SmallTail: Scaling Cores and Probabilistic Cloning Requests for Web Systems

Ewnetu Bayuh Lakew*, Robert Birke†, Juan F. Pérez‡, Erik Elmroth* and, Lydia Y. Chen§

*Umeå University, Umeå, Sweden. Email: {ewnetu,elmroth}@cs.umu.se

†ABB Research, Baden-Dättwil, Switzerland. Email: robert.birke@ch.abb.com

‡Universidad del Rosario, Bogotá, Colombia. Email: juanferna.perez@urosario.edu.co

§IBM Research Zurich, Rüschlikon, Switzerland. Email: yic@zurich.ibm.com

***Abstract*—Users quality of experience on web systems are largely determined by the tail latency, e.g., 95th percentile. Scaling resources along, e.g., the number of virtual cores per VM, is shown to be effective to meet the average latency but falls short in taming the latency tail in the cloud where the performance variability is higher. The prior art shows the prominence of increasing the request redundancy to curtail the latency either in the off-line setting or without scaling-in cores of virtual machines. In this paper, we propose an opportunistic scaler, termed SmallTail, which aims to achieve stringent targets of tail latency while provisioning a minimum amount of resources and keeping them well utilized. Against dynamic workloads, SmallTail simultaneously adjusts the core provisioning per VM and probabilistically replicates requests so as to achieve the tail latency target. The core of SmallTail is a two level controller, where the outer loops controls the core provision per distributed VMs and the inner loop controls the clones in a finer granularity. We also provide theoretical analysis on the steady-state latency for a given probabilistic replication that clones one out of N arriving requests. We extensively evaluate SmallTail on three different web systems, namely web commerce, web searching, and web bulletin board. Our testbed results show that SmallTail can ensure the 95th latency below 1000 ms using up to 53% less cores compared to the strategy of constant cloning, whereas scaling-core only solution exceeds the latency target by up to 70%.**

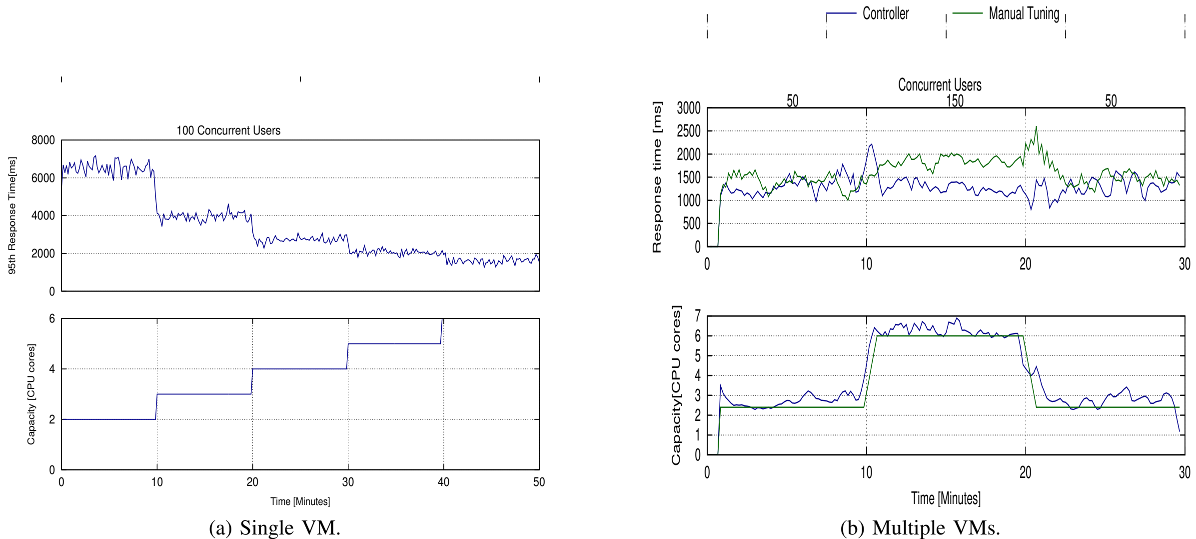
I. Introduction

On-line interactive services have become indispensable for today's business and private users. It is of paramount importance for on-line services to ensure that the latency is satisfactory at all times. Delay in the latency can cause not only significant financial losses but also degrade user quality of experience. Back in 2008, Amazon estimated [1] that a latency delay of 100 millisecond can cause 1% drop of its sales. A recent study from Akamai [5] shows that one second delay in page response can result in 7% loss in conversions for e-commerce. Meanwhile, user's quality of experience is often decided by tail latency values, e.g., 95th percentile, instead of average latency values.

The long standing challenge to manage the latency for on-line services is the workload variability, i.e., users requests exhibit a high time variability. In the era of cloud computing, such an issue is largely addressed by vertical or horizontal resource scaling, i.e., adjusting the number of either virtual cores or virtual machines respectively [27], [43], [32]. On the one hand, scaling Virtual Machines (VMs) is more suitable for stateless services that do not require closed synchronization across VMs and where the time overhead to spawn a VM on demand is in the order of minutes. Pointed by numerous studies, simply provisioning a higher number of VMs may not be a cost-effective solution to guarantee, particularly, the tail latency [14], [18] because of the high performance variability in the Cloud [51], stemming from the resource contention with co-located users coupled with a high sensitivity to small variations in the system. On the other hand, scaling virtual cores is more suitable for stateful services, e.g., back-end DBs, and incurs lower provisioning overhead, which conveniently renders itself for fine-grained control against the highly volatile workload and large-scale stateful database applications [21]. Moreover, allocating more cores to a single VM increases the opportunities to obtain physical resources, against co-located VM.

Recently, it is shown to be effective to control the tail latency by introducing workload redundancy [16], i.e., proactively cloning arriving requests by an integer factor, sending to different VMs, and returning only the fastest request. Most interactive systems then leave the rest of redundant requests to complete due to the complexity of implementing cancellation policies and thus result into a significant load increase [48]. For instance, having a cloning factor of two easily doubles the system load. Consequently, workload redundancy is deemed expensive and it is an utter challenge to determine the optimal cloning level that strikes the optimal trade-off between the latency gain and additional processing overhead. Moreover, an implicit underlying assumption of cloning requests is that a sufficient number of VMs exist, strongly arguing for co-optimization of virtual resources and request clones in curtailing latency.

However, few prior art simultaneously explore resource, i.e., only VM, and workload redundancy, either focusing on off-line setting in real testbed [35] or evaluated on-line solution in a simulation environment [38], [37] where the whole incoming requests are cloned by an integral factor. It largely remains unknown how scaling virtual cores and partial request cloning can maintain the tail latency target for interactive web service hosted on the Cloud. More importantly, it is crucial for service providers to minimize the additional cost, e.g., the total number of provisioned core, that occurs to prevent the loss associated with latency violation.



