TraceSplitter: A New Paradigm for Downscaling Traces

Sultan Mahmud Sajal*
sxs2561@psu.edu
The Pennsylvania State University

Rubaba Hasan*
rxh655@psu.edu
The Pennsylvania State University

Timothy Zhu
tuz68@psu.edu
The Pennsylvania State University

Bhuvan Urgaonkar
bhuvan@cse.psu.edu
The Pennsylvania State University

Siddhartha Sen
sidsen@microsoft.com
Microsoft Research

Abstract

Realistic experimentation is a key component of systems research and industry prototyping, but experimental clusters are often too small to replay the high traffic rates found in production traces. Thus, it is often necessary to *downscale* traces to lower their arrival rate, and researchers/practitioners generally do this in an ad-hoc manner. For example, one practice is to multiply all arrival timestamps in a trace by a scaling factor to spread the load across a longer timespan. However, temporal patterns are skewed by this approach, which may lead to inappropriate conclusions about some system properties (e.g., the agility of auto-scaling). Another popular approach is to count the number of arrivals in fixed-sized time intervals and scale it according to some modeling assumptions. However, such approaches can eliminate or exaggerate the fine-grained burstiness in the trace depending on the time interval length.

The goal of this paper is to demonstrate the drawbacks of common downscaling techniques and propose new methods for realistically downscaling traces. We introduce a new paradigm for scaling traces that splits an original trace into multiple downscaled traces to accurately capture the characteristics of the original trace. Our key insight is that production traces are often generated by a cluster of service instances sitting behind a load balancer; by mimicking the load balancing used to split load across these instances, we can similarly split the production trace in a manner that captures the workload experienced by each service instance. Using production traces, synthetic traces, and a case study of an auto-scaling system, we identify and evaluate a variety of scenarios that show how our approach is superior to current approaches.

CCS Concepts

• **General and reference** → **Experimentation**; • **Computer systems organization** → *Cloud computing*.

Keywords

Workload Analysis, Cloud Computing, Performance Evaluation

*Both authors contributed equally to this research.

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456262

ACM Reference Format:
Sultan Mahmud Sajal, Rubaba Hasan, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. 2021. TraceSplitter: A New Paradigm for Downscaling Traces. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3447786.3456262

1 Introduction

To realistically evaluate the performance of systems, both researchers and practitioners heavily rely upon production traces, which capture the incoming traffic patterns in real-world systems. A number of these traces are publicly available, such as the recently updated Google trace [25, 40, 49, 53], Microsoft Azure trace [13], Alibaba trace [27], Wikipedia trace [51], and SNIA IOTTA traces [3]. Additionally, researchers in companies have access to private traces. However, because these traces are captured on large production clusters, they often have too high of a traffic intensity for smaller experimental clusters. As a result, replaying such a trace on a smaller experimental cluster requires a pre-processing step to *downscale* the trace.

The ability to downscale traces is important for both academia and industry. Researchers often use downscaled traces on a smaller experimental cluster and claim their results realistically compare the benefits and drawbacks between state-of-the-art approaches [6, 7, 23]. Developers often use downscaled traces to test and debug new features in a smaller test cluster to detect and fix as many bugs as possible before deploying the code in a production environment. In both cases, it is important for the downscaling process to preserve characteristics of the original trace as much as possible.

The purpose of this work is to study the effects of downscaling traces. We demonstrate how common downscaling techniques fail to capture key characteristics of the original trace. For example, the burstiness of the original trace may be exaggerated or diminished depending on how a trace is scaled. Failing to capture these characteristics could lead to designing new systems to deal with phenomena that do not occur in practice, or worse, it could present an overly optimistic view of a prototype's ability to handle real-world phenomena. To alleviate this, we introduce a new paradigm for trace downscaling that more realistically preserves these characteristics. Our technique helps both academics and developers in downscaling traces realistically, thereby improving the fidelity of results on smaller experimental clusters.

