its applicability to highly distributed web applications.

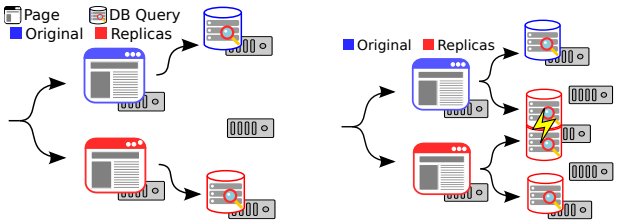
Replicating redundant requests is an inexpensive and yet effective alternative to improve the tail latency of web applications [9, 10] and to mitigate the effect of stragglers in big data applications [11, 12], particularly addressing the issues of performance variability among computing units. Another argument to support the replication strategy is that the existing cloud datacenters [13, 14] are under utilized, say around 20%, for most of time, meaning there are resources available to serve redundant requests. Redundant requests are issued either right upon their arrival [11, 9] or after detecting slow servers [15], and only the result of the first request replica that completes processing is returned to the user.

The effectiveness of replication depends on the trade-off between the overhead of processing additional loads and the potential performance gain of processing request replicas at fast servers [16, 17]. Furthermore, the variability of the processing times has been shown to be central to the effectiveness of replication on the latency tail [17, 10]. The main challenges of developing replication strategies are thus to find optimal request redundancy levels based on the observed variability, and to determine how to best process and schedule redundant requests. To the best of our knowledge, all existing studies have centered on a particular application tier, such as Domain name server and file transfer [17], or map-reduce-like applications, e.g., SPARK [15]. However, how to design general replication strategies for distributed multi-tier web applications hosted in the cloud largely remains an open challenge, given the workload interdependency across tiers and the need to avoid communication overheads among tiers [18].

In this paper, we develop a **PA**rtial **RE**plication system, termed sPARE, to exploit workload redundancy for distributed multi-tier web applications undergoing strong capacity variability, as for instance in the cloud. We particularly focus on read-dominated or read-only workloads, as they represent a significant class of workloads in today's web traffic, e.g., only 0.03% of requests to the Wikipedia website result in a page update or creation [19]. Partial replication in sPARE means that disparate replication factors are defined for each tier, launching redundant requests to mitigate and exploit the high variability experienced at specific tiers.The aim of sPARE is to use redundant requests to increase their chances of being processed at fast servers.To this end, the two key features of sPARE are: i) a centralized replicator that coordinates the

- *R. Birke is with ABB Corporate Research, Switzerland. This work was done while he was at IBM Research Zurich Lab, Switzerland. E-mail: robert.birke@ch.abb.com*
- *M. Börkqvist and L. Y. Chen are with IBM Research Zurich Lab, Switzerland. E-mail: {bir, mbj, yic}@zurich.ibm.com*
- *J. F. Pérez is with the Department of Applied Mathematics and Computer Science, Universidad del Rosario, Colombia. E-mail: juanferna.perez@urosario.edu.co*
- *Z. Qiu is with the Department of Computing, Imperial College London, UK. E-mail: {z.qiu11}@imperial.ac.uk*

1. Servers here refer to software components.



(a) Page replicated (b) Page and query replicated

Fig. 1: Flows of replicated pages and queries for MediaWiki by an example of one page containing one DB query. Each system has front-end Apache and back-end MySQL DB servers.

replication levels at all tiers based on the estimated capacity and observed latency variability, and ii) distributed arbiters at each tier that dispatch requests to servers for which we propose a novel token-based arbitration policy (TAD).

We test sPARE on multi-tier web serving and web searching applications, namely MediaWiki [20] and Solr [21], deployed at our private cloud and in the wild on Amazon EC2. Our extensive evaluation results for different combinations of interference patterns and loads show that sPARE is able to improve the latency, particularly the tail, by almost a factor of three when compared to the original non-replicated system. The performance advantage of sPARE in reducing the latency is particularly significant when the load and the interference from neighboring workloads are higher.

The specific contributions arising from the design of sPARE are three-fold. First, we develop a first of its kind replication strategy, sPARE, for multi-tier systems, which is able to adaptively replicate requests at different tiers according to the observed capacity variability, turning this variability into a performance advantage, particularly for the latency tail. Secondly, the proposed arbitration policy, TAD, agilely adapts to server capacity variations and uses the aggregate tier capacity, instead of being restricted by the variable per server capacity. Last but not least, the proposed searching algorithm is aware of the time-variability in each tier capacity and is able to reach near-optimal replication levels within a few iterations for a wide range of loads and interference patterns.

2 The Case for Partial Replication

In this section, we illustrate how replication might (not) work out of the box for distributed web applications hosted in the cloud, where virtual machines undergo different degrees of capacity variability due to neighboring effects. We base our description on the MediaWiki application [20], the open source platform for Wikipedia, as an example, though it applies in general for multi-tier applications. Fig. 1 provides a high level overview of MediaWiki and its main components: multiple front-end Apache servers and multiple back-end database (DB) servers. In the remainder we interchangeably use front-end (back-end) and Apache (DB) servers. Additionally, both front-end and back-end have dispatchers in front of the corresponding servers. The performance metrics of interest are the mean and tail latency, e.g., 99^{th} percentile, to retrieve a complete page of Wikipedia. The full specification of MediaWiki and its setup can be found in Section 6.

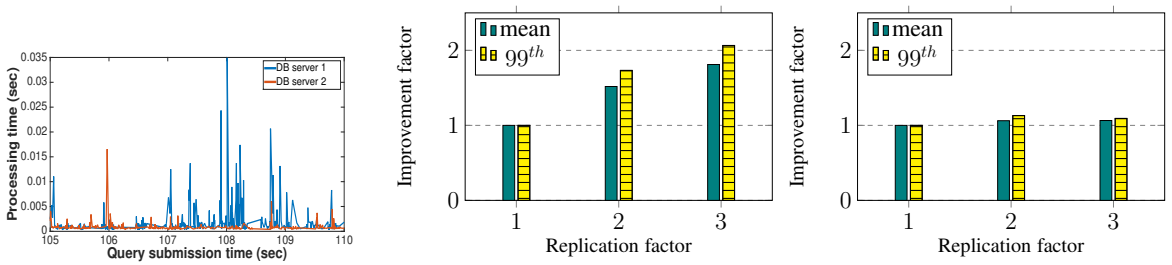
To serve a Wiki page request, multiple queries need to be retrieved from the DB. Upon arrival of a Wiki page request, the front-end dispatcher sends the request to one of the Apache servers according to the round-robin policy – a load oblivious strategy. The default back-end load dispatcher of MediaWiki then dispatches each query to one of the DB servers. To see the impact of replication, we implement standard speculative replication by modifying the front-end dispatcher and replicating each page upon arrival by a factor r, without considering other tiers separately. Thus, there are r identical replicas for each page request and only the result obtained from the first completed replica is returned. Fig. 1(a) illustrates the flow of replicated page requests with an example where a page containing a single DB query is replicated. The front-end dispatcher sends two page request replicas to two different Apache servers, which then independently execute one DB query at one of the DB servers. As a result, the workload of both Apache and DB servers increases by a factor of two.

To illustrate the effect of replication under different neighboring effects, our motivating example considers two types of interference at (i) Apache servers only, and (ii) DB servers only. The inference thus happens either at the front-end or at the back-end. To emulate the neighboring effects, we collocate `iperf` [22] to create random network transfers between pairs of VMs. Due to the non-negligible CPU overhead in processing traffic in a virtualized environment [23], `iperf` causes both CPU and network contention. Corresponding to the two interference patterns, we activate `iperf` at either tier individually, causing the effective capacity of the Apache and DB servers to vary greatly. We explain the patterns of activating `iperf` in detail in Section 6. Fig. 2(a) depicts the query processing time at two different DB servers during an observation window of 5 seconds, where time-varying processing times of each server are clearly visible, as is the spatial variability across servers at any given point in time. Consequently, during most of the observation period one of these DB servers offers lower processing times, a feature that can be exploited by a scheduler that is aware of the servers speed.

To demonstrate the effectiveness of the replication strategy on latency reduction, we execute the MediaWiki application with replication factor r_1=$\{1, 2, 3\}$ at the front-end, i.e., replicating only the page requests, where r_1=1 corresponds to the baseline case without replication, and a request arrival rate of 20 requests per second. With an observed average of 60 DB queries per request, this entails an arrival rate at the back-end of around 1200 DB requests per second. Fig. 2(b) and (c) summarize the performance gains on the mean and the 99^{th} percentile of the page latency normalized by the case with $r = 1$, under different front-end replication factors, with respect to two interference patterns. An improvement factor larger than one indicates that replicating the pages by r times improves the performance, while a value less than one implies that the overhead caused by processing additional replicas defeats the potential performance gain.

One can clearly see that simply replicating Wiki pages can improve the mean and 99^{th} latencies under both interference patterns. Replication factor r=3 is able to achieve the most significant improvement for both metrics under both patterns, except for the 99^{th} percentile under the second pattern. However, while replication can achieve a factor of 1.5-2.0x improvement when `iperf` interferes with the Apache servers, there is barely a factor of 1.1x under the second interference pattern. Clearly, replicating *page requests* has a more significant effect if the variability is mostly experienced at the *front-end*, while it has little impact if the variability is present at the back-end.

The take-home message is that simple request redundancy is effective if the capacity variability is observed at the front-end, but it offers little gains when this variability is present at other



(a) Query processing time (b) `iperf` at MediaWiki VMs (c) `iperf` at DB VMs

Fig. 2: (a) Query processing time for 2 different DB servers and (b)-(c) normalized latency of MediaWiki application under two interference patterns and replication factors.

tiers. Consequently, we advocate *request partial replication*, where requests are replicated at those tiers experiencing high capacity variability, instead of uniformly replicating requests at each tier.

2.1 Challenges of Partial Replication

The design and implementation of replication strategies for distributed multi-tier systems faces several technical challenges. We explain these challenges using MediaWiki as a running example, where we need to determine replication factors (r_1, r_2) for front-end and back-end servers (see Fig. 1).