(a) Single VM.

(b) Multiple VMs.

Fig. 1: The tail latency under (a) single VM and (b) mutliple VMs. The former has constant arrival rate of 100 requests per second and the number of cores is steadily increased. The latter has dynamic arrival rates and the number of cores is adjusted by two approaches: the virtual controller, and manual-tuning (an off-line optimal solution).

In this paper, we address the following research questions: (i) how to simultaneously provision virtual cores to VMs and clone requests to achieve the target tail latency at a minimum provisioning cost, (ii) what is the optimal cloning factor so as to harvest the latency gain without excessive processing overhead of redundant requests. To such an end, we propose a novel probabilistic cloning strategy, which only allows to clone a fraction of the total requests to limit the cloning overhead, i.e., only portion of N arriving requests (called *clone factor*) which are deemed enough to meet the tail latency is cloned. The clone factor is periodically updated to dynamically adapt to changes in the system. We derive the theoretical latency analysis on the probabilistic cloning in steady state. We also develop a two-layer on-line controller, termed SmallTail, whose outer control loop adjusts the virtual core provisioning in a distributed fashion and whose inner loop decides the probabilistic clone factors in a finer time granularity. We extensively evaluate SmallTail on three web services, namely RUBiS [7], RUBBoS [6], and SOLR [2], and four workload traces. Our results show that SmallTail can effectively adhere to meet the stringent tail latency target and consume a lower number of resources and power consumption, compared to the strategy that combines core scaling and integer cloning.

Our contributions are analytical and practical. We propose a novel probabilistic cloning strategy that allows a fraction of requests to be cloned. Its effectiveness is rigorously proven by the latency models. We develop a first of its kind on-line controller that combines both virtual core allocation and request cloning in fine granularity. Our controller is extensively evaluated on multiple applications and workload patterns, against alternatives that do not leverage the feature of dynamic cloning.

II. Motivation

Prior to introducing SmallTail, we first present a small scale motivation study to illustrate the effectiveness of scaling virtual cores only in mitigating the tail latency. We conduct two types of experiments, one with single VMs and one with multiple VMs.

Single VM. We deploy RUBiS [7], a web shopping benchmark, on a single VM in our private cloud. The testbed is detailed in Section VI. We generate a constant arrival rate, i.e., 100 requests per second, for a duration of 50 minutes. We increase the number of cores in a step-wise manner, as shown in the top plot of Fig. 1a. The top plot in the figure depicts the 95th percentile (the tail latency) while the bottom plot depicts the allocated cores. One can clearly see that the tail latency decreases with the increasing number of virtual cores, but with decaying marginal gain. The latency improvement between 5 and 6 virtual cores is very minimum, compared to the difference between 2 and 3 cores. If today's latency target for the 95th percentile is 1000 ms, scaling virtual cores does not seem to meet the target with just a low number of virtual cores.

The reason behind is that the RUBiS VM is co-located with other VMs, which are CPU hungry, and the resulting performance variability per VMs and per core is high. The effective capacity received per VM is not linearly proportional

to its allocation of virtual cores. Consequently, increasing the number of cores is more effective when the number of virtual cores is low (2-3) but becomes less effective for larger number of cores.

Multiple VMs. Now we move into the cluster scenario where three RUBiS VMs are serving requests that arrive with a step-wise time-varying pattern as indicated on the upper x-axis of Fig. 1b, ranging between 50 to 100 requests per second. The performance target is to ensure no more than 5% of requests has a latency longer than 1000 milliseconds, i.e., the 95th percentile needs to be lower than 1000 milliseconds. Each VM is subject to emulated interference and we refer the readers to Section VI for details. As the workload dynamically varies, we adjust the number of virtual cores per VM. We use two approaches to adjust the cores on each VM: (i) manual-tuning, and (2) the control mechanism proposed in [29]. We plot their latency of 95th percentile (top row) and the average number of allocated cores (bottom row) on the three VMs in Fig. 1b.

In manual-tuning, we run the workload a few times and manually tweak the virtual core allocations so as to find the optimal allocation of virtual cores. In contrast to the single VM result, the resulting tail latency here is better by manually tuning the cores on the three VMs and load balancing the requests in a round robin fashion. However, it rarely meets the target of 1000 ms. One can see the manual-tuning as the optimal solution that one can achieve by scaling the number of virtual cores only. As for the vertical controller, its 95th percentile is consistently lower than the manual-tuning due to its reactive nature while the target is being met only occasionally. From these results, we can clearly see the limitations of scaling virtual resources, be it virtual cores or machines, in meeting the stringent target of the tail latency.

Our objective in this paper is to introduce another control variable, the clone factor, to further strengthen the power of resource scaling, particularly virtual scaling on virtual cores.

III. Related Work

On-line user facing applications deployed in the cloud manifest variable performance in their life time due to various reasons such as variability in workload patterns [15], [25], resource contention due to co-located workloads [45], [17], [28], and heterogeneity in the infrastructure [18], [39]. Different techniques were proposed to mitigate such performance variability: (i) scaling resources, (ii) prioritization, and (iii) more recently query cloning.

Scaling Resources. A number of studies have centered on elastic resource provisioning that either determines the size of a VM by adjusting the different resource types (e.g., CPU, memory)–`vertical scaling`– [27], [43], [23], [29] or the number of VMs–`horizontal scaling`– [32], [36], [31] such that the costs of operation, energy, and performance penalty are minimized. Vertical elasticity is about adding or removing resources (e.g., cores, memory) from a VM at a fine granularity, both in amount and time. Horizontal elasticity consists in adding or removing VMs to or from an application, e.g., based on the number of end-users. However, almost all resource scaling approaches target average system behaviors focusing on different Key Performance Indicators (KPIs) (e.g., latency, resource utilization, number of requests arriving at the system, number of jobs done) with little attention to tail behavior.

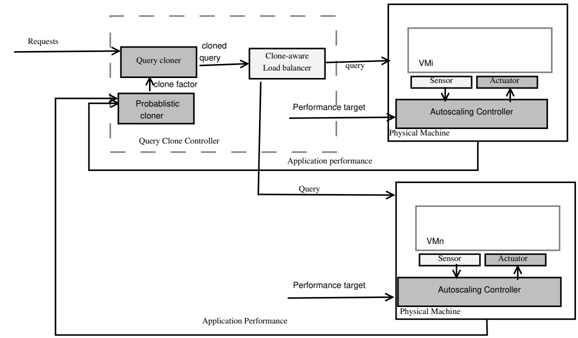


Fig. 2: SmallTail's architecture.