Fig. 1 provides an overview of our research. The original trace contains information such as the request arrival time and request size collected from a production cluster. To enable replaying the trace on a smaller target experimental cluster, the downscaling process will

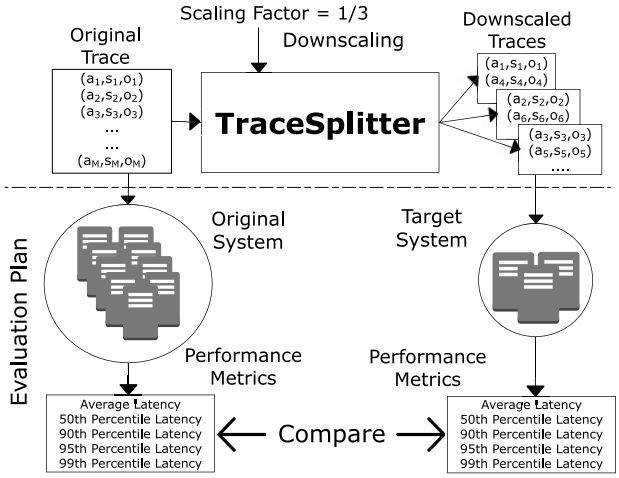


Figure 1: Overview of TraceSplitter.

generate scaled down versions of the trace that have a lower request arrival rate than the original trace according to a user-specified *scaling factor*. Users of our TraceSplitter tool will use one or more of these downscaled traces to evaluate their work on their target cluster. Using all of the downscaled traces will ensure that all the requests in the original trace are replayed. However, we find that in many cases, the downscaled traces are similar; so using a random subset of them would accelerate experimentation. To evaluate the quality of the downscaling process, we compare performance metrics of running the original trace on the original system with running the downscaled traces on the smaller target system. In practice we do not assume/require users to have access to the original system; we only use this access in our study to develop and evaluate TraceSplitter. The focus of our work is on latency-sensitive applications so our primary metrics of interest are the average and tail latency (i.e., time to complete the slowest requests).

Downscaling is fundamentally a lossy process, and thus there is no perfect method for downscaling since there are many sources of randomness, including the system, the original trace, and the downscaling process. At the same time, downscaling is also a somewhat subjective matter, in that the expected outcome from the downscaling process can be different depending on the use case. For example, whereas our interest in this work is in preserving latency properties when downscaling, others may wish to preserve a different set of characteristics such as resource utilization, throughput, cache hit/miss rate, or distribution of user types, to name a few.

Thus, rather than trying to create an "optimal" downscaler, the goal of this paper is to raise awareness of the potential pitfalls of common techniques employed today and introduce a new approach that avoids these pitfalls in a wide range of use cases. Our approach focuses on preserving latency, which is a popular characteristic of interest in experiments. While downscaling can never entirely replace the need to experiment with systems at larger scales, we believe experiments with traces downscaled using TraceSplitter still provide useful insights that can help in developing new system ideas and designs.

From reading through a number of papers, we have identified some of the most popular downscaling approaches used today. Unfortunately, most papers only describe their approach in passing to the effect: "We scale the trace to an appropriate load for our cluster." Of the papers that do describe their scaling methodology in more detail, one common approach is to measure the arrival rate in time intervals (i.e., count the number of arrivals within time windows) and scale the rate as needed [4, 21, 23, 52, 55]. However, this approach is highly sensitive to the length of the time interval. Choosing too large of an interval would eliminate any fine-grained burstiness found in the original trace. Choosing too small of an interval would result in rounding issues that may exaggerate the burstiness and potential introduction of artificial burstiness that is not present in the original trace. Another popular approach is to scale the arrival timestamps in the original trace to spread the load across a longer timespan [6, 20, 34, 52]. However, this approach distorts the temporal patterns found in the original trace. For example, diurnal load patterns may be stretched over multiple days, and this may affect the amount of time available for auto-scaling systems to react. Lastly, another common approach for downscaling is to randomly sample requests from the original trace. While this avoids some of the problems from the previous approaches, sampling-based approaches omit some requests in the original trace, which may not be ideal for traces with a small number of rare requests that disproportionately impact performance.