Multiplicative effect on replication loads, resulting in a tree-like structure. The amount of requests received at the back-end is essentially amplified by the multiplication of the replication factors at the front-end and at the back-end. Fig. 1(b) depicts an example where one DB query is embedded in the requested page. Whereas with no replication the query is retrieved only once at the back-end, with $(r_1, r_2) = (2, 2)$ the query is retrieved $2{*}2{=}4$ times. Introducing replication at both tiers significantly magnifies the workload at the back-end in a tree-like fashion and can risk the system stability.

Collision of replicated requests at the second tier. One of the basic principles of a replication policy is to send replicated requests to *multiple* servers so as to best take advantage of the diversity caused by their capacity variability. While it is straightforward to dispatch the replicated page requests to different Apache servers, there exists a high probability that DB queries originated from replicas of the same page request collide at a DB server. Specifically, DB queries embedded in a page might end up being retrieved from the same DB server, as highlighted in Fig. 1(b) by the yellow flash. We refer to this behavior as a *collision* as replicas of the same query end up being executed on the same back-end server, defeating the advantage of exploring the diversity of resources. Moreover, the collision probability grows with the number of queries embedded in a page, the number of tiers, and the replication factors.

Selecting "fast" VMs simultaneously on both tiers. To ensure the effectiveness of any partial replication strategy, one needs to ensure that fast servers are selected at both tiers. Let us consider again the example shown in Fig. 1(b) with replication factors $(2, 2)$ and three objects embedded in the page. To get the best result out of replication, we want at least one page replica to be processed by a fast MediaWiki server, and for such page replica at least one query replica of each embedded object to be processed by a fast DB server. Although this is a desirable goal, it is actually very challenging to achieve since the back-end load-balancer is unaware of the front-end requests and their replicas. Clearly, load-oblivious policies, like round-robin, fall short in addressing this challenge. such as the DB queries at the back-end.

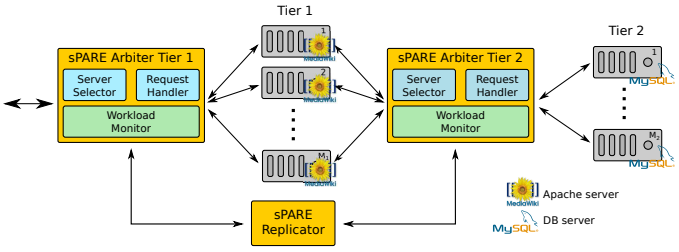


Fig. 3: Example architecture of sPARE with MediaWiki.

3 sPARE: Partial Replication System

We now introduce sPARE, a **PA**rtial **RE**plication system for distributed multi-tier applications, which determines the replication factors for all tiers and arbitrates the dispatching of requests (and their replicas). We particularly focus on read-only requests. To illustrate the design of sPARE, we continue our MediaWiki example, where replicating requests is possible at front-end Apache servers (tier 1) and at back-end DB servers (tier 2). We note that while a large body of related work [12, 15] centers on replicating requests reactively after detecting performance degradation, we focus on a proactive strategy since the "fast" system dynamics, e.g., hundreds of msec at tier 1 and few msec at tier 2 for MediaWiki, limit the benefits of a reactive approach given the delay necessary to identify potential stragglers and submit replicas. The underlying assumption made here is that the application is properly dimensioned, meaning that it has sufficient or redundant capacity to handle the incoming loads. In the following, we first detail out the design features of sPARE and conclude with the analysis of the collision probability at tier 2.

3.1 Architecture Overview

To achieve the dual goals of obtaining optimal replication factors and replication-aware arbitration, sPARE relies on two key components: a centralized replicator and distributed arbiters at each tier. These two components are depicted in Figure 3: one arbiter for each tier and one sPARE replicator. There are M_i servers at tier i. To ease the readability, we summarize all key notations used throughout this paper in Table 1.

TABLE 1: Summary of key notation and definitions

i	subscript for tier $i \in [1, N]$
M	number of servers
V	no. of virtual cores per server
s	maximum sustainable throughput of a VM without interference
r	replication factor
μ	average capacity of single core with interference
T	average processing time at a core and an inverse of μ
λ	arrival rate
P_n	non-collision probability
C	no. of token per server set in TAD
n	no. of toke look-up set in TAD
R^T	response time target
w	the measured variability used in searching alg.

Centralized replicator. The central replicator determines the replication factor for each tier, (r_1, r_2), based on the load and capacity variability at each tier, which are derived from statistics collected by the arbiters. Particularly, the replicator searches for (r_1^*, r_2^*) within a set of boundary conditions that ensure the system stability and pose a bound on the collision probability, as described in detail in Section 5.

Distributed arbiters. The arbiters actuate the replication decisions of the replicator by replicating requests and dispatching them to the servers in their tier. Each arbiter implements three logical blocks: server selector, request handler, and workload monitor. The server selector is responsible for choosing *fast* servers for the request handler, whereas the request handler is responsible for cloning the incoming requests and dispatching them to the servers in the tier. The workload monitor passively collects the key statistics required by the replicator.

sPARE offers support for two types of arbitration policies, namely round-robin (RR) and TAD. Whereas RR is load oblivious and distributes requests immediately to the servers at the tier, TAD is aware of the load and capacity variability at each tier. Although RR is known to have a robust performance and low implementation overhead, we expect a load-aware arbitration, such as the proposed TAD, to be able to better sustain the extra workloads introduced by replication. Moreover, TAD increases the probability that requests are processed by fast servers hosted on VMs with low interference.

3.1.1 Connection Reuse

Another feature of the sPARE arbiter is its ability to reuse connections. To reduce the overhead to process the extra load created by replication, particularly for the MySQL protocol, sPARE reuses connections to the servers across different requests. Essentially, connection reuse can avoid the latency overhead not only of the TCP three-way handshake, but also of any protocol specific connection setup phases. Whereas the impact is limited for HTTP requests with no setup phases, this optimization greatly benefits the MySQL protocol, which includes an initial handshake and authentication phase.

3.2 Tier 2 Collision Probability

Each arbiter fully manages the local replication within its tier, hence it is straightforward to select different servers for the local replicas. However, each arbiter is unaware of the tree-like relationship between local requests and child requests in later tiers, such as the relationship between page requests and DB queries in the MediaWiki example. Consequently, DB queries originated from the same page request can be sent to the same DB server in tier 2. We define the tier 2 collision probability P_c, under replication factors (r_1, r_2), as the probability that at least one DB server receives more than one query from the same page request. Thanks to the connection re-use mechanism, this is equal to the probability that at least one DB server receives more than one *connection* from the same page request.

If r_1=1, P_c is zero, since each page request is submitted only once to tier 1, and the tier 2 arbiter avoids all collisions between DB queries of the same page replica. For the case r_1>1, to obtain P_c we focus on the non-collision probability P_n=1−P_c to derive the following.

Proposition 1. The tier 2 non-collision probability under replication factors (r_1, r_2) is

$$P_n = \frac{\prod_{j=0}^{r_1-1} \binom{M_2 - j r_2}{r_2}}{\binom{M_2}{r_2}^{r_1}}. \quad (1)$$

The proof comes straightforwardly as there are no tier 2 collisions if all $r_1 r_2$ connections are made to different DB servers. We note that when the total number of query replicas is at least as large as the number of tier 2 servers, i.e., $r_1 r_2 \geq M_2$, the collision probability is simply one.

Replicas that collide can not benefit from the diversity of capacity variability, defeating the goal of replication. One can also see from Eq. (1) that the collision probability increases very fast with both r_1 and r_2. Achieving a low collision probability may require the replication factor r_1 or r_2 to be very small, limiting the benefits of exploring the resource diversity at either tier. Thus a trade-off arises as the overall benefit of one replication factor with a higher collision probability may outperform the one with a lower collision probability.

3.3 Tier N Collision Probability

To extend the above result to the N-tier case we define the amplification factor $a_{i,j}$, which is the total number of tier-j requests that correspond to each request in tier i, with $1 \leq i \leq j \leq N$. Thus, under replication factors $(r_1, r_2, \ldots, r_N)$, $a_{i,j} = \prod_{k=i}^{j} r_k$. Next, we note that tier j receives $a_{i,j-1}$ requests for each request in tier i, each of which it processes independently, replicating them r_j times. We can thus state the following result for $P_c^{i,j}$, the collision probability of tier-i requests at tier j, and its complement $P_n^{i,j}$.

Proposition 2. The non-collision probability $P_n^{i,j} = 1 - P_c^{i,j}$ under replication factors $(r_1, r_2, \ldots, r_N)$ is

$$P_n^{i,j} = \frac{\prod_{k=0}^{a_{i,j-1}-1} \binom{M_j - k r_j}{r_j}}{\binom{M_j}{r_j}^{a_{i,j-1}}}. \quad (2)$$

The key difference between (2) and (1) lies in the use of the factor $a_{i,j-1}$ to capture the traffic amplification caused by replication between tiers i and j. As before, if the amplification factor $a_{i,j}$ is one, the collision probability is simply zero. Finally, as we are interested in the collision probability at the first tier, the tier-N non-collision probability is given by (2) with i=1 and j=N.

In the Supplementary Material we generalize the expression for the non-collision probability. Specifically, we allow a request to hit a subset of all tiers, and consider the constraint that a request may be served by only some of the servers in a tier, e.g., because the data requested is located in some of the servers only.

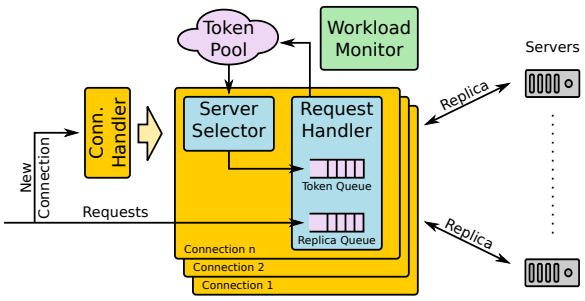


Fig. 4: Arbiter design with TAD.

4 TAD: TOKEN-BASED ARBITRATION

The objective of the TAD policy is to explore the spatial capacity variability across servers using replicated requests. The key design principle of TAD is a lightweight mechanism that is aware of the replication and the capacity variability without actively probing the servers' speeds. To achieve this, the selection of servers and dispatching of requests is based on the concept of tokens. Each token represents an admission ticket for a request to be processed at the corresponding server. TAD assigns tokens to incoming connections to process all requests therein. Tokens are a cheap mechanism to dynamically adapt the server load to its capacity, by implicitly limiting the arrival rate at each server by the token returning rate. In the following we describe the TAD-based arbiter implementation in detail.