Prioritization. Applications are divided into priority classes and capacities are distributed among applications based on their priority during resource shortage [26], [33], [28], [46]. Kleinrock [26] studied time-sharing of a Physical Machine (PM) among processes with different priorities in non-virtualized environments. Padala et al. [33] present a control-theoretic approach that dynamically allocates capacity in order to meet performance targets of different services running on different VMs and proportionally distributes resources based on fixed weight. The authors in [28], [46] proposed a mechanism to distribute resources based on observed performance besides fixed weight during resource shortage. The underlying assumption in these works is that the resource demand exceeds the infrastructure limit. As a result these works are orthogonal to our approach.

Query Cloning. Cloning queries speculatively has been shown to be effective strategy to improve the latency of interactive web services [37], [38], [50] as well as big data platforms [12], with the implicit assumption of sufficient capacity to accommodate clones. Recently proposed queueing models [20], [41] try to identify the optimal cloning levels that achieve the minimum latency. All existing approaches proactively clone arriving requests by an integer factor, sending to different VMs, and returning only the fastest request. Such strategy of cloning all incoming queries puts much pressure and cost on the system as the rest of redundant requests are left to complete due to the complexity of implementing cancellation policies resulting a significant load increase in the system [48].

In this work we focus on determining the optimal fraction of requests that should to be cloned that strikes the trade-off between the latency gain and additional processing overhead.

IV. SmallTail

This section formally introduces the key components of SmallTail. We consider a cloud infrastructure that hosts inter-

active services, each with different characteristics, as well as variable and unpredictable workload dynamics. Each service has a Service Level Agreement (SLA) that stipulates a Service Level Objective (SLO) and optionally minimum and maximum resource requirements. The minimum resource constraints are often used to allow each service to maintain some functionality at all times. The maximum limits are usually set to shield the user from unexpectedly high costs due to service malfunctioning or an attack. For example the minimum or maximum resource requirements can be expressed as number of cores per instance or total number of instances. The SLO is a target value for a KPI for example, a specific value for average or tail response time of the system. Even though the focus of this work is for tail latency, the approach can be applied for median, average or quartiles. The goal is to continuously adjust the allocated resource levels and clone factor, without human intervention, to drive KPIs toward their targets. Specifically, the resource allocation strategy should be capable of allocating just the right amount of resources for a service composed of multiple instances at the right time in order to meet its respective performance target, avoiding both resource under- and over-provisioning while the replication factor augments the performance variations across the instances.

A key feature of this system is that it is subject to a time varying demand pattern, i.e., the arrival rate of queries $\lambda(t)$ varies with time t and with potentially variable load for each request, i.e., variable resource requirement for each request. At any given time t, the system consists of $V(t)$ VM instances, each with $C(t)$ cores processing μ requests per second (i.e, μ is the service rate). We further assume that all VMs are replicas of the same service and capable of executing all queries. Queries arrive at a central *query clone controller*, where they can be cloned before being dispatched to the next available instance, as shown in Fig. 2. The query clone controller also takes care of returning the response to the client once the first clone completes, and initiates the cancelling of outstanding clones if such a policy is in place.

To maintain the tail latency target, the system is able to scale along three dimensions each at different time scale: (i) the number of clones, or clone factor, $r(t)$, (ii) the number of virtual cores per instance, $C(t)$, and (iii) the number of provisioned VMs, $V(t)$. In this work we focus on the first two dimensions and assume enough number of instances can be spawned when resources in the physical machine are not sufficient to meet the demand. The third dimension is left as future work. In what follows, we explain the design considerations regarding these two control dimensions. For simplicity we drop the index t unless it is explicitly required.

Query Cloning. Cloning queries has been proposed as a solution to mitigate slow execution, either reactively after experiencing long delays, or proactively upon the arrival of queries [37], [12]. All clones are sent to different VMs so as to best take advantage of the variability across VMs, i.e., to increase the probability of a clone being executed on a "fast" VM. For each query, as soon as one of its clones completes, the result is returned to the user. Existing approaches clone all arriving requests by an integer factor. In this work, we particularly focus on probabilistic query cloning, i.e., a query is replicated with some probability. The intuition behind probabilistic query cloning is that cloning all requests may induce further performance degradation due to load increase in the system and the performance of some requests may be guaranteed even without cloning due to their light-weight nature.

To adapt to the dynamic load and VM conditions, the `query clone manager` continually adjusts the cloning probability using a simple reactive step control. The controller periodically checks the KPI value every Δ seconds, compares it to its previous value and modifies accordingly the clone factor r by $\pm r_\Delta$. If the KPI trend is improving the controller continues modifying r in the same increasing/decreasing direction as in the previous control interval. If the trend is worsening, the controller modifies r in the opposite direction. After having modified r the controller uses clipping to limit r within $r_{min} < r < r_{max}$. Once the value of r is determined, each incoming request is replicated randomly with probability r for the next Δ seconds. In our experiments we set $\Delta = 0.5$ seconds and $r_\Delta = 0.25$ (these values were determined experimentally).

Vertical Autoscaling Controller. The vertical autoscaling controller is based on our previous works [27], [29]. The controller takes a target latency value and observed average latency of the system as input during an interval and outputs the amount of cores required for the next interval to meet the target. Its goal is to continuously adjust the allocated resource levels, without human intervention, to drive the observed value toward the target irrespective of variations in workload patterns. Specifically, the resource allocation strategy tries to allocate just the right amount of resources for a service at the right time in order to meet its respective performance target, avoiding both resource under- and over-provisioning. The controller is slightly modified to control the tail latency instead of the mean in collaboration with the query cloning controller.

The vertical autoscaling controller loosely follows a Monitor Analyze Plan and Execute with Knowledge *(MAPE-K)* loop based on self-adaptive software terminology [24]. The monitor periodically collects measurements such as average and tail latency and resource utilization. The period between observations is also used as the period between activations of the controller. The analysis and planning phase is the main part of the controller where the computation of CPU cores for the next time interval is predicted based on previous observations. Execution consists of configuring the hypervisor to effect the computed resources. Previous monitoring data is used to fit the model parameters, which represent the knowledge component in the MAPE-K loop. To enforce the control decision and easily enable vertical elasticity, we used the Xen hypervisor [13] which supports CPU hotplugging without the need to restart the application. In our experiments, we recompute capacity to the application periodically, with 10 seconds interval, which is short enough to make the system reactive and long enough to observe the effects of the new

capacity allocation on the performance of the service [34].

V. Probabilistic Cloning

To capture the impact of partial replication we develop a queueing model that explicitly considers that each incoming request is cloned with probability p, i.e. $p = r - 1$. With this model we are able to obtain the mean request response time, where both cloned and non-cloned requests contribute to the overall result.

