Power of Redundancy: Designing Partial Replication for Multi-tier Applications

Robert Birke*, Juan F. Pérez†, Zhan Qiu‡, Mathias Björkqvist*, and Lydia Y. Chen*
*IBM Research Zurich, Rüschlikon, Switzerland. Email: {bir,mbj,yic}@zurich.ibm.com
†Universidad del Rosario, Colombia. Email: juanferna.perez@urosario.edu.co
‡Imperial College, London, United Kingdom. Email: zhan.qiu11@imperial.ac.uk

***Abstract*—Replicating redundant requests has been shown to be an effective mechanism to defend application performance from high capacity variability – the common pitfall in the cloud. 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, where interference from neighboring workloads is entangled with complex tier interdependency. In this paper, we design a first of its kind PArtial REplication system, sPARE, that replicates and dispatches read-only workloads for multi-tier web applications, determining replication factors per tier. 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 web searching applications, i.e., MediaWiki and Solr, deployed on our private cloud testbed. 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.**

I. Introduction

Performance variability is considered as one of the major pitfalls in the cloud computing paradigm [27, 12], because the virtualization technology does not guarantee performance isolation [15]. Applications hosted in the cloud thus are subject to interference from unknown neighboring workloads [24, 10]. 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 [23], 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). Nonetheless, the amount of resources required to fulfill SLAs in a cloud environment with highly varying VM capacities, can grow rapidly [26], and thus hinder its applicability for highly distributed web applications.

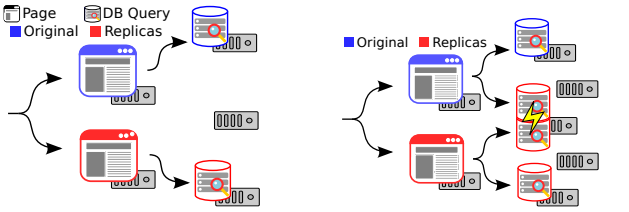
Replicating redundant requests is an inexpensive and yet effective alternative to improve the tail latency of web applications [11, 17] and to mitigate the effect of stragglers in big data applications [6, 7], particularly addressing the issues of performance variability among computing units. Redundant requests are issued either right upon their arrival [6, 11] or after detecting slow servers [21], 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 [22, 25]. Furthermore, the variability of the processing times [9] has been shown to play a central role in the actual impact of replication on the latency tail [25, 17]. The main challenges of developing replication strategies are thus to find optimal request redundancy levels based on the observed variability, as well as 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 [25], or map-reduce-like applications, e.g., SPARK [21]. 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 any communication overhead among tiers [28].

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-only workloads. 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, which may be caused by neighboring interferences. The aim of sPARE is to use redundant requests to increase their chances of being processed at fast servers, which is achieved by deciding the optimal number of replicas at all tiers and implementing smart dispatching policies. To this end, the two key features of sPARE are: i) a centralized replicator that coordinates the 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).

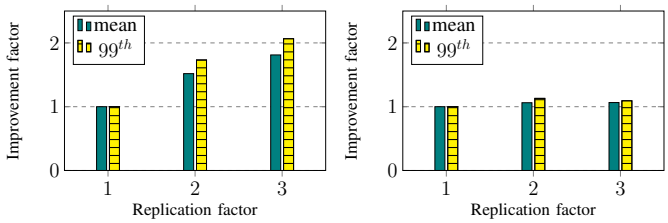
We test sPARE on multi-tier web serving and web searching applications, namely MediaWiki [3] and Solr [5], deployed at our private cloud testbed. Our extensive evaluation results for different combinations of interference patterns and loads show that sPARE is able to improve the latency, particularly

[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.



(a) `iperf` at Apache VMs (b) `iperf` at DB VMs

Fig. 2: Normalized latency of MediaWiki application under different interference patterns and replication factors.

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.