4.1 Arbiter Implementation

The arbiter is initialized with one or more tokens per server, which are maintained in a central token pool. Fig. 4 illustrates the internal design of the TAD-based arbiter including the token pool. The arbiter listens for incoming connections, which are handled in a multi-threaded fashion. For every connection, the arbiter spawns a new pair of server selector and request handler threads and reroutes all related requests to it. Hence every server selector and request handler pair is dedicated to a specific connection, whereas the token pool is shared across all connections.

After the creation of a server selector and request handler pair, the server selector immediately starts scanning the token pool to acquire tokens of *fast* servers and hands them over to the request handler via a token queue. The request handler waits for requests to arrive, which it then clones and stores the replicas in a replica queue. As soon as both queues are not empty, the request handler retrieves a token-replica pair from the head of each queue, and dispatches the replica to the server specified by the token. Once a replica completes, the token is returned to the token queue, whereas after the connection tokens are returned to the token pool.

4.1.1 Server Selector

Motivated by the effectiveness of the power of many [15, 24] in reducing latency, we incorporate this idea when scanning for tokens. At any tier i the server selector acquires r_i tokens by performing $n_i{+}r_i$ token look-ups, i.e., n_i is the number of additional look-ups, so as to maximize the probability of finding the fastest r_i available servers. In a cloud setting these fastest servers may be those not currently being disturbed by neighbors. Moreover, the server selector skips tokens of the same server to avoid collisions between local replicas of the same request.

$n_i + r_i$ Token Look-ups The look-up process is greedy, meaning that it tries to scan as many as $n_i{+}r_i$ tokens to select at once the r_i fastest servers, where the speed of a server is defined by its last-observed latency. The choice of the additional number of look-ups n_i is empirically decided. Since the overhead of look-ups is low[2], we resort to the extreme case where $n_i{+}r_i$ is equal to M_i, the number of servers in the tier. The benefits of additional look-ups are easily illustrated through a simple probabilistic argument. Say that among the M servers, M_f are currently fast and M_s are slow. By performing $m_i = n_i{+}r_i$ lookups, the probability of finding at least one fast server is

$$\sum_{j=1}^{m_i} \frac{\binom{M_f}{j}\binom{M_s}{m_i-j}}{\binom{M}{m_i}} = 1 - \frac{\binom{M_f}{0}\binom{M_s}{m_i}}{\binom{M}{m_i}}, \tag{3}$$

which is non-decreasing in m_i. This can be shown by considering the terms on the right-hand side $K_{m_i} = \frac{\binom{M_f}{0}\binom{M_s}{m_i}}{\binom{M}{m_i}}$ with $m_i + 1$ lookups, which leads to

$$K_{m_i+1} = \frac{\binom{M_f}{0}\binom{M_s}{m_i+1}}{\binom{M}{m_i+1}} = \frac{\binom{M_f}{0}\binom{M_s}{m_i}\frac{M_s-m_i}{m_i+1}}{\binom{M}{m_i}\frac{M-m_i}{m_i+1}} = K_{m_i}\frac{M_s-m_i}{M-m_i}.$$

This shows that $K_{m_i+1} \leq K_{m_i}$ since the factor $\frac{M_s-m_i}{M-m_i} \leq 1$ as $M_s \leq M$. Thus the sequence $\{K_{m_i}\}_{m_i \geq 1}$ is non-increasing in m_i, making $\{1 - K_{m_i}\}_{m_i \geq 1}$ and (3) non-decreasing in m_i. As a result, increasing the number lookups $m_i = n_i + r_i$ increases the chances of finding at least one fast server.

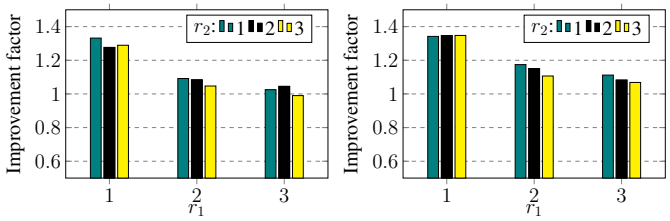
We thus expect the look-up scheme to be beneficial, particularly when there is a large degree of freedom in selecting fast servers. However, the benefit of this scheme can be limited in cases where the number of tokens in use is high, thus leaving only few tokens in the pool to choose from. This causes a trade-off between the delay introduced to wait for *fast* server tokens and the extra processing time of slow servers. As the latter is difficult to predict, we restrict the choice to the currently available tokens, say h, at the token pool. If $h{<}r_i$, the server selector makes use of these h tokens and waits for the remaining $r_i{-}h$ to return to the token pool.

We verify the performance benefit of the extra look-ups via a small empirical evaluation. To this purpose we use the MediaWiki experiment setup detailed out in Section 6 with TAD and interference at both tiers. Fig. 5(a)-(b) summarize the performance gains in terms of latency normalized by the case *without redundant look-ups* in sPARE, under different replication factors, and request rates of $\lambda_1{=}20$ and 5 page requests per second (pps). One can see that the performance gains of having redundant look-ups decrease with r_1, and shows minor differences across different values of r_2. Such a difference can be explained by the particular setup of the MediaWiki: it has a much larger number of Apache servers at tier-1 than DB servers at tier-2, i.e., $36{>}12$. Therefore, as TAD allocates tokens in batches, it has a higher chance to collect all tokens required by the redundant look-ups in tier 1 than in tier 2.

4.1.2 Request Handler

The request handler continuously replicates incoming requests and dispatches the replicas to the servers. Once the first replica completes, the response is sent back. The other replicas will be discarded but not canceled due to the non-negligible canceling

2. Even with a naive implementation the average overhead was below 20μs in all our testbed runs.



(a) $\lambda_1 = 20$ pps (b) $\lambda_1 = 5$ pps

Fig. 5: MediaWiki: the normalized improvement factor of look-ups, under replication factors (r_1, r_2) and page arrival rates.

overhead, which may easily nullify the effort. We refer readers interested in the exact impact of canceling overhead to [10, 25]. This is especially true in our example applications with msec latencies. However, the request handler tries to avoid unnecessary resource usage by not submitting replicas of an already completed request, e.g., when the first replica completes before all tokens for this connection are allocated.

Choice of C_i The performance of TAD depends very much on the number of tokens per server C_i. A low number of tokens can under-utilize the resources, e.g., limiting the number of concurrent executions, and potentially lead to long replica queueing times at the request handler waiting for tokens. To determine the number of tokens, we empirically experiment with different C_i, for different load conditions and replication factors. The rule of thumb practice here is that we choose a number of tokens that is able to maintain the system stable and bound the waiting time at the arbiter. A more in-depth discussion can be found in Section 5.1.2.

4.1.3 Performance Monitor

The workload monitor collects key statistics about replicas, requests and connections and feds them to the replicator. All the monitoring is done passively to minimize the overhead.More details on the statistics collected are found in Section 5.

4.2 Analytical Properties

Here we provide an analytical argument on the performance advantages of TAD in defending against high performance variability of servers and managing additional replication loads. TAD can regulate the traffic to each server according to its observed capacity, which varies with time, and best use the aggregate capacity of all servers at the same tier. As the following discussion applies for any tier, we drop the index i from the parameter M.

Let each of the M servers at the tier serve s requests per second with no interference. One can see s as the maximum sustainable throughput for a server, but such a value fluctuates over time depending on the interference of neighboring workloads. To illustrate how TAD overcomes the impact of interference we consider the *instantaneous* capacity and stability condition. To capture the impact of interference, we assume that in an interference period the capacity drops from s to γs with $0 < \gamma \leq 1$, and that servers experience interference a fraction of time f.

The main advantage of TAD is that it forwards requests to all servers, but it does so by submitting more requests to the fast servers. Consider for instance an example with two servers where one of the servers is suffering from the interference, thus their capacities are s and γs, and since TAD assigns requests to both servers according to their capacities, the instantaneous stability condition is $\lambda < s(1+\gamma)$, which means that the overall capacity is being used. Instead, if the incoming requests are evenly assigned to the servers, the instantaneous stability condition needs to hold for each server. For instance, with the RR policy the request arrivals are split evenly between the two servers, thus the arrival rate at *each* server must be less than its capacity, i.e., $\lambda/2 < \min\{s, \gamma s\} = \gamma s$, thus $\lambda < 2\gamma s$. This limit is clearly more restrictive than $s(1+\gamma)$ obtained with TAD.

Generalizing the above example to M servers, each undergoing a slowdown from its standard rate s to γs for a fraction f of the time, the capacity under TAD is

$$Ms((1-f) + f\gamma) = Ms(1 - f(1-\gamma)).$$

As long as the arriving request rate is less than this capacity, TAD can maintain the system stability. Instead, with RR the system is limited by the slowest server, which receives a fraction $1/M$ of the traffic, i.e., $\lambda/M < \gamma s$. Thus, we expect TAD to outperform load oblivious strategies, e.g., RR, especially in systems with high capacity variability and under moderate loads.