A. Definitions

We assume requests arrive according to a Poisson process with arrival rate λ. Each request is replicated with probability p, whereas with probability $q = 1 - p$ no cloning is applied. Each request is processed by one of C available servers (total number of cores) in first-come-first-served (FCFS) order. If all servers are busy, requests (and their clones) are kept in a queue where they wait for the next available server. The processing time of a single clone is exponentially distributed with mean $1/\mu$.

B. Building and Solving the Model

To obtain the mean response time we setup a Markov chain where the state is the number of clones in the system, both in processing and waiting. The state space is thus $\{0, 1, 2, \dots\}$, and the transition rates among states can be describe with the generator matrix $\boldsymbol{Q}$ given by

$$\boldsymbol{Q} = \begin{bmatrix} \circ_0 & \lambda q & \lambda p & & & & & \\ \mu & \circ_1 & \lambda q & \lambda p & & & & \\ & 2\mu & \circ_2 & \lambda q & \lambda p & & & \\ & & \ddots & \ddots & \ddots & \ddots & & \\ & & & C\mu & \circ & \lambda q & \lambda p & \\ & & & & C\mu & \circ & \lambda q & \lambda p \\ & & & & & \ddots & \ddots & \ddots & \ddots \end{bmatrix}. \quad (1)$$

The entries above the diagonal of $\boldsymbol{Q}$ show that from a state with k clones, the system moves to a state with $k+2$ clones with rate λp as the incoming request is replicated with probability p. Instead, with rate λq the system moves to state $k+1$ as no replication is applied to the incoming request. The entries of $\boldsymbol{Q}$ below its diagonal show the rate at which requests complete service. In a state with k clones, a request completes service with rate $\min\{k, C\}$ as there can be at most C requests in service. The symbols $\circ$ simply represent the elements in the diagonal that make the row sums of $\boldsymbol{Q}$ equal to zero.

From the matrix $\boldsymbol{Q}$ in (1) we aim to obtain the steady-state probabilities $\{\pi_k\}_{k\geq 0}$, where π_k is the probability of finding the system with k clones. To this end we group together the states in pairs to make levels, such that level 0 is made of state 0, level 1 is made of states 1 and 2, and in general level k is made of states $2k-1$ and $2k$. From here onward we assume that the number of servers C is even, but the steps can be easily adapted to the case with an odd number of servers. By grouping states in levels we end up with a quasi-birth-and-death process (QBD) [30] where transitions are only allowed among states in adjacent levels. Since levels are made of multiple states, transitions rates among adjacent levels are put together into square matrices with size equal to the number of states per level, i.e., 2 in this case. Although we can define these matrices for all levels, in the interest of space we focus on the levels $k \geq C/2$, where a repetitive pattern starts such that transitions from level k to levels $k-1$, k, and $k+1$ are ruled by matrices $\boldsymbol{A}_{-1}$, $\boldsymbol{A}_0$, and $\boldsymbol{A}_1$, respectively, given by

$$\boldsymbol{A}_{-1} = \begin{bmatrix} 0 & C\mu \\ 0 & 0 \end{bmatrix}, \ \boldsymbol{A}_0 = \begin{bmatrix} \circ & \lambda q \\ C\mu & \circ \end{bmatrix}, \ \boldsymbol{A}_1 = \begin{bmatrix} \lambda p & 0 \\ \lambda q & \lambda p \end{bmatrix}.$$

As with the matrix $\boldsymbol{Q}$, we also group together the stationary probabilities $\{\pi_k\}_{k\geq 0}$ by levels such that $\boldsymbol{\theta}_0 = \pi_0$ and $\boldsymbol{\theta}_k = [\pi_{2k-1} \ \pi_{2k}]$ for $k \geq 1$.

We can now find the probabilities $\{\pi_k\}_{k\geq 0}$ in two steps. First, we must find the rate matrix $\boldsymbol{R}$ as the minimal non-negative solution to the matrix equation

$$\boldsymbol{A}_1 + \boldsymbol{R}\boldsymbol{A}_0 + \boldsymbol{R}^2\boldsymbol{A}_{-1} = \boldsymbol{0}.$$

This matrix allows us to write the stationary probabilities for levels $k \leq C/2$ as

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{C/2} R^{k-C/2}, \ k \leq C/2,$$

thanks to the matrix-geometric property [30], which means that it is sufficient to find $\boldsymbol{\theta}_{C/2}$ to determine all vectors $\boldsymbol{\theta}_k$ for levels $k \geq C/2$. The second step is to use the matrix $\boldsymbol{R}$ to solve a linear system associated to the levels $\{0, 1, \dots, C/2\}$ to find the vectors $\boldsymbol{\theta}_k$ for these levels. We omit the details in the interest of space, but we highlight that this is a linear system of size $C+1$ that has the same block-tridiagonal structure as $\boldsymbol{Q}$ and can therefore be solved very efficiently even for very large values of C.

C. Obtaining the Mean Response Time

Having found the vectors $\{\boldsymbol{\theta}_k\}_{k\geq 0}$, and thus the stationary probabilities $\{\pi_k\}_{k\geq 0}$, we can now determine the mean response time. We start by determining the probability that a request must wait before being served as

$$\gamma = \sum_{k\geq C} \pi_k = \pi_C + \sum_{k\geq 1} \boldsymbol{\theta}_{C/2+k}\mathbf{1} = \boldsymbol{\theta}_{C/2}(\boldsymbol{I} - \boldsymbol{R})^{-1}\mathbf{1} - \pi_{C-1}, \quad (2)$$

where the last equality results from the matrix-geometric property mentioned above. The response time experienced by a request will depend on the state in which it finds the system. Thanks to the PASTA property [49] the probability that an incoming request finds the system k is precisely π_k. Thus, the probability that a request finds at least two idle servers is

$$\sum_{k=0}^{C-2} \pi_k = 1 - \gamma - \pi_{C-1}.$$

In this case the response time is made of the service time only, and is given by

$$p\frac{1}{2\mu} + q\frac{1}{\mu},$$

since with probability p the request is replicated and the mean service time is $1/2\mu$, whereas with probability q the request is not replicated and its mean service time is simply $1/\mu$. Instead, the request finds at most one idle server with probability $\gamma + \pi_{C-1}$, and in this case its service time is given by

$$p\left(\frac{1}{C}\frac{1}{\mu}+\frac{C-1}{C}\frac{1}{2\mu}\right)+q\frac{1}{\mu}.$$

This expression captures that if a request is replicated, with probability p, its first replica may finish before any other request in service with probability $1/C$, and in this case the mean service time is simply $1/\mu$. Alternatively, any other request finishes service before, with probability $(C-1)/C$ and in this case the mean service time is $1/2\mu$. As before, if the request is not replicated, with probability q, its mean service time is simply $1/\mu$. Finally, if the request must wait, finding $C+k$ requests in the system, it must wait for the completion of $k-C+1$ requests before it can start service. As each request service completion takes on average $1/C\mu$ time, the mean waiting time is

$$\sum_{k\geq C}\pi_k\frac{k-C+1}{C\mu}.$$

The following lemma puts everything together.