II. 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 [3] – the open source platform for Wikipedia, as an example, though it applies in general for multi-tier applications. Fig. 1 gives a high level overview of MediaWiki and its main components: multiple front-end Apache servers and multiple back-end database (DB) servers. Additionally, both front-end and back-end have dispatchers in front of the corresponding servers. Fig. 1 gives a high level overview. The performance metrics of interest are the mean and tail latency, e.g., 99^{th} percentile, to retrieve a complete page of Wikipedia. The detailed specification of MediaWiki and its setup can be found in Section VI.

Next, we introduce interference at the Apache servers, collocating `iperf` [2] to create random network transfers between pairs of VMs, causing both CPU and network contention. Fig. 2(a) shows the performance gains obtained with replication, measured as the ratio between the latency metric (mean and the 99^{th} percentile) without replication to the same metric with replication. Clearly, replicating Wiki pages can improve the mean and 99^{th} latencies by a factor of 1.5-2.0x. Instead, Fig. 2(b) shows the same metrics for the case where we introduce interference at the DB servers only, where the factor of improvement is barely 1.1x. 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 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.

III. 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 and continue using MediaWiki as an example. We note that while a large body of related work [7, 21] 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. In the following, we first detail out the design features of sPARE and conclude with the analysis of the collision probability at tier 2.

A. 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 server at tier i.

B. 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 (r_1^*, r_2^*) within a set of boundary conditions that ensure the system stability and

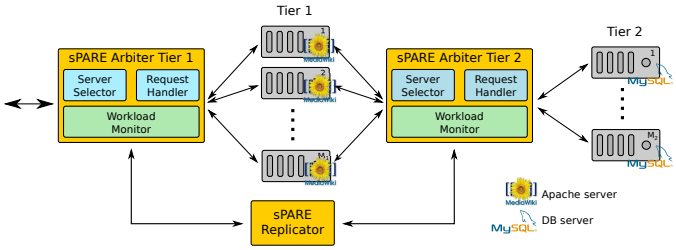


Fig. 3: Example architecture of sPARE with MediaWiki.

pose a bound on the collision probability, as described in detail in Section V.

C. 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.

Finally, sPARE supports multiple protocols via specific arbiters. This allows for protocol specific optimizations, as well as easy protocol extension via additional arbiters. We have developed arbiters for the HTTP and MySQL protocols. The details of the protocol-independent characteristics of the TAD arbiter and the replicator are explained in Sections IV and V, respectively.

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.

D. 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

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

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.

E. 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 notice 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

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

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 as seen from the first tier, the tier-N non-collision probability is given by (2) with $i{=}1$ and $j{=}N$.

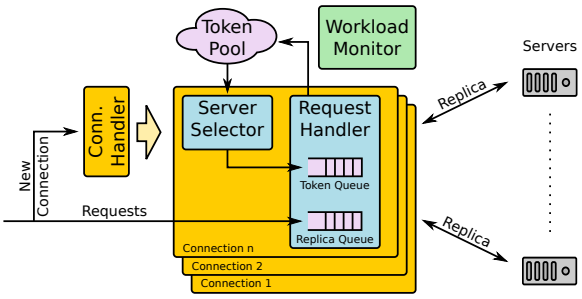


Fig. 4: Arbiter design with TAD.

IV. 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.

A. 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 at the end of the connection tokens are returned to the token pool.

1) Server Selector: Motivated by the effectiveness of the power of many [21] 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.

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 overhead, which may easily nullify the effort. 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 V-A2.

3) Performance Monitor: The workload monitor collects key statistics about replicas, requests and connections, to be fed to the replicator. All the monitoring is done passively to keep the impact on the system to a minimum. More details on the specific statistics collected are found in Section V.

V. 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 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, which is necessary to define the stability conditions, to 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.

A. 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, particularly for the case of concurrent execution.

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. 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 [16], 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 [16].

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

$$\lambda_i r_i < M_i V_i \mu_i. \tag{3}$$

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.

2) Token Processing Capacity: In addition to the tier server processing capacity constraint in Eq. (3), 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. 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 $C_i M_i$, 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 request 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 $\mu_i^K = 1/T_i^K$ of each token.