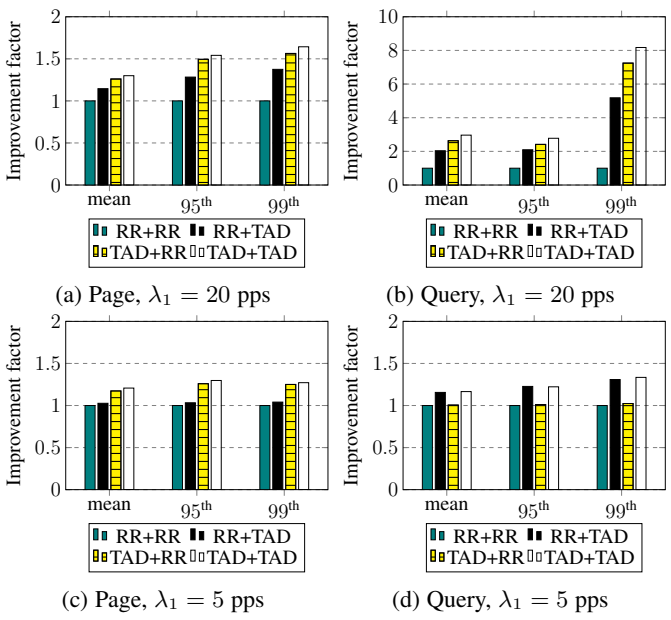
4.3 Empirical Comparison of TAD and RR

Here we empirically compare the proposed TAD against RR, using the running example of MediaWiki under an interference pattern where `iperf` is active at both tiers (see Section 6 for details). Particularly, we consider the four combinations of applying RR and TAD arbitration policies at both arbiters, namely, (i) RR+RR, (ii) RR+TAD, (iii) TAD+RR, and (iv) TAD+TAD, under any given replication factors (r_1, r_2), where $r_1, r_2 \in [1, 3]$, i.e., a total of 9 replication factor combinations. In order to ease the comparison of TAD with RR at different tiers, we use RR+RR as a normalization baseline for any given load and summarize in Fig. 6 the average improvement across the 9 combinations of replication factors. Fig. 6(a)-(b) present the page and query latency for the higher load case, i.e., arrival rate of λ_1=20 pps, and Fig. 6(c)-(d) show the lower load case, i.e., arrival rate of λ_1=5 pps. Overall, we observe that TAD on both tiers outperforms RR significantly, particularly regarding the query latency under λ_1=20. Here and in many other setups we have observed that TAD outperforms the simple RR policy under the mid-range of loads. As the load of λ_1=5 pps is well below the system capacity, the difference induced by a smart arbitration policy is less significant. Instead, for a load of 20 pps the introduction of TAD provides an improvement of almost 8x in query latency, as shown in Fig. 6(b). Here the two main features of TAD contribute to the its success, i.e, issuing the correct number of tokens, and performing n+r look-ups.

5 Replicator

The sPARE replicator determines the replication factors $(r_1, r_2, \ldots, r_N)$ for all N tiers, based on the observed latency variability, constraints on stability, as well as a predefined limit on the collision probability. As the value of r_i at each tier i is bounded by the number of servers M_i at the tier, the total number of replication factor combinations is thus given by the product $\prod_{i=1}^{N} M_i$, resulting in a large search space.

To perform a fast search for the optimal replication factors, the replicator first leverages two boundary conditions: i) a set of system stability conditions; ii) a limit on the collision probability at tier N. These conditions define a narrower search space, termed as the feasible set. Afterward, the replicator iteratively searches



(a) Page, $\lambda_1 = 20$ pps (b) Query, $\lambda_1 = 20$ pps

(c) Page, $\lambda_1 = 5$ pps (d) Query, $\lambda_1 = 5$ pps

Fig. 6: Page and query latency of MediaWiki: factor of improvement under `iperf` interference at both tiers, comparing RR+RR, RR+TAD, TAD+RR, TAD+TAD. The normalization base is the performance of RR+RR.

through potential replication factors based on the observed latency variability at each tier. Thus, the operation of the replicator consists of two steps: i) estimating the average tier capacity to define the stability conditions and determine the feasible set; ii) searching for the optimal replication decision within the feasible set. In particular, estimating the tier capacity requires considering both the server speed to process requests, as well as the availability of tokens when applying the TAD policy, as the number of tokens needs to be appropriately dimensioned to prevent them from becoming a bottleneck. The following sections detail these steps.

5.1 Estimating Average Tier Capacity

As we consider two types of resources, i.e., servers and tokens (when using TAD arbitration), we estimate two types of average *tier capacity*, namely *server processing capacity* and *token processing capacity*. The former is defined as the average number of requests that can be processed by *all* the *servers* at a tier per time unit, whereas the later defines the average number of connections that can be sustained by *all* the *tokens* at a tier per time unit. To make each tier stable, sPARE ensures that request arrival rates and connections arrival rates are less than the server and token processing capacity, respectively. We note that capacity estimation is long considered a challenging research topic [26], particularly for the case of concurrent execution.

5.1.1 Server Processing Capacity

To estimate the server processing capacity at tier i, we need to estimate the average processing speed per server hosted by a VM in the tier. As we consider multi-core VMs, we actually estimate μ_i, the *average* number of requests processed by *each core* per unit of time. Note that μ differs from s defined in Section 4.2, as μ is the maximum sustainable throughput for *each core*, whereas s is the same for *each VM*. Further, whereas s was the *nominal* capacity without any interference, here μ is the actual *observed* capacity, which is affected by any undergoing interference. We employ μ_i to derive the system stability conditions to determine the feasible replication levels at each tier. As each VM at tier i has V_i virtual cores, the total tier capacity, i.e., the average number of requests that can be processed by a tier, is the product of the processing speed μ_i of each core, the number of cores V_i per server, and the number of servers M_i in tier i, i.e., $\mu_i V_i M_i$.

To find μ_i, we focus on its inverse $T_i = 1/\mu_i$, which is the expected processing time of a request by a core at tier i. Due to the processor-sharing core operation, in the estimation of T_i we need to take into account that a request receives only a fraction of the processing capacity of a server during its execution, and this fraction depends on the number of requests executing concurrently. Thus to obtain T_i, and given that we want to keep the required monitoring overhead to a minimum, we rely on the simple Baseline (BL) estimation algorithm introduced in [27], which uses as input the request arrival and departure times at each server. The basic idea of the algorithm is to keep track of the number of concurrent requests and split the corresponding CPU time across the requests with special care to handle multi-core scheduling. Observations from a few hundred requests are typically enough to obtain a reliable estimate [27].

After obtaining μ_i, we can write the stability constraint on the server processing capacity as

$$\lambda_i r_i < M_i V_i \mu_i. \tag{4}$$

which ensures that the total *request* arrival rate $\lambda_i r_i$ at tier i is less than the total tier processing capacity. Recall that the arrival rate λ_i already includes the amplification caused by the replication at any tiers upstream of tier i.

5.1.2 Token Processing Capacity

In addition to the tier server processing capacity constraint in Eq. (4), the tokens introduced with TAD become a soft resource and their scarcity can limit the system capacity. We are thus interested in estimating the token processing rate μ_i^K, the inverse of which, $T_i^K = 1/\mu_i^K$, is the average time that a token is held by a connection. The token processing rate μ_i^K implicitly depends on the VM processing capacity in each tier as tokens co-share the VM capacity. However, whereas the core capacity μ_i is specific to a single tier, the token capacity μ_i^K depends on many tiers since the time a tier-i token remains busy includes effective execution time in tier i, tier $i+1$, and all the way down to tier N.

As each of the M_i servers in tier i issues C_i tokens, the token processing capacity at tier i is the product of the token processing rate μ_i^K and the total number of tokens $M_i C_i$, i.e., $M_i C_i \mu_i^K$. As with the server processing rate, we estimate the token processing rate via its inverse $T_i^K = 1/\mu_i^K$. Note that the processing time T_i^K per token is different from the effective processing time T_i of a request. The processing time of a *token* in any tier i includes not only the processing of all the requests in a connection in this tier, but also the time spent waiting for processing at any other downstream tiers. We can estimate the token processing time T_i^K by keeping track of the token allocation and release times. From a set of such observations we can obtain the mean T_i^K and the associated mean processing rate $\mu_i^K = 1/T_i^K$.

To ensure the token processing stability, the token demand rate should be less than the token processing capacity. As one token is used for each arbiter-server connection, the token demand is

essentially the product of the connection arrival rate λ_i^{conn}, and the replication factor at this tier r_i, thus

$$\lambda_i^{conn} r_i < M_i C_i \mu_i^K. \quad (5)$$

We note that the connection arrival rate is the request arrival rate divided by the average number of requests within each connection.

Now we can leverage this token stability constraint to determine the minimum number of tokens that can satisfy the token demands for any feasible replication factor. Particularly, assume r_i^{max} is the maximum replication factor at tier i that is considered feasible, i.e., it complies with constraints (4) and (5) and can be identified through off-line profiling. Thus the minimum number of tokens at tier i is

$$C_i > \frac{\lambda_i^{conn} r_i^{max}}{M_i \mu_i^K}. \quad (6)$$

Note that this limit is only a lower bound and we can set C_i to be the smallest integer that complies with this constraint.

5.2 Finding the Near-optimal Number of Replicas

To determine the optimal number of replicas we rely on three key observations: i) the introduction of replication is most helpful when the request processing times, and therefore the latency, are highly variable; ii) adding replicas is only feasible when the system has enough (token) capacity to process the additional load introduced, as established by the stability constrains (4) and (5); iii) the gains obtained with replication are more significant when the diversity of the resources is exploited by submitting replicas of any request to different servers in all tiers. Thus we introduce a limit on the collision probability, as defined in Eq. (1). Based on these observations, we define Alg. 1 to find the optimal number of replicas for a given target metric R^T, and a measure of the variability in each tier w_i. In this study we consider the tier 1 latency mean, 95^{th}, and 99^{th} percentiles as target metrics, while for the variability measure we adopt the ratio of a percentile, either the 95^{th} or the 99^{th}, to the mean latency. However, the search algorithm is flexible to consider other target and variability metrics, e.g., the latency variance.

The algorithm inputs are the estimated request arrival rate, connection arrival rate, tier capacity, token capacity, target and variability metrics. The first step in Alg. 1 is to determine the set $\mathcal{F}$ of feasible replication factors $\boldsymbol{r} = (r_1, r_2, \ldots, r_N)$. We consider $\boldsymbol{r}$ feasible if it complies with the stability constrains (4) and (5) at each tier, and the resulting collision probability (1) is at most equal to a pre-defined threshold $P_c^{\max} \leq 1$. These constraints make the set $\mathcal{F}$ relatively small, especially because the collision probability increases very fast with any r_i, and because the load in tier N increases with $\prod_{i=1}^{N} r_i$. Whereas our experiments focus on the two-tier case, with MediaWiki as our running example, Alg. 1 is stated for the general N-tier case, where the threshold $P_c^{\max}$ is applied to the collision probabilities $P_c^{1,j}$, for $j = 2, \ldots, N$, defined in Proposition 2.