Lemma 1. *The mean response time in a system with partial replication and the characteristics described in Section V-A is given by*

$$(1-\gamma-\pi_{C-1})\left(p\frac{1}{2\mu}+q\frac{1}{\mu}\right)+$$
$$(\gamma+\pi_{C-1})\left(p\left(\frac{1}{C}\frac{1}{\mu}+\frac{C-1}{C}\frac{1}{2\mu}\right)+q\frac{1}{\mu}\right)+$$
$$\sum_{k\geq C}\pi_k\frac{k-C+1}{C\mu}.$$

VI. Experimental Setup

Hardware and Instance Setup. The experiments were conducted on three PMs each equipped with a total of 32 cores[1] and 56 GB of memory. To emulate a typical cloud environment and easily enable vertical elasticity, we used the Xen hypervisor [13]. An instance of the application was deployed in each PM. Each instance of the tested application was deployed with all of its components, such as web servers and database servers, inside its own VM as is commonly done in practice [44], e.g. by using a LAMP stack [10].

Benchmark Applications. We used three different benchmark applications: RUBiS [7], RUBBoS [6] and SOLR [2] which simulate a web commerce, bulletin board and web searching system, respectively. All three are widely-used interactive cloud benchmark applications – see e.g., [42], [47], [19], where computations are only performed as a result of a user request. We used as web-server Apache for all three applications. Each instance was configured as follows. The `Apache MPM prefork` module, which is thread safe and therefore suitable to be used with PHP applications, was enabled. We set parameters regarding `Apache` processes (e.g., `MaxClients` and `ServerLimit`) to relatively high values. This value is well above the number of concurrent requests that `Apache` has to deal with during any of the experiments to prevent Apache from becoming a bottleneck.

Collocated applications. At each PM we collocated different micro-benchmarks which put pressure on different resource types such as CPU, memory, network and IO to make the setup as close as possible to the real cloud environment. To load the CPU, we used the CPU-bound benchmark from the `SysBench` [9] package. To pressure the memory management system, we used STREAM [8]. For `disk IO` and `network` we used the Flexible I/O (FIO) [4] benchmark and the netperf [22] benchmark, respectively. We also use another RUBiS instance as a background process to emulate a cloud environment where different interactive applications are deployed on the same machine.

While the actual load intensity for each type of background workloads was slightly varied to fit the applications under study, the intensity remains constant across the different cloning strategies, i.e. no, probabilistic and deterministic cloning. For example, the collocated applications perform similar operations during the run of the different cloning strategies shown in Fig. 4a.

Workloads. The experiments were performed using the four combinations of workloads shown in Fig. 3: predictable vs unpredictable and high vs low loads. The predictable workloads, see Figs. 3a and 3b, are based on traces collected at the Wikipedia website [11] which show a typical, rather smooth sinusoidal day-night pattern. These stand in contrast to the bursty, difficult to predict traces, see Figs. 3c and 3d collected during the FIFA world cup [3]. For each source we randomly selected two days and scaled the corresponding trace in time, from 24 hours to 2 hours, and space, i.e. number of requests, to adapt to the different service times of the different benchmarks. In particular we used the low-load traces with RUBiS and SOLR, and the high-load traces for RUBBoS.

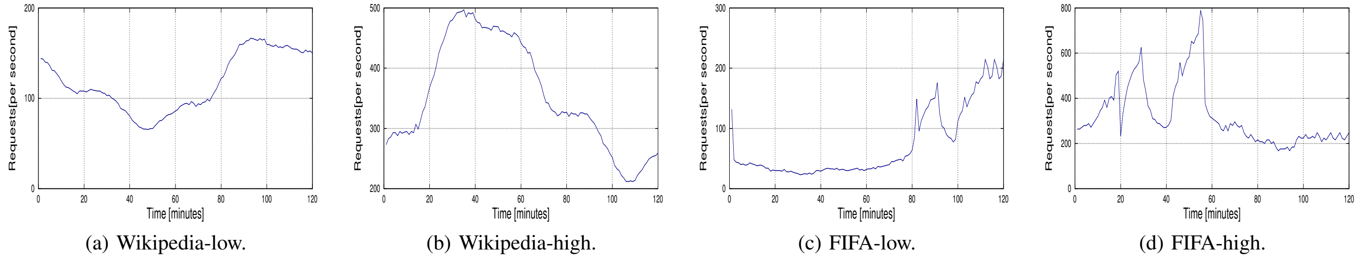
The workload was generated following a closed-system model [40]. A closed user loop model is defined when the arrival of new requests is only triggered by previous request completions, followed by a delay according to *thinktime*. The effective average request inter-arrival time is the sum of the average *thinktime* and the average response time of the application.

To emulate the users accessing the applications, we used `httpmon` tool, a custom workload generator[2], which supports the closed-system model client behavior. The think-time of each client was fixed at 1 second and the number of users was varied following each workload trace.

Note that the workload generator, `httpmon`, sends the request to the `query clone manager or load balancer`, where requests are dispatched to each instance according to the `clone factor`.

[1]Two AMD Opteron™ 6272 processors, 2100 MHz, 16 cores each, no hyper-threading.

[2]https://github.com/cloud-control/httpmon



(a) Wikipedia-low. (b) Wikipedia-high. (c) FIFA-low. (d) FIFA-high.

Fig. 3: Real-world workloads: predictable Wikipedia-based and unpredictable FIFA-based traces.

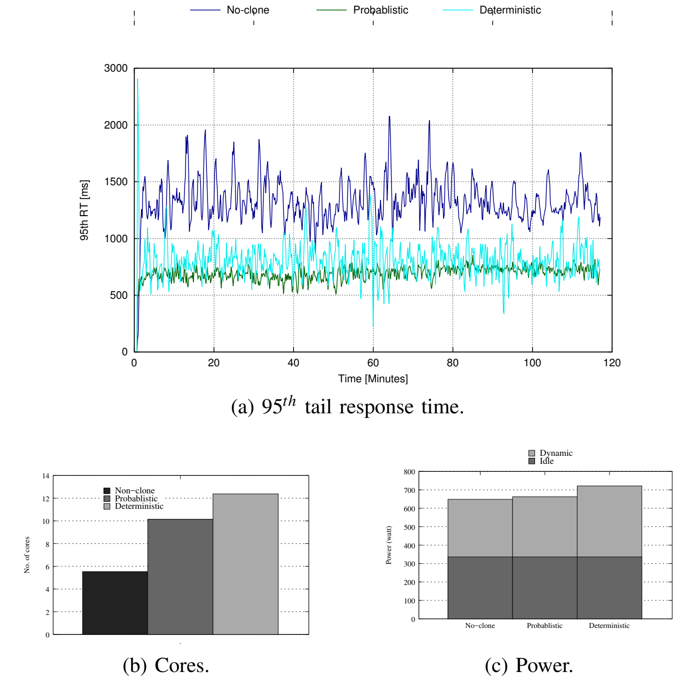
Metrics. The response time of a request is defined as the time elapsed from sending the first byte of the request to receiving the last byte of the reply. The average, and 95% response times were collected over 10 seconds. A rack-mounted HP AF525A Power Distribution Unit (PDU) meter was used to measure the power used by the servers. The pwoer readings were extracted via Simple Network Management Protocol (SNMP) from each server in 10 seconds intervals.