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

$$\lambda_i^{conn} r_i < M_i C_i \mu_i^K. \tag{4}$$

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 (3) and (4) and can be identified through off-line profiling. Thus the minimum number of tokens at tier i is

$$C_i > \frac{\lambda_i^{conn} r_i^{max}}{M_i \mu_i^K}. \tag{5}$$

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.

B. Finding the 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 (3) and (4); 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 mean, 95^{th}, and 99^{th} percentiles of tier 1 latency 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 (3) and (4) 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 $j = 2, \ldots, N$, shown in Section III-D

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 potential $\tilde{\boldsymbol{r}}$ 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 $\mathcal{F}$.

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 compared to the values obtained with the current $\tilde{\boldsymbol{r}}$, 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.

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 (1), $P_c^{\max}$, (3) and (4)
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** $\boldsymbol{r}$

VI. Evaluation

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, load conditions and cloud setups. Particularly, we deploy sPARE on MediaWiki [3] and Solr [5] in a controlled environment – our private cloud testbed, where we can control the neighboring effect. We focus on the mean and high percentile latency, i.e., 95^{th} and 99^{th} percentiles, as performance metrics. In the following, we present the results of MediaWiki and Solr on our private cloud testbed

A. 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. Three components of sPARE, i.e., one replicator and two arbiter, are deployed are individual VMs.

Fig. 5: MediaWiki Architecture

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` [2], 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

B. 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. We generate random wiki page requests with httperf [1], an open-loop workload generator. Fig. 5 summarizes the software chain. We evaluate sPARE in this MediaWiki cluster under two request rates, i.e., λ_1=20 and 5 page requests per second,

TABLE I: MediaWiki latency [sec] without replication.

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

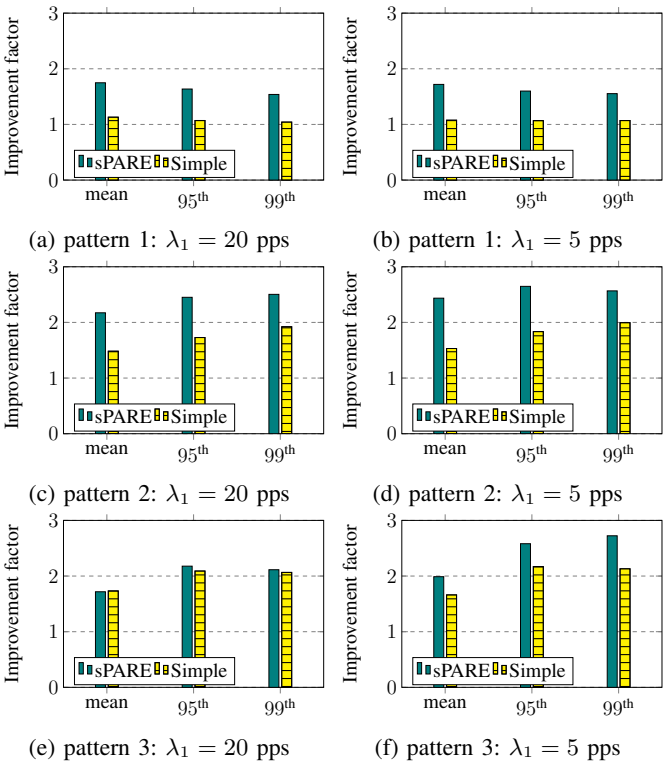
and all three interference patterns, for a total of six load scenarios. We configure sPARE with TAD in both tiers, where the number of tokens per server in tier 1 and tier 2 is 1 and 12, respectively, and the collision probability threshold is $P_c^{\max} = 0.5$.

Wiki Baseline Before showing the latency improvement achieved by sPARE, we first summarize the latency metrics of the original MediaWiki system in Table I, as comparison baseline. 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.

1) Improvement in Latency Metrics: Fig. 6 summarizes 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. For the sPARE replicator, in Alg. 1 we set the target metric according to the metric of interest, i.e., the mean, 95^{th} and 99^{th} percentile of tier 1 response time, and the variability metric as 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 a factor ranging between 1.5 and 3, depending on the metrics of interests and the interference patterns. There are two key observations. Firstly, the higher the variability is, the higher the performance gains that can be achieved by sPARE. When `iperf` interference occurs at 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.