The algorithm searches for the optimal $\boldsymbol{r}^*$ within the feasible set, $\mathcal{F}$, by iteratively (a) selecting a new feasible $\tilde{\boldsymbol{r}}$ in the neighborhood of the current one based on the measured latency variability w_i, and (b) testing the candidate $\tilde{\boldsymbol{r}}$ and measuring the achieved latency $\tilde{R}^T$. The initial $\boldsymbol{r}$ is with all replication factors set to 1, i.e., $\boldsymbol{r} = (1, \ldots, 1)$. For each $\boldsymbol{r}$ we run, we define a set of feasible directions, $\mathcal{I}$, which holds the indexes of the tiers i whose replication factor r_i can be increased such that the neighbor point $\hat{\boldsymbol{r}} = \boldsymbol{r}, \hat{r_i} = r_i + 1$ is still in $\mathcal{F}$.

Algorithm 1 Computing the optimal number of replicas. Target metric is R^T and variability metric is w_i.

Require: λ_i, μ_i, V_i, M_i, $P_c^{\max}$ and λ_i^{conn}, μ_i^K (if TAD is used), for $i \in [1, N]$

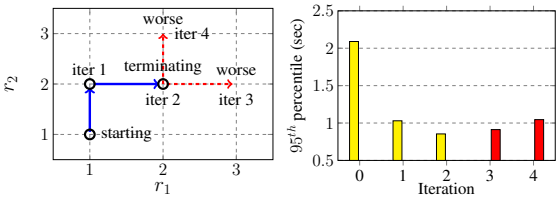
1: Determine set $\mathcal{F}$ based on (2), $P_c^{\max}$, (4) and (5)
2: $\boldsymbol{r} = (1, 1, \ldots, 1)$
3: $\mathcal{I} = \{i | \exists \hat{\boldsymbol{r}} = \boldsymbol{r}, \hat{r}_i = r_i + 1, \hat{\boldsymbol{r}} \in \mathcal{F}\}$
4: Measure R^T and w_i under $\boldsymbol{r}$
5: **while** $\mathcal{I} \neq \emptyset$ **do**
6: Choose tier $j = \arg\max_{i \in \mathcal{I}} \{w_i\}$
7: $\tilde{\boldsymbol{r}} = \boldsymbol{r}$, $\tilde{r}_j = r_j + 1$
8: Measure $\tilde{R}^T$ and $\tilde{w}_i$ under $\tilde{\boldsymbol{r}}$
9: **if** $\tilde{R}^T < R^T$ **then**
10: $\boldsymbol{r} = \tilde{\boldsymbol{r}}$
11: $R^T = \tilde{R}^T$, $w_i = \tilde{w}_i$ for $i \in [1, N]$
12: **else**
13: $\mathcal{F} = \mathcal{F} \setminus \{\tilde{\boldsymbol{r}} | \tilde{r}_j > r_j, \tilde{r}_i = r_i, i \neq j\}$
14: **end if**
15: $\mathcal{I} = \{i | \exists \hat{\boldsymbol{r}} = \boldsymbol{r}, \hat{r}_i = r_i + 1, \hat{\boldsymbol{r}} \in \mathcal{F}\}$
16: **end while**
17: **return** $\boldsymbol{r}$

From the set of feasible directions $\mathcal{I}$, the algorithm chooses the tier with the highest variability measure (line 6 in Alg. 1), and runs the associated neighbor point $\tilde{\boldsymbol{r}}$. Ties can be broken arbitrarily. If the target metric R^T is reduced under the candidate replication vector $\tilde{\boldsymbol{r}}$ compared to the values obtained with the current $\boldsymbol{r}$, we update $\boldsymbol{r}$ to this neighbor and continue the search process (lines 9-11 in Alg. 1). However, when no improvement is observed, we remove all points in this direction from set $\mathcal{F}$, including the point just evaluated (line 13 Alg. 1). In both cases, we update the set of feasible directions $\mathcal{I}$ and continue the search. The search algorithm stops when the set of feasible directions $\mathcal{I}$ becomes empty. Note that each iteration of the searching algorithm requires a sufficient amount of time to reach the steady state and obtain meaningful measurements of the response times. In future work we plan to explore more sophisticated models that lead to a starting point which is close to the optimal solution to expedite the search time.

5.2.1 *Reaching Near-optimal Replication Factors*

Here we illustrate how the iterative algorithm proposed in Alg. 1 searches for near-optimal replication factors on our running MediaWiki example. By fixing the threshold of the collision probability $P_c^{\max}$ to 0.5 and using the 95^{th} percentile response time at tier 1 (R_1^{95}) as target metric, we show in Fig. 7(a) the search trajectory, i.e., replication factors at each iteration, and in Fig. 7(b) the corresponding 95^{th} percentile. The starting point is always the no-replication case, i.e., (r_1, r_2)=$(1, 1)$. In the next step, we find the variability at tier 2 is larger than at tier 1, i.e., $w_2 > w_1$, thus we try the factor (1,2), and find it improves the 95^{th} percentile, $\tilde{R}_1^{95} < R_1^{95}$, thus we move to (1,2). From this point, we recompute and find that $w_1 > w_2$, thus the new factor to try is $(2, 2)$. Here again the new 95^{th} percentile is better, thus we move to (2,2). From this point, we try first (2,3), then (3,2), but both points lead to a larger 95^{th} percentile. Thus Alg. 1 terminates and returns (r_1^*, r_2^*)=$(2, 2)$ as the best replication factor found.

6 Evaluation



(a) Replication factor (b) Request latency

Fig. 7: Replication factor and corresponding latency for each iteration: pattern-2, $\lambda_1 = 20$ pps.

In this section, we evaluate the use of sPARE on distributed multi-tier applications, i.e., web serving and web searching, as a mechanism to defend latency under various performance variability patterns and load conditions in the cloud. Particularly, we deploy sPARE on MediaWiki [20] and Solr [21] in our private cloud testbed and focus on performance metrics of mean and high percentile latency, i.e., 95^{th} and 99^{th} percentiles. We also execute sPARE on Solr in a public Cloud (Amazon EC2) where we have no control of co-located workloads. We show that the latency gains achieved by sPARE can be truly significant, in comparison to a simple replication strategy that only replicates requests at the first tier and uses RR policy to arbitrate the replicated requests. To decide the optimal replication factor for the simple strategy, we resort to an exhaustive search through different values of r_1 and keep $r_2 = 1$, i.e., not replicating tier 2 requests. Moreover, we show that the replication factors determined by the sPARE replicator are nearly optimal for all scenarios considered here, as empirically proven by comparing against an exhaustive search.

6.1 Testbed

Our private cloud testbed is composed of eight identical physical servers, seven used to run the experiments and one used as experiment orchestrator and repository. Each server is equipped with 32 cores, 128 GB DDR4 RAM, six 1-TB solid state disks in RAID5, and two 10-Gigabit Ethernet adapters. Each component of MediaWiki and Solr is deployed on an individual VM equipped with 2 virtual cores and 4 GB of memory. The same holds for the three sPARE components, i.e., one replicator and two arbiters, but the arbiters are hosted on larger VMs, i.e., equipped with 8 cores, to ensure that they are not the bottleneck.

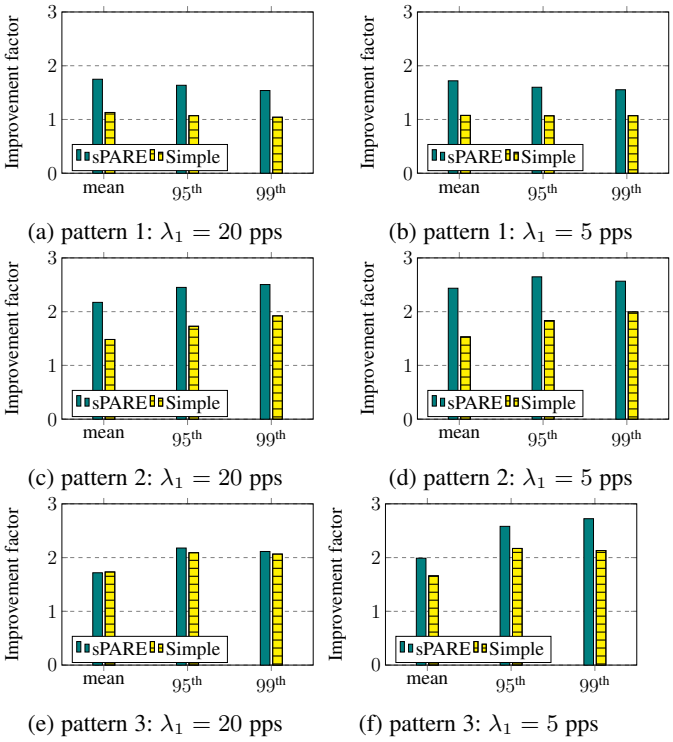
Neighboring Workload To emulate performance variability in the cloud, particularly the public cloud, we artificially spawn neighboring workloads following Poisson arrivals with mean inter-arrival time of 40 sec and exponential run times with mean of 10 sec. The specific neighboring workload used is `iperf` [22], emulating file transfers via the network. Particularly, we consider three types of interference patterns: (i) interference 1, `iperf` is active on tier 2; (ii) interference 2, `iperf` is active on both tiers; and (iii) interference 3, `iperf` is active on tier 1.

6.2 MediaWiki: Web Serving Application

MediaWiki is a latency-sensitive web application composed of Apache (v2.4.7) plus PHP (v5.5.9) as front-end application server , and MySQL (v5.5.40) as back-end DB server. Requests are generated with httperf [28], an open-loop workload generator. The MediaWiki cluster is composed of 36 front-end Apache VMs and 12 back-end DB VMs. We evaluate sPARE in this MediaWiki cluster under two request rates, i.e., λ_1=20 and 5 pages per second (pps), and all three interference patterns described earlier, for a total of six load scenarios. We configure sPARE with TAD in both tiers, where the number of tokens per server in tier 1 and tier 2 is 1 and 12, respectively, and the collision probability threshold is $P_c^{\max} = 0.5$. Before showing the latency improvement achieved by sPARE, we summarize the latency metrics of the original MediaWiki system in Table 2, as a baseline for comparison. As pattern 2 imposes a high variability on both tiers, we can see that the difference between the latency mean and 99^{th} percentile is larger than for the other two patterns.

TABLE 2: MediaWiki latency without replication strategy.