VII. Experimental Evaluation

This section highlights the observations from the different experiments when the `query clone manager` was deployed in parallel with the `vertical auto-scaling` controller. The KPI for the `vertical auto-scaling` was set to the 95th percentile response time with a target of 1 second. An instance of the `vertical auto-scaling` controller was deployed at each PM to manage the VM (or instance) at that particular PM. The `query clone manager` was configured and run with three different setups: no cloning – $r = 1$ always–, deterministic cloning –$r = 2$ always– and probabilistic cloning. We use the no cloning and deterministic cloning as baselines. For each experiment we record the response times, with a particular focus on the tail via the 95th percentile, the allocated cores across all instances, and the overall power consumption. In particular here we consider the dynamic power which is influenced by the system activity. The total power would include also the aggregate idle power of 336W.

A. *RUBiS*

The RUBiS benchmark emulates a web auctioning site similar to eBay. We ran RUBiS using the low-load based workloads. Fig. 4 presents results of the Wikipedia-based workload, see Fig. 3a. The x-axis of Fig. 4a shows the elapsed time since the start of the experiment, whereas the y-axis measures the 95th percentile of the response time under the no, deterministic, and probabilistic cloning. One can see that in all cases `vertical auto-scaling` ensured a constant mean performance across time even if the load was changing. However without cloning, `vertical auto-scaling` was not able to meet the target of 1 second. Only when activating deterministic or probabilistic cloning it was possible to meet the target. Cloning counters the performance variability observed at the instances. This is limpid here. Indeed one can observe



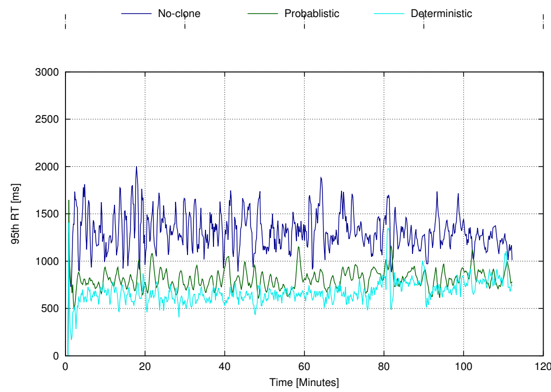
(a) 95th tail response time.

(b) Cores. (c) Power.

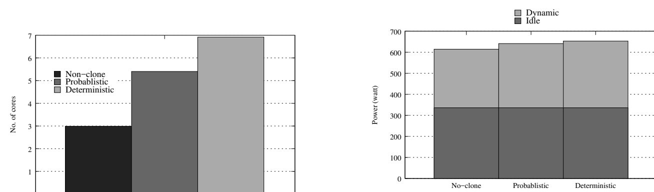
Fig. 4: RUBiS with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3a workload.

that from no cloning ($r = 1$ always) to deterministic cloning ($r = 2$ always) to probabilistic cloning (mean $r = 1.62$) the 95th percentile not only improves but also stabilizes. This can be observed from the reduced oscillations in latency across time. Probabilistic cloning exceeds deterministic cloning because it tries to hit the sweet spot of countering the performance variability and, as we see next, the increase in computational demand.

The cost of cloning is shown in Fig. 4b, i.e. an increased demand in computational resources and energy. Using no cloning the number of cores allocated to RUBiS was 5.5. Deterministic cloning doubles the total load across the instances. To cope with this increase in computational demand, the `vertical auto-scaling` more than doubles the assigned cores, i.e. 12.4 (+123%). Probabilistic cloning only clones on average 62% of the requests and hence stays in between, i.e. 10.1 cores (+83%). The increased computational demand translates into an increase in the dynamic power consumption of the



(a) 95th tail response time.



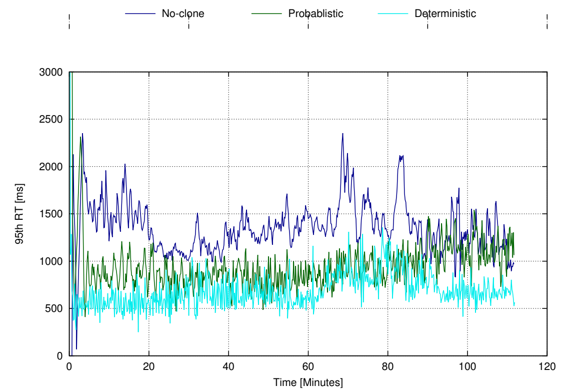
(b) Cores.

(c) Power.

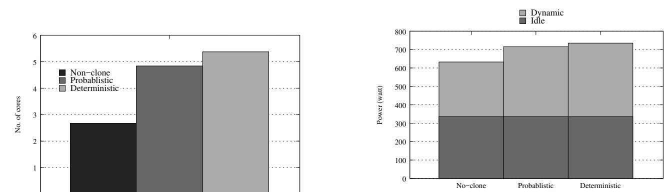
Fig. 5: RUBiS with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3c workload.

PMs, see Fig. 4c. No, deterministic and probabilistic cloning consume 313W, 385W (+23%) and 327W (+4.5%) of dynamic power, respectively. The discrepancy between the increase in cores and power consumption is due to the `vertical auto-scaling` overestimating the computational demand of probabilistic cloning. Indeed the number of cores only sets an upper bound on the dynamic power consumption, but the effective number of used cores (and dynamic power consumption) might be lower. Aggregating the results of the figures, one can see that probabilistic cloning is able to achieve the best response times, while saving on computational demand and consumed power, i.e. -18.0% cores and -15.2% dynamic power with respect to deterministic cloning.