Key Insight: To address these shortcomings, we introduce a new downscaling approach based on the idea of *splitting traces similar to how a load balancer splits traffic*. Our idea is motivated by the insight that many systems today have a front-end load balancer that distributes work across the servers within the original system. Since production systems often employ a well-designed load balancer, we may reasonably assume that any two servers (or equal-sized subsets of servers) experience statistically similar traffic in terms of the workload features that affect latency. This in turn suggests a very natural way of constructing a downscaled trace: pick the traffic serviced by a subset of the servers that matches the workload intensity desired by the experimenter on the target system. What is particularly worth highlighting is that this subset of servers may be an entirely hypothetical construct that may be determined by *simulating* an imagined load balancer. In this manner, multiple downscaled traces, one for each distinct subset of the appropriate size, may be derived. A key benefit of this approach is that since the original trace is split into multiple downscaled traces, we do not lose any information in the sense that any rare requests would show up in at least one of the downscaled traces.

Our new paradigm of framing trace downscaling in terms of load balancing opens up many possibilities based on the selected load balancing policy. For example, a random load balancing policy reduces to the prior approach of random sampling, except that we retain each request in at least one of the scaled traces. The round robin load balancing policy would result in a deterministic sampling approach to downscaling. Deterministic sampling could result in skewed behaviors in traces with deterministic patterns, so a randomized round robin approach may yield a good balance between the round robin and random sampling approaches. However, we find that, in the presence of request size variability, the least work policy tends to achieve some of the best scaling results. When splitting a

trace into multiple downscaled traces, this policy assigns requests to the downscaled trace with the least work so far in the trace. Since many systems use a dynamic load balancing policy such as least work (left), we find that splitting using this load balancing policy most closely matches the original trace behavior for a wide range of scenarios. Our experimental results with a mixture of real-world production traces and synthetic traces show that this approach works better than the common approaches used today.

Contributions: Our contributions are three-fold:

- We demonstrate how the most popular scaling techniques used in research have the potential to lead to skewed latency results depending on the original workload and how it was downscaled (Sec. 4). We conduct a case study to show how downscaling can lead to picking incorrect parameters in an autoscaling system (Sec. 5).
- We introduce a new paradigm for thinking about downscaling as a load balancing problem (Sec. 3). This idea leads to multiple downscaling approaches that are superior to current approaches used in practice today.
- TraceSplitter is available as an open-source tool at https://github.com/smsajal/TraceSplitter.

2 Related Work

In our review of the literature, we find that very often the downscaling methodology is not described in enough detail to be reproduced [15, 48, 54, 58]. A small number of papers do describe their downscaling methodology, and these tend to fall into one of the following three broad categories: (i) sampling, (ii) model-based scaling, and (iii) timespan scaling.

Sampling: One popular approach for downscaling is based on sampling from the original trace [7, 12, 30, 31, 35, 37, 47]. The key drawback of sampling is that it may fail to include important requests. Furthermore, in the case of random sampling, the probability of failing to include an important request is higher for rarer request types. As a result, sampling-based approaches may yield downscaled traces that are not representative of the original trace. On the other hand, random sampling can also cause rare events to be over-represented. With workloads where rare events also happen to have very different performance characteristics from the remaining events, such under/over-representation can lead to misleading results in experiments using the downscaled trace.

Model-Based Scaling: These approaches are based on devising parameterized analytical/computational models from the original trace and then using these models with suitable parameter settings for the target system. Perhaps the most popular example of such an approach is based on measuring the average request arrival rate within time intervals of a fixed length, multiplying the arrival rates by the scaling factor, and generating scaled traces via an arrival process (e.g., Poisson process) [4, 21, 23, 52, 55]. Our evaluation compares against this approach, which we label as AvgRate. The efficacy of such an approach tends to be very sensitive to the choice of this fixed interval—choosing too small an interval may result in exaggerating the burstiness in the arrival process relative to the original trace, while choosing too large an interval may suppress it. Fig. 15 in Sec. 4 offers concrete examples of this sensitivity to the choice of the interval. Similar models are also developed for other features besides the arrival rate. We find such approaches used with the Google trace [9–11, 28, 33, 38, 43–46, 56, 57], ClarkNet trace [8], Wikipedia trace [8], Microsoft Azure trace [16, 19], Alibaba trace [36], and others [22, 26, 42]. The key shortcoming of these modeling-based approaches is that their efficacy crucially depends on the quality of the model. Generally speaking, the number of workload properties relevant to an experimenter's interest may be large, making it next to impossible to devise effective models for each property. This means the onus of choosing the right set of properties is on the experimenter. This is fraught with the risks of missing out key properties, choosing irrelevant ones, etc., because more often than not, (a) the experimenter is not an expert in devising such models, and (b) their core interests are orthogonal to such modeling. Finally, relevant properties worth modeling can vary from one research problem to another, implying that the modeling insights from one work may not readily transfer to another.