λ_1	pattern 1			pattern 2			pattern 3		
[pps]	mean	95^{th}	99^{th}	mean	95^{th}	99^{th}	mean	95^{th}	99^{th}
20	0.73	1.35	1.80	0.89	2.09	3.08	0.75	1.96	2.84
5	0.70	1.28	1.68	0.92	2.15	3.16	0.79	2.07	3.01



(a) pattern 1: $\lambda_1 = 20$ pps (b) pattern 1: $\lambda_1 = 5$ pps

(c) pattern 2: $\lambda_1 = 20$ pps (d) pattern 2: $\lambda_1 = 5$ pps

(e) pattern 3: $\lambda_1 = 20$ pps (f) pattern 3: $\lambda_1 = 5$ pps

Fig. 8: Factor of page latency improvement: comparing sPARE and simple-replication strategy under scenarios of two arrival rates and three interference patterns.

6.2.1 *Improvement in Latency Metrics*

In Fig. 8, we summarize the performance gains of sPARE and simple-replication over the original MediaWiki in terms of the normalized tier 1 page latency, for all the six scenarios considered. Here we set the target metric in Alg. 1 according to the metric of interest, i.e., the mean, 95^{th} and 99^{th} percentile of tier 1 response time and the ratio between the percentile of interest, or the 95^{th} if the target is the mean, and the mean.

Clearly, sPARE is able to achieve considerably better performance gains than simple-replication, with an improvement factor ranging between 1.5 and 3, depending on the metrics of interests and the interference patterns. There are two key observations. First of all, the higher the variability is, the higher performance

TABLE 3: Optimal MediaWiki replication factors reached by sPARE Replicator: optimal pairs (r_1^*, r_2^*), number of iterations, and normalized latency ratios against empirical optimal latencies.

Pattern	Target Metric	$\lambda_1 = 20$ pps			$\lambda_1 = 5$ pps		
		mean	95^{th}	99^{th}	mean	95^{th}	99^{th}
1	(r_1^*, r_2^*)	(1,3)	(1,3)	(1,2)	(2,3)	(2,3)	(2,3)
	Iterations	3	3	3	6	6	6
	Ratio	1.00	1.00	1.01	1.02	1.03	1.04
2	(r_1^*, r_2^*)	(2,2)	(2,2)	(2,2)	(3,2)	(3,2)	(2,3)
	Iterations	4	4	5	5	5	6
	Ratio	1.00	1.00	1.00	1.00	1.04	1.11
3	(r_1^*, r_2^*)	(2,1)	(2,1)	(2,1)	(3,2)	(3,2)	(2,3)
	Iterations	3	3	4	5	5	6
	Ratio	1.00	1.00	1.00	1.00	1.00	1.19

gains can be achieved by sPARE. When `iperf` interference occurs to both Apache and DB servers, i.e., pattern 2, sPARE can improve the page tail latency by a factor of 2.1x to 2.7x, whereas the performance gain of partial replication is less significant for weaker interference patterns, i.e., 1.5 and 2.7, where `iperf` only occurs at either Apache or DB tier. This observation resonates well with the original motivation of sPARE: defeat the performance disadvantage caused by the capacity variability and turn it into an advantage. Secondly, the power of partial replication is particularly significant for the tail latency, i.e., 95^{th} and 99^{th} percentiles, for most interference patterns and load conditions.

Let us zoom into the individual interference patterns. On the one hand, under inference pattern 1, one can see that sPARE can achieve a factor of 1.7x improvement for the page latency across all metrics considered. In contrast, simple-replication barely gains, compared to the no-replication MediaWiki system, because replicating only tier 1 requests does not really address the performance variability happening at tier 2. On the other hand, under a high degree of interference, i.e., pattern 2 where `iperf` is active on both tiers, sPARE still outperforms simple-replication, with an even bigger difference than pattern 1. The smaller gain in pattern 1 is attributed to the fact that executing all the DB queries in a page requires on average 127 msec, which is just 26% of the average page latency of 482 msec. Thus most of the page execution time occurs in tier 1, where no interference in present with pattern 1. Also, under interference pattern 3, simple-replication comes close to sPARE for the case with $\lambda_1 = 20$. However, when the baseline load is lower, i.e., $\lambda_1 = 5$, sPARE reduces all three latency metrics under all three interference patterns.

6.2.2 Reaching Optimal Replication Factors

We summarize the replication factors obtained with the sPARE replicator and the number of iterations required in Table 3. We also report the normalized latency ratio computed against the empirical optimal latency, which is obtained by exhaustive search through possible pairs (r_1, r_2). The value of one indicates that the replicator indeed reaches the optimal replication factors.

From the six scenarios and three metrics considered in Table 3, we see that the sPARE replicator successfully reaches the optimal replication factor that results in the minimum latency for most of the 18 combinations, as shown by normalized latency values equal to one. In some cases, when the base load is low, λ_1=5, the replicator tends to reach sub-optimal replication factors, particularly for the 99^{th} percentile. However, even in these cases the obtained latency is close to the optimal one.

We also observe that the replicator increases the replication levels for those tiers experiencing interference, when $\lambda_1 = 20$, i.e., increasing DB query redundancy under interference 1, increasing page redundancy under interference 3, and increase redundancy at both tiers under interference 2. However, when $\lambda_1 = 5$, the replicator increases the replication levels on both tiers for the three interference patterns, as the load is sufficiently low and provides extra room to accommodate additional replicas. In terms of the number of iterations required by the replicator, this is typically larger for $\lambda_1 = 5$ than for $\lambda_1 = 20$, due to a broader range of potential replication levels. Also, the number of iterations grows with the complexity of the inference patterns, i.e., it is higher for pattern 2 than for patterns 1 and 3.

6.3 SOLR: Web Searching

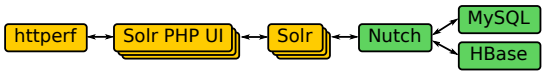


Fig. 9: Solr Architecture.

Our Web Searching use case is based on Nutch [29] and Solr [21]. Nutch crawls the web for documents to index, whereas Solr creates the search index and answers the queries. Fig. 9 shows the complete software setup, which can be split at the boundary between Solr and Nutch. This is highlighted in yellow and green colors. The green part is the batch workload part used to initialize the benchmark. It comprises Nutch (v2.1.2) plus a storage backend, e.g., HBase and MySQL among others. Here we settle for MySQL (v5.5.40). We initialize the benchmark by first using Nutch to crawl 50000 random URLs from the dmoz [30] repository and then sending the fetched documents to Solr to be indexed. This workload is not affected by sPARE.

The yellow part in Fig. 9 is the interactive part handling the user requests on which we apply sPARE. It comprises replicated instances of Solr-PHP-UI (v15.12.11) and Solr (v4.10.4). Solr-PHP-UI [31] offers a web-based user interface to access the query API of Solr. This part is similar to the MediaWiki setup, but Solr-PHP-UI and Solr communicate via HTTP, hence the tier 2 arbiter is an HTTP arbiter. The other notable difference is that each request at the web GUI forwards only one request to Solr.

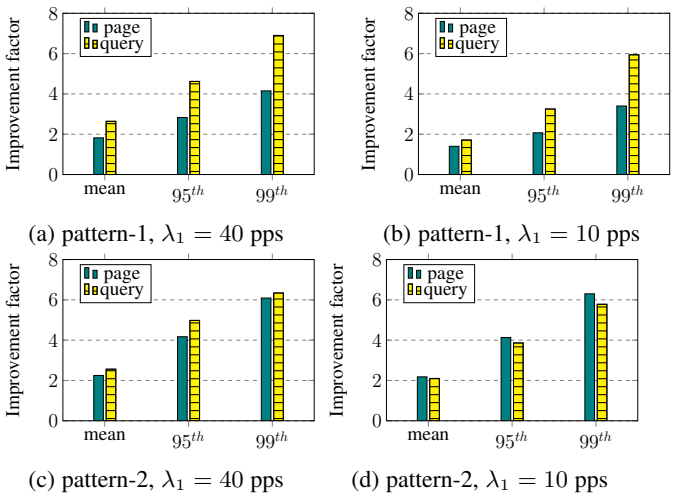
In the following, we evaluate the effectiveness of sPARE to reduce page and query latency for Solr. Here we use TAD on both tiers, with 1 and 3 tokens per server at tier 1 and tier 2, respectively, and collision probability threshold for the replicator $P_c^{\max} = 0.5$. We focus on scenarios with interference patterns 1 and 2, and with request arrival rates $\lambda_1 = 40$ and $\lambda_1 = 10$. In Table 4 we summarize the latency metrics obtained from the baseline Solr. Similarly to MediaWiki, the latency degrades with the degree of interference, particularly the 99^{th} percentile.

TABLE 4: Latency of Solr without replication strategy.

λ_1	pattern 1			pattern 2		
[pps]	mean	95^{th}	99^{th}	mean	95^{th}	99^{th}
40	28.6	56.6	97.8	35.3	87.6	173.7
10	23.8	45.7	86.5	35.9	90.8	167.0

6.3.1 Improvement in Latency

We summarize the factor of latency improvement in Fig. 10, comparing to the baseline latency for different scenarios. While



(a) pattern-1, $\lambda_1 = 40$ pps (b) pattern-1, $\lambda_1 = 10$ pps

(c) pattern-2, $\lambda_1 = 40$ pps (d) pattern-2, $\lambda_1 = 10$ pps

Fig. 10: Solr: Factor of page and query latency improvement of sPARE under two arrival rates and two interference patterns.

the improvement for the mean latency is comparable to Mediawiki, sPARE achieves a remarkable performance gain for the 99^{th} percentile, with factors ranging between 2x and 7x, for both page and query requests. The best improvement factor with sPARE is achieved for the 99^{th} percentile of page latency, under interference pattern 2, supporting the effectiveness of sPARE in defeating the long tail latency caused by high capacity variability. We also observe that sPARE is able to provide the largest gains on those tiers that suffer the largest variability. Thus, under interference pattern 1 the gains are more significant for the query latency, while under pattern 2 the gains are very similar for page and query latency. In addition, while the gain in mean is close to a factor of 2, the gains for the tail percentiles are much larger, with factors of up to 6x for both page and query 99^{th} latency percentile. The highest percentiles, which suffer the largest degradation due to the capacity variability, are the most benefited by the introduction of sPARE. Moreover, thanks to the TAD arbiter, the performance gains of both pages and queries are higher for the higher arrival rate, meaning that sPARE arbiters are able to efficiently leverage the spatial capacity variability across servers and sustain the additional loads introduced by replication. From the experiments with both MediaWiki and Solr, we observe that sPARE can effectively improve the performance of distributed multi-tier applications, particularly under high capacity interference, especially the tail latency of tier 2 requests.