The Wikipedia-based workload is rather smooth which eases the control. To present a more challenging yet real-world inspired case, we run RUBiS with the FIFA-based workload shown in Fig. 3c. This workload includes some load spikes. Fig. 5a depicts the response time results. We see that even when the load bursts arrive the `vertical auto-scaling` is able to well control the response times, but cloning is necessary to achieve the target of 1 second for the 95th percentile. Due to the toggling nature of our simple control, the cloning manager slightly underestimated the required cloning (mean $r = 1.45$) and deterministic cloning slightly outperforms probabilistic cloning in the lower load phase. The corresponding costs are shown in Figs. 5b and 5c. We see that probabilistic cloning saves on resources, i.e. 5.4 cores (+81%), and power, i.e. 305W (+9.6%) over deterministic cloning, i.e. 6.9 cores (+131%) and 317W (+13.9%). Probabilistic cloning is able to achieve the response time target at a reduced cost in terms of both cores (-21.9%) and dynamic power (-3.7%).



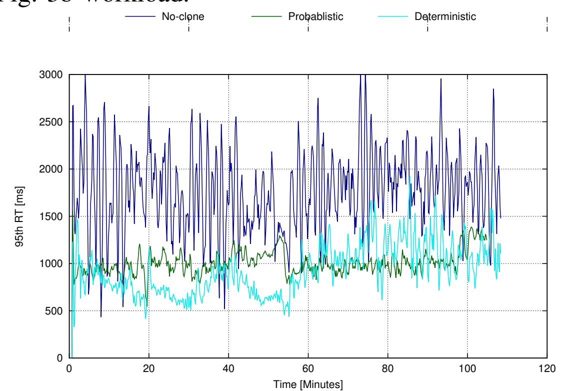
(a) 95th tail response time.



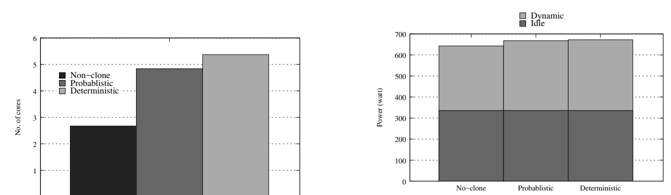
(b) Cores.

(c) Power.

Fig. 6: RUBBoS with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3b workload.



(a) 95th tail response time.



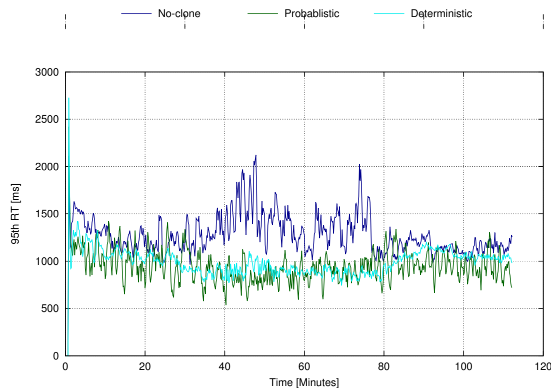
(b) Cores.

(c) Power.

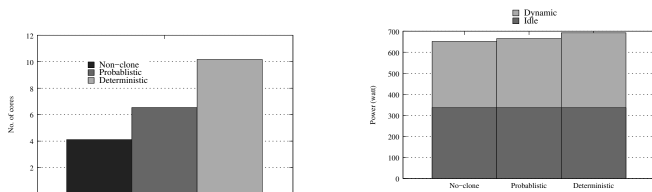
Fig. 7: RUBBoS with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3d workload.

B. RUBBoS

Here we test SmallTail against the RUBBoS benchmark which emulates a bulletin board news forum similar to the slashdot website. This benchmark is able to sustain higher

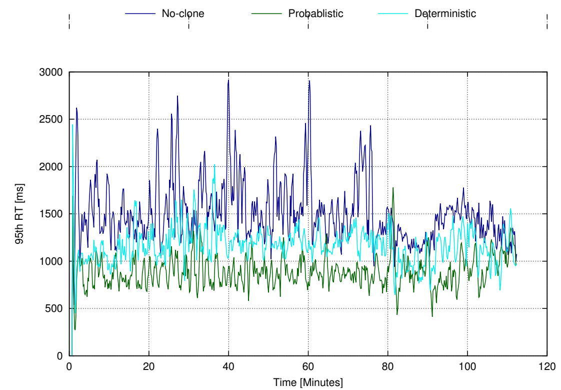


(a) 95th tail response time.

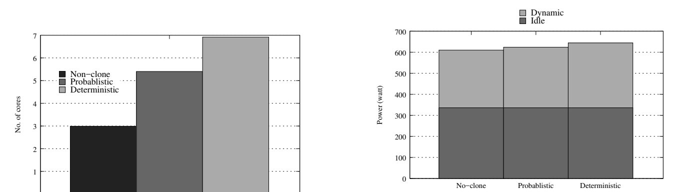


(b) Cores. (c) Power.

Fig. 8: SOLR with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3b workload.



(a) 95th tail response time.



(b) Cores. (c) Power.

Fig. 9: SOLR with 95th percentile target of 1 second under no cloning, deterministic cloning, and probabilistic cloning and Fig. 3d workload.

rates under the same resources as RUBiS. Hence here we used the high load traces.

Figs. 6a and 7a depict the 95th percentile response time under our three cloning strategies. Again cloning is essential in achieving the target under both the Wikipedia- and FIFA-based workloads. The gains of probabilistic cloning over deterministic cloning are -28.1% cores and -4.9% dynamic power, and -9.9% cores and -1.2% dynamic power for the Wikipedia and FIFA trace, respectively, see Figs. 6b, 6c, 7b and 7c.

Comparing RUBBoS against the RUBiS results, we see that here the impact of cloning is higher. We observe a larger difference between no cloning and cloning results. This stems from the fact that RUBBoS requests are subject to a higher variability. Indeed the coefficient of variation for RUBBoS is 1.55 against 1.41 for RUBiS. For example comparing Fig. 5a and Fig. 7a, probabilistic cloning on RUBBoS is able to achieve -43.2% reduction in the 95th percentile of response time against the -37.8 reduction obtained on RUBiS.

C. SOLR

SOLR is a web searching application based on the Apache Lucene library which we set to work on a DB populated offline by crawling 50000 random webpages. Figs. 8 and 9 summarize the results. Cloning is key to achieve the target. Here probabilistic cloning performs even slightly better than in our previous experiments by not only using less cores and power, but also achieving lower response times than deterministic cloning. In case of the FIFA-based trace, probabilistic cloning lowers the 95th percentile response time by -25.9% and the costs by -53.7% and -6.6% against deterministic cloning for computation and power demand, respectively. In the case of the Wikipedia-based trace, these gains become even -40.8% for response time, -35.7% for cores and -7.7% for dynamic power.

We further observe that smoother loads ease the control leading to better results. Comparing the two types of workloads across the three benchmark applications, once can observe that in both the RUBiS and SOLR cases predictive cloning achieves higher latency reductions in case of the smoother Wikipedia-based load.