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 WikiMedia system, because replicating only tier 1 requests does not really address the performance variability happening at tier 2. On the other hand, under pattern 2 where `iperf` is active on both tiers, sPARE still outperforms simple-replication, with a 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



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

in present with pattern 1. Also, under interference pattern 3, simple-replication comes close to sPARE for the case with λ_1=20. However, when the baseline load is low, i.e., λ_1=5, sPARE reduces all three latency metrics under all interference patterns.

We further compare the replication factors and latency of sPARE, against the empirical optimal values that are found through exhaustive search. Due to the space limit, we skip the presentation of detailed statistics and only report the final values here. For $\lambda_1 = 2$ pps, sPARE attains the empirical optimal mean, 95^{th}, and 99^{th} for all patterns. For $\lambda_2 = 5$ pps, sPARE achieve latency that 4 % higher the empirical optimal for most of combinations of patterns and metrics, except 99^{th} at the third pattern.

C. SOLR: Web Searching

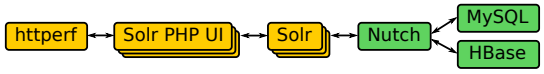


Fig. 7: Solr Architecture.

Our Web Searching use case is based on Nutch [4] and Solr [5]. Nutch crawls the web for documents to index, whereas Solr creates the search index and answers the queries. Fig. 7 shows the complete software chain which can be split

at the boundary between Solr and Nutch. This is highlighted in yellow and green colors. The green part operates as a batch workload used to create and update the search index. It comprises Nutch (v2.1.2) plus a storage backend, e.g., HBase or 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 repository and then sending the fetched documents to Solr to be indexed. This workload is performed only once before running our tests and is not affected by sPARE.

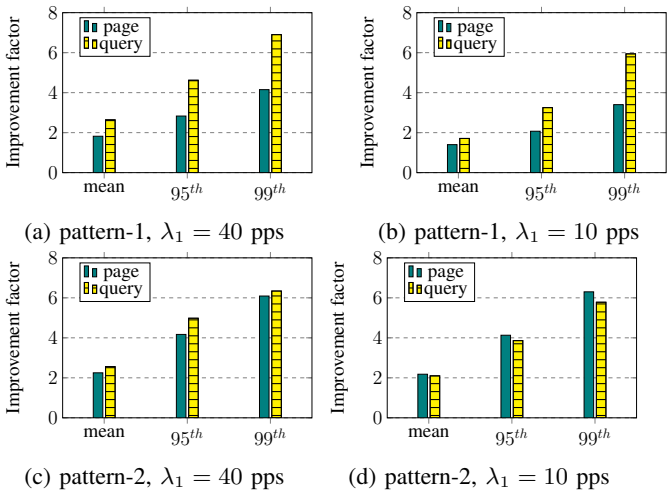
We apply sPARE on the yellow part in Fig. 7, which handles the interactive user requests. It comprises replicated instances of Solr-PHP-UI (v15.12.11) and Solr (v4.10.4). Solr-PHP-UI offers a web-based user interface to access the query API of Solr. Search requests are generated with httperf [1]. This 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 mostly forwards only one request to the Solr back-end.

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 $P_c^{\max} = 0.5$ as collison probability threshold for the replicator. We focus on scenarios with interference patterns 1 and 2, and with request arrival rates $\lambda_1 = 40$ and $\lambda_1 = 10$. Before evaluating the effectiveness of sPARE to reduce page and query latency for Solr, we first summarize the latency metrics obtained from the baseline Solr in Table II.

TABLE II: Latency [msec] of Solr without applying 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

1) Improvement in Latency: We summarize the factor of latency improvement in Fig. 8, comparing to the baseline latency for different scenarios. While 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, 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



(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. 8: Solr: Factor of page and query latency improvement of sPARE under scenarios of two arrival rates and two interference patterns.

the TAD arbiter, the performance gains of either page 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 a high replication loads. Combining observations from MediaWiki, we summarize that sPARE can effectively improve the performance of distribution multi-tier applications, particularly under higher capacity interference, for tail latency of the tier 2 requests.