6.3.2 Reaching Optimal Replication Factors

Table 5 summarizes the optimal replication factors determined by the replicator, the number of iterations, and the normalized latency against the empirical optimal one. As indicated by the normalized values, which are close to one for all load scenarios and metrics considered, the replicator successfully reaches near-optimal replication factors, in spite of being a greedy algorithm. Similar to the findings in MediaWiki, the replication factors are positively correlated with the location of the interferences, i.e., replication levels at tier 2 are higher under pattern 1 and both replication levels are high under pattern 2. However, there are a few exceptions, which actually do not achieve the empirical optimal values, as indicated by normalized latency ratios greater than one. Particularly, under pattern 2 and $\lambda = 40$ pps, sPARE does not

TABLE 5: Optimal Solr replication factors reached by sPARE Replicator: optimal pairs (r_1^*, r_2^*), number of iterations, and normalized latency ratios against empirical optimal latencies.

Pattern	Target Metric	$\lambda_1 = 40$ pps			$\lambda_1 = 10$ pps		
		mean	95^{th}	99^{th}	mean	95^{th}	99^{th}
1	(r_1^*, r_2^*)	(1,3)	(1,3)	(1,3)	(2,2)	(1,2)	(1,3)
	Iterations	3	3	3	5	3	3
	Ratio	1.01	1.00	1.00	1.00	1.00	1.00
2	(r_1^*, r_2^*)	(1,3)	(1,3)	(2,3)	(2,2)	(2,2)	(2,2)
	Iterations	3	3	5	4	4	4
	Ratio	1.07	1.04	1.15	1.00	1.00	1.00

TABLE 6: Latency of Solr without replication.

λ_1	Ireland			Oregon		
[pps]	mean	95^{th}	99^{th}	mean	95^{th}	99^{th}
5	26.13	34.48	94.26	24.21	34.24	39.67
1.25	26.68	33.72	60.06	25.98	35.59	40.31

reach the optimal mean, 95^{th} and 99^{th} latency percentiles. For the targets of mean and 95^{th} percentile latency, sPARE only increases the replication levels at the tier 2, though the interferences occur at both tiers. For the 99^{th} percentile, sPARE increases replication levels for both tiers drastically, resulting into higher replication factors than for the case with $\lambda = 10$ pps, i.e., $(2, 3)$ v.s. $(2, 2)$. This observation counters the intuition that optimal replication factors are lower for more loaded systems, because of lower free capacity for replication loads [10]. This is a result of the greedy steps in the algorithm which may result in it getting trapped at a sub-optimal solution. In spit of this, the largest optimality ratio observed here is 1.15, which implies that the latency obtained with sPARE is just 15% larger than the empirical optimal.

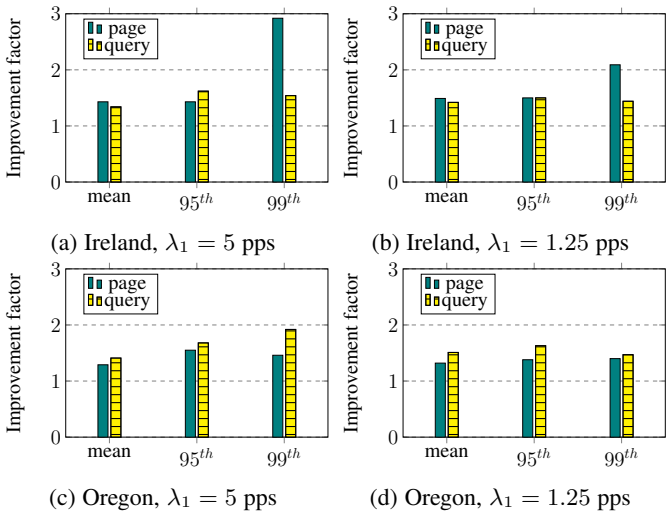
6.4 SOLR on EC2

Amazon EC2 Testbed We evaluate sPARE on Solr in the wild on Amazon EC2. The Solr cluster is composed of eight front-end Solr-PHP-UI VMs and eight back-end Solr VMs, each corresponding to Amazon t1.micro instances. The same holds true for the three sPARE components. These VMs are subject to the performance variability experienced in a public cloud deployment. We use TAD on both tiers with 1 token per server, $P_c^{\max} = 0.5$ as collision probability threshold for the replicator and request arrival rates λ_1=5 and 1.25.

Solr Baseline We summarize the latency metrics obtained from the baseline Solr deployed on Amazon EC2 in Table 6. Since on Amazon EC2 we have no control on the performance variability of the VMs, we report results from deployments in two different datacenters: Ireland and Oregon. The two datacenters achieve comparable mean latency values, however the performance variability in Ireland is higher than the one in Oregon as can be seen from the higher 99^{th} percentile values.

6.4.1 Improvement in Latency

We summarize the factor of latency improvement in Fig. 11, comparing to the baseline latency for different scenarios. Here sPARE achieves mostly improvements for the latency in the range from 1.3x to 1.6x, whereas the best improvement factor is 2.9x for the 99^{th} percentile of page latency. These values, although lower than the improvements on our private testbed, are quite significant as they are obtained in the wild, where we do not have



(a) Ireland, $\lambda_1 = 5$ pps (b) Ireland, $\lambda_1 = 1.25$ pps

(c) Oregon, $\lambda_1 = 5$ pps (d) Oregon, $\lambda_1 = 1.25$ pps

Fig. 11: Solr: Factor of page and query latency improvement of sPARE under scenarios of two arrival rates and two datacenters.

control on the performance variability as in our controlled private cloud. This means that they reflect the expected gains in real-world deployments, making the 2.9x gain even more impressive. The fact that the highest gain is achieved in the datacenter with the higher observed performance variability, i.e., Ireland as seen in Table 6, supports on the one hand the effectiveness of sPARE in defeating the long tail latency caused by high capacity variability. On the other hand, it underlines that the achievable gains depend on the inherent performance variability of the cloud infrastructure. The highest percentiles, which suffer the largest degradation due to capacity variability, are the most benefited by sPARE.

7 Related Work

Our work is related to prior art in the areas of (i) performance interference management in the cloud, (ii) speculative replication, (iii) tail latency optimization, and (iv) models for replication-enabled systems. As a detailed survey of the extensive related work is not possible here, we outline some particularly relevant work and highlight its relation to the solutions proposed in this work. We structure the related work in these four areas.

Performance interference in Cloud. Cloud computing offers many advantages, simplifying the deployment and management of web services and HPC applications, but also suffers from high performance variability due to its co-located nature. A plethora of related work addresses this pitfall, from characterization analysis [32] to solution strategies, which are either in the direction of scaling out/up provisioned resources in an economic way [33, 34, 35] or introducing replicated requests [17]. Cloud providers particularly offer cheaper instances at higher performance instability, such as AWS spot instances [36] and the prior art [35, 37] focuses on striking the optimal trade off between cost and performance variability. Wang et. al. [33] take the perspective of the cloud provider and demonstrate that combining pricing design and effective capacity modulation, i.e., the root cause of interference, can achieve better profit for the cloud providers and performance satisfaction for cloud users. Both Farley et. al. [2] and Björkqvist et. al. [34] proposed to intelligently and opportunistically choose VMs with better performance, whereas Subramanya et. al. [7] explore the pricing structure in the cloud to maintain performance in a cost-effective way. While increasing the resource redundancy only increases the probability that requests are served by a fast server in a coarse time granularity, ensuring fast server selection for individual requests still requires the assistance of a scheduling algorithm [1] that is aware of the interference.

Speculative replication. Speculatively replicating requests has been shown to be an effective strategy to strengthen system dependability [38] and to improve the response time [17], particularly the high percentiles. Most work on replication centers on a single tier for a wide range of applications, from conventional web services [9, 39] to recent big data platforms [11, 40, 15]. Replication policies in web and big data systems can be grossly classified by the issuing time of the replicated requests and by the canceling policy on the remaining redundant requests. Dolly and Grass [11, 12] advocate the efficacy of cloning all MapReduce tasks upon their arrival, a typical practice in speculative computing [41]. Wang et al. [42] design an efficient algorithm to reactively spawn speculation requests and further save computation capacity from serving unnecessary requests especially for systems that have much longer execution times than MapReduce applications. Chen et al. [40] extend the full cloning strategy to scenarios with jobs of different priority. Upon receiving the first result from replicated requests/jobs, the majority of replication policies leave the rest of replicas in the system due to the overhead of terminating requests, while a few studies show the benefits of terminating requests for certain benchmarks [9]. To best harvest the performance gains of request redundancy, Hopper [15] further develops a replication-aware scheduling algorithm for Spark.

Optimizing tail latency. As the tail latency significantly affects users' quality of experience, different resource management policies are designed to minimize the tail latency and exhibit disparate merits at different loads. For example, replication strategies [11, 12, 9] are effective for low loads, admission control [43] is particularly good for overload management, and scheduling policies [15, 1, 24, 44] are applicable to wider load ranges. Elastic resource scaling [45, 46, 47, 48], in terms of VMs or containers, is able to improve the overall latency in a cost effective manner when encountering dynamic loads, although it may not necessarily be effective on the tail latency [25]. The disadvantage of request replication is the resulting redundant load [17] that may destabilize the system. SNC Meister [43] controls the tail latency in datacenters by enforcing priorities and limiting rates in both storage and network. State of the art schedulers aim to minimize the tail latency by figuring out which servers are faster, as found in systems such as big data processing engines [24], in-memory datastores [44], and cloud systems [1]. Sparrow [24] advocates to reduce the task size of MapReduce jobs and power of many choices to minimize the impact of scheduling tasks to slow servers. C3 [44] leverages the number of queueing requests as an indicator for the speed of distributed memcached servers and schedules requests to servers with the shortest queue. An advantage of scheduling policies is that they can be combined with replication or admission control policies [15] and therefore are suitable for any load conditions.