VIII. Conclusion

Guaranteeing the tail latency target is critical for users' quality of experience when using cloud-based interactive services. Due to the high performance variability caused by co-locating workloads, scaling virtual resources is not sufficient to meet stringent tail latency targets and proactively cloning all requests to curtail latency is too costly. In this paper, we propose the novel concept of probabilistic request cloning, which only replicate a fraction of arriving requests. We also develop a controller, termed SmallTail, that dynamically orchestrates vertical core scaling and probabilistic cloning for interactive web systems. Given the tail latency target, SmallTail adjusts the number of virtual cores in a coarser granularity, e.g., 10 seconds, and the fraction of requests to be cloned in a finer granularity, e.g., 0.5 seconds. We evaluate SmallTail on three web systems, i.e., RUBiS, RUBoS, and Solr, under four real-world traces, with the focus on latency fulfillment, core allocation, and power consumption. Our evaluation results show that SmallTail can maintain the tail latency according

to the target by using only half of the virtual cores that are needed when deterministically cloning all requests.

Acknowledgement

This research has received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement no. 732366 (ACTiCLOUD).

References

[1] Amazon latency study. Available online:https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, Last visited 2017-12-01.

[2] Apache Solr. Available online: http://lucene.apache.org/solr/, Last visited 2017-09-10.

[3] FIFA 1998 Web site Page View Statistics. Available online: http://ita.ee.lbl.gov/html/contrib/WorldCup.html, Last visited 2017-09-10.

[4] Fio. Available online: https://github.com/axboe/fio, Last visited 2017-09-10.

[5] How loading time affects your bottom line. Available online: https://blog.kissmetrics.com/loading-time/, Last visited 2017-12-01.

[6] RUBBoS. Available online: http://jmob.ow2.org/rubbos.html, Last visited 2017-09-10.

[7] RUBiS. Available online: http://rubis.ow2.org, Last visited 2017-09-10.

[8] STREAM. Available online: http://www.cs.virginia.edu/stream/, Last visited 2017-09-10.

[9] Sysbench. Available online: https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench, Last visited 2017-09-10.

[10] Tutorial: Installing a LAMP web server. Available online: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html, Last visited 2017-09-10.

[11] Wikipedia Access Traces. Available online: http://www.wikibench.eu/?page_id=60, Last visited 2017-09-10.

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

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177, 2003.

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

[15] M. C. Calzarossa, L. Massari, and D. Tessera. Workload characterization: A survey revisited. *ACM Comput. Surv.*, 48(3):48:1–48:43, Feb. 2016.

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

[17] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ASPLOS*, pages 127–144, 2014.

[18] 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 *ACM SoCC*, pages 20:1–20:14, 2012.

[19] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48, 2012.

[20] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia. Reducing latency via redundant requests: Exact analysis. In *ACM SIGMETRICS*, pages 347–360, 2015.

[21] G. I. Goumas, K. Nikas, E. B. Lakew, C. Kotselidis, A. Attwood, E. Elmroth, M. Flouris, N. Foutris, J. Goodacre, D. Grohmann, V. Karakostas, P. Koutsourakis, M. L. Kersten, M. Luján, E. Rustad, J. Thomson, L. Tomás, A. Vesterkjaer, J. Webber, Y. Zhang, and N. Koziris. ACTiCLOUD: Enabling the next generation of cloud applications. In *IEEE ICDCS*, pages 1836–1845, 2017.

[22] R. Jones et al. NetPerf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

[23] E. Kalyvianaki, T. Charalambous, and S. Hand. Adaptive resource provisioning for virtualized servers using Kalman filters. *ACM TAAS*, 9(2):10:1–10:35, 2014.

[24] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[25] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *IEEE NOMS*, pages 1287–1294, 2012.

[26] L. Kleinrock. Time-shared systems: a theoretical treatment. *J. ACM*, 14(2):242–261, Apr. 1967.

[27] E. B. Lakew, C. Klein, F. Hernández-Rodriguez, and E. Elmroth. Towards faster response time models for vertical elasticity. In *IEEE/ACM UCC*, pages 560–565, 2014.

[28] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Performance-based service differentiation in clouds. In *IEEE CCGrid*, pages 505–514, 2015.

[29] E. B. Lakew, A. V. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth. KPI-agnostic control for fine-grained vertical elasticity. In *IEEE/ACM CCGrid*, pages 589–598, 2017.

[30] G. Latouche and V. Ramaswami. *Introduction to matrix analytic methods in stochastic modeling*. SIAM, 1999.

[31] Z. Liu, A. Wierman, Y. Chen, B. Razon, and N. Chen. Data center demand response: Avoiding the coincident peak via workload shifting and local generation. In *ACM SIGMETRICS*, pages 341–342, 2013.

[32] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, 2014.

[33] P. Padala, K. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, pages 13–26, 2009.

[34] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, pages 289–302, 2007.

[35] J. F. Pérez, R. Birke, Z. Qiu, M. Björkqvist, and L. Y. Chen. Power of redundancy: Designing partial replication for multi-tier applications.'. In *IEEE INFOCOM*, 2017.

[36] I. Pietri and R. Sakellariou. Mapping virtual machines onto physical machines in cloud computing: A survey. *ACM Comput. Surv.*, 49(3):49:1–49:30, Oct. 2016.

[37] J. F. Prez, R. Birke, M. Bjrkqvist, and L. Y. Chen. Dual scaling VMs and queries: Cost-effective latency curtailment. In *IEEE ICDCS*, pages 988–998, 2017.

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

[39] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, Sept. 2010.

[40] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *USENIX NSDI*, 2006.

[41] N. B. Shah, K. Lee, and K. Ramchandran. When do redundant requests reduce latency? *IEEE Trans. Communications*, 64(2):715–722, 2016.

[42] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *ACM SoCC*, page 5, 2011.

[43] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith. Runtime vertical scaling of virtualized applications via online model estimation. In *IEEE SASO*, pages 157–166, 2014.

[44] K. Sripanidkulchai et al. Are clouds ready for large distributed applications? *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[45] S. K. Tesfatsion, L. Tomás, and J. Tordsson. OptiBook: Optimal resource booking for energy-efficient datacenters. In *IEEE/ACM IWQoS*, pages 1–10, 2017.

[46] L. Tomas, E. B. Lakew, and E. Elmroth. Service level and performance aware dynamic resource allocation in overbooked data centers. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)(CCGRID)*, volume 00, pages 42–51, 2016.

[47] N. Vasic, D. M. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ASPLOS*, pages 423–436, 2012.

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

[49] R. W. Wolff. Poisson arrivals see time averages. *Operations Research*, 30:223–231, 1982.

[50] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *NSDI*, pages 543–557, 2015.

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