VII. Related Work

Speculatively replicating requests has been shown to be an effective strategy to strengthen the system dependability [13] and to improve the response time [25, 20, 19], particularly the high percentiles. Most work on replication centers on a single tier for a wide range of applications, from conventional web services [11, 8], to recent big data platforms [6, 21]. 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 [6, 7] advocate the efficacy of cloning all MapReduce tasks upon their arrivals, instead of spawning the replicated tasks after receiving inferior performance indication – a typical practice in speculative computing [14]. 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 small number of studies show the benefits of terminating requests for certain benchmarks [11, 18]. To best harvest the performance gain brought by request redundancy, Hopper [21] further develops a replication-aware scheduling algorithm for Spark.

In contrast to the related work, sPARE explores request redundancy in the scenario of *multi-tier* applications hosted in the cloud, where each tier faces highly varying capacity and load dynamics are tightly interdependent across tiers. sPARE

combines the strategies of tier-specific optimal replication and fine-grained arbitration, exploiting the latency variability across servers and tiers.

VIII. 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 is able to significantly improve 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 by the ARC Centre of Excellence for Mathematical and Statistical Frontiers (ACEMS).

References

[1] httperf. http://www.hpl.hp.com/research/linux/httperf.

[2] iperf. http://iperf.sourceforge.net.

[3] Mediawiki. https://www.mediawiki.org.

[4] Nutch. http://nutch.apache.org.

[5] Solr. http://lucene.apache.org/solr.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, pages 185–198, 2013.

[7] G. Ananthanarayanan, M. C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: trimming stragglers in approximation analytics. In *NSDI*, pages 289–302, 2014.

[8] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao. Improving web availability for clients with MONET. In *NSDI*, pages 115–128, 2005.

[9] M. Björkqvist, L. Y. Chen, and W. Binder. Opportunistic Service Provisioning in the Cloud. In *IEEE CLOUD*, pages 237–244, 2012.

[10] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder. Achieving application-centric performance targets via consolidation on multicores: myth or reality? In *HPDC*, pages 37–48, 2012.

[11] J. Dean and L. A. Barroso. The tail at scale. *ACM Commun.*, 56(2):74–80, 2013.

[12] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *SoCC*, page 20, 2012.

[13] 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*, pages 109–120, 2005.

[14] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[15] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *EuroSys*, pages 237–250, 2010.

[16] J. F. Pérez, G. Casale, and S. Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE TSE*, 41:264–278, 2015.

[17] Z. Qiu and J. F. Pérez. Evaluating the effectiveness of replication for tail-tolerance. In *CCGrid*, pages 443–452, 2015.

[18] Z. Qiu and J. F. Pérez. Evaluating replication for parallel jobs: An efficient approach. *IEEE Trans. Parallel Distrib. Syst.*, 27(8):2288–2302, 2016.

[19] Z. Qiu, J. F. Pérez, R. Birke, and L. Y. C. P. G. Harrison. Cutting latency tails via replication: Analysis and experimental validation. Technical Report RZ 3908, IBM Research, 2017.

[20] Z. Qiu, J. F. Pérez, and P. G. Harrison. Variability-aware request replication for latency curtailment. In *INFOCOM*, pages 1–9, 2016.

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

[22] N. B. Shah, K. Lee, and K. Ramchandran. When do redundant requests reduce latency? In *Allerton*, pages 731–738, 2013.

[23] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi. An analytical model for multi-tier internet services and its applications. In *Sigmetrics*, pages 291–302, 2005.

[24] J. Vallone, R. Birke, L. Y. Chen, and B. Falsafi. Contention detection by throttling: A black-box on-line approach. In *IWQoS*, pages 237–242, 2015.

[25] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CoNEXT*, pages 283–294, 2013.

[26] C. Wang, B. Urgaonkar, A. Gupta, L. Y. Chen, R. Birke, and G. Kesidis. Effective capacity modulation as an explicit control knob for public cloud profitability. In *IEEE ICAC*, pages 95–104, 2016.

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

[28] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, pages 57–70, 2011.