Models for speculative replications. Motivated by the efficacy of replication strategies in real systems, various stochastic models aim to answer the fundamental question of the optimal number of replicas under various system assumptions, i.e., arrival processes, canceling policies, service processes, and performance metrics, i.e., average v.s. tail response times. A common scenario

is the automatic canceling of the remaining redundant replicas upon receiving the results from the fastest tasks [49] under Poisson arrivals and exponential service times. The average response time is a frequently used metric [17, 49, 16] to study the impact of additional replicas, except for Qiu et al. [10] who derive a tail latency model under the assumption of automatic replica canceling. Pérez et. al. [25] derive a closed form solution of the average response time for three different canceling policies, i.e., no cancel, cancel with delay overhead, and immediate canceling. Given the fact that replicas request the same content, the majority of modeling work still assumes independent service times, except for [50, 51, 52]. Qiu et. al. [52] model the distribution of response times for cloud-hosted web services that have Markovian arrival times via matrix analytic methods. The majority of analytical models are validated via simulations, except [52, 17] that provide empirical validation on real systems.

8 Concluding Remarks

To guarantee latency performance for multi-tier applications in the cloud, where high capacity variability is experienced, we propose a partial replication strategy, sPARE, which introduces workload redundancies according to the load and capacity variability observed in each tier. sPARE features a centralized replicator, which can attain near-optimal replication factors, and a distributed token-based arbiter, whose multi-threaded design and lightweight implementation effectively dispatches replicated requests to *fast* servers. Our extensive evaluation results, applying sPARE to multi-tier web serving and web searching applications, show that the proposed design and implementation of partial replication can greatly improve the latency, particularly its tail, under diverse neighboring interference patterns. In summary, sPARE significantly improves the latency for multi-tier applications in the cloud, turning the pitfall of capacity variability into a performance advantage.

Acknowledgment

This work has been partly funded by SNSF projects 407540_167266 and 200021_141002. The research of Juan F. Pérez has been supported in part by the ARC Centre of Excellence for Mathematical and Statistical Frontiers (ACEMS).

References

[1] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *NSDI*, pp. 329–342, 2013.

[2] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart *et al.*, "More for your money: exploiting performance heterogeneity in public clouds," in *SoCC*, p. 20, 2012.

[3] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *EuroSys*, pp. 237–250, 2010.

[4] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer *et al.*, "An analytical model for multi-tier internet services and its applications," in *SIGMETRICS*, pp. 291–302, 2005.

[5] P. Tembey, A. Gavrilovska, and K. Schwan, "Merlin: Application- and platform-aware resource allocation in consolidated server systems," in *SoCC*, pp. 14:1–14:14, 2014.

[6] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao *et al.*, "True elasticity in multi-tenant data-intensive compute clusters," in *SoCC*, p. 24, 2012.

[7] S. Subramanya, T. Guo, P. Sharma, D. E. Irwin *et al.*, "Spoton: a batch computing service for the spot market," in *SoCC*, pp. 329–341, 2015.

[8] S. Islam, S. Venugopal, and A. Liu, "Evaluating the impact of fine-scale burstiness on cloud elasticity," in *SoCC*, pp. 250–261, 2015.

[9] J. Dean and L. A. Barroso, "The tail at scale," *ACM Commun.*, vol. 56, no. 2, pp. 74–80, 2013.

[10] Z. Qiu and J. F. Pérez, "Evaluating the effectiveness of replication for tail-tolerance," in *CCGrid*, pp. 443–452, 2015.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *NSDI*, pp. 185–198, 2013.

[12] G. Ananthanarayanan, M. C. Hung, X. Ren, I. Stoica *et al.*, "GRASS: trimming stragglers in approximation analytics," in *NSDI*, pp. 289–302, 2014.

[13] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, "Virtualization in the private cloud: State of the practice," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 3, pp. 608–621, 2016.

[14] A. Rosà, L. Y. Chen, and W. Binder, "Failure analysis and prediction for big-data systems," *IEEE Trans. Serv. Comput.*, vol. PP, no. 99, pp. 1–1, 2017.

[15] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *SIGCOMM*, pp. 379–392, 2015.

[16] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" in *Allerton*, pp. 731–738, 2013.

[17] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry *et al.*, "Low latency via redundancy," in *CoNEXT*, pp. 283–294, 2013.

[18] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford *et al.*, "Profiling network performance for multi-tier data center applications," in *NSDI*, pp. 57–70, 2011.

[19] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[20] "Mediawiki," https://www.mediawiki.org.

[21] "Solr," http://lucene.apache.org/solr.

[22] "iperf," http://iperf.sourceforge.net.

[23] S. Gamage, R. R. Kompella, D. Xu, and A. Kangarlou, "Protocol responsibility offloading to improve TCP throughput in virtualized environments," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, p. 7, 2013.

[24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *SOSP*, pp. 69–84, 2013.

[25] J. F. Pérez, R. Birke, M. Björkqvist, and L. Y. Chen, "Dual scaling vms and queries: Cost-effective latency curtailment," in *ICDCS*, pp. 988–998, 2017.

[26] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *EuroSys*, pp. 31–44, 2007.

[27] J. F. Pérez, G. Casale, and S. Pacheco-Sanchez, "Estimating computational requirements in multi-threaded applications," *IEEE Trans. Softw. Eng.*, vol. 41, pp. 264–278, 2015.

[28] "httperf," http://www.hpl.hp.com/research/linux/httperf.

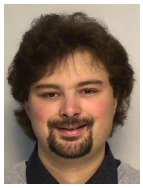
[29] "Nutch," http://nutch.apache.org.

[30] "dmoz," https://www.dmoz.org.

[31] "Solr-ui," http://www.opensemanticsearch.org/solr-php-ui.

[32] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing

variance," *VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.
[33] C. Wang, B. Urgaonkar, A. Gupta, L. Y. Chen *et al.*, "Effective capacity modulation as an explicit control knob for public cloud profitability," in *IEEE ICAC*, pp. 95–104, 2016.
[34] M. Björkqvist, L. Y. Chen, and W. Binder, "Opportunistic Service Provisioning in the Cloud," in *IEEE CLOUD*, pp. 237–244, 2012.
[35] J. Wen, L. Lu, G. Casale, and E. Smirni, "Less can be more: Micro-managing vms in amazon EC2," in *IEEE CLOUD*, pp. 317–324, 2015.
[36] "Amazon spot instances," https://aws.amazon.com/ec2/spot.
[37] P. Sharma, S. Lee, T. Guo, D. E. Irwin *et al.*, "Spotcheck: designing a derivative iaas cloud on the spot market," in *EuroSys*, pp. 16:1–16:15, 2015.
[38] S. Jain, M. J. Demmer, R. K. Patra, and K. R. Fall, "Using redundancy to cope with failures in a delay tolerant network," in *SIGCOM*, pp. 109–120, 2005.
[39] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao, "Improving web availability for clients with MONET," in *NSDI*, pp. 115–128, 2005.
[40] H. Xu and W. C. Lau, "Task-cloning algorithms in a mapreduce cluster with competitive performance bounds," in *ICDCS*, pp. 339–348, 2015.
[41] S. Melnik, A. Gubarev, J. J. Long, G. Romer *et al.*, "Dremel: Interactive analysis of web-scale datasets," *PVLDB*, vol. 3, no. 1, pp. 330–339, 2010.
[42] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 7–11, Nov. 2015.
[43] T. Zhu, D. S. Berger, and M. Harchol-Balter, "Snc-meister: Admitting more tenants with tail latency slos," in *SoCC*, pp. 374–387, 2016.
[44] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *NSDI*, pp. 513–527, Oakland, CA, 2015.
[45] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1378–1391, 2013.
[46] Z. Liu, A. Wierman, Y. Chen, B. Razon *et al.*, "Data center demand response: avoiding the coincident peak via workload shifting and local generation," in *SIGMETRICS*, pp. 341–342, 2013.
[47] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *SoCC*, p. 5, 2011.
[48] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *IEEE ICAC*, pp. 23–27, 2013.
[49] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter *et al.*, "Reducing latency via redundant requests: Exact analysis," in *SIGMETRICS*, pp. 347–360, 2015.
[50] G. Joshi, E. Soljanin, and G. W. Wornell, "Queues with redundancy: Latency-cost analysis," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, pp. 54–56, 2015.
[51] ——, "Efficient replication of queued tasks for latency reduction in cloud systems," in *Allerton*, pp. 107–114, 2015.
[52] Z. Qiu, J. F. Pérez, R. Birke, L. Y. Chen *et al.*, "Cutting latency tail: Analyzing and validating replication without canceling," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, 2017.

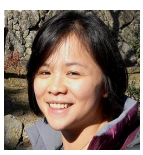


Robert Birke is at ABB Research Lab. He received his Ph.D. in Electronics and Communications Engineering from the Politecnico di Torino, Italy. His research interests are in the broad area of virtual resource management for large-scale datacenters, including network design, workload characterization and big-data application optimization.

Juan F. Pérez is an Assistant Professor at Universidad del Rosario, Colombia, Department of Applied Mathematics and Computer Science. He obtained a PhD in Computer Science from the University of Antwerp, Belgium, in 2010. He was a Research Associate at Imperial Collegeand a Research Fellow at the University of Melbourne.His interests center around the performance analysis of computer systems, especially cloud and cluster computing and optical networks.

Zhan Qiu is a PhD student in computer science at Imperial College London. She received a MSc in Computer Science from Imperial College London in Oct 2012. Her research interests include data-driven performance engineering and evaluation of computer and communication systems, parallel processing, performance optimization and resource provisioning.

Mathias Björkqvist is a software engineer at IBM Research – Zurich, Switzerland. He received a MSc. in Computer Science from the Helsinki University of Technology, Finland, in 2007, and a Ph.D. in Informatics from the University of Lugano, Switzerland, in 2015. His research interests include blockchain, encryption key management, security, storage, and resource and service provisioning in clouds.



Lydia Y. Chen is a research staff member at the IBM Zurich Research Lab, Zurich, Switzerland. She received a Ph.D. from the Pennsylvania State University. Her research interests include performance evaluation for datacenters and big data systems. She has served on several technical program committees in various performance and network conferences.She is a IEEE senior member.