Timespan Scaling: Another trace scaling approach, which we call timespan scaling, multiplies the arrival times in the original trace by (1/scaling factor) to spread the load over a longer period of time [6, 20, 34, 52]. There are obvious downsides to this approach. Most prominently, downscaling by stretching arrival times results in distorting temporal patterns. For example, time of day effects may be stretched over multiple days. This can be problematic in some systems such as autoscaling systems, which would have more time to react to load increases than the original workload would have allowed. This can lead to an improper assessment of the agility and overall efficacy of the autoscaling system. A more subtle (and perhaps counter-intuitive) downside to this approach is that, since it downscales by "stretching" arrival times, it may create longer-lasting bursts than in the original workload. This may translate into a deceptive overload with correspondingly degraded performance. We demonstrate this concretely in Fig. 9a vs. Fig. 9c.

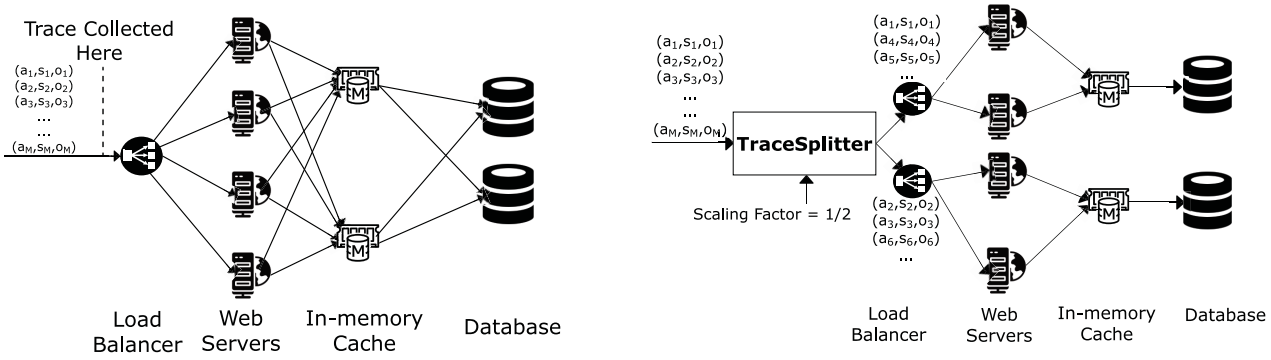
The effects of potentially unrealistic downscaling on an experiment depends on the context in which the traces are used, the particular property being evaluated, the trace characteristics, the care the authors have taken in their evaluation process, etc. Due to these factors, the discussed works may not have suffered significantly from unrealistic downscaling depending on the context of their evaluation. Nevertheless, it is important to understand the potential pitfalls and weaknesses of current downscaling approaches. As an example, our autoscaler case study in Sec. 5 shows how downscaling techniques can lead to erroneous conclusions in real world scenarios.

3 TraceSplitter Design and Implementation

We first describe the goal and scope of TraceSplitter. Next, we present the key insights underlying its design. Finally, we discuss some practical considerations that arise when using TraceSplitter.

3.1 Goals and Scope

Given a trace collected from an *original system*, the goal of TraceSplitter is to produce downscaled traces that—when used on a smaller *target system*—faithfully preserve characteristics of the original trace such as average and tail latency. Downscaling traces enables users to (a) use experimental systems of different sizes, and (b) conduct experiments on a particular setup with traces of different arrival rates to study the effect of different loads. TraceSplitter takes as input an original trace and a user-specified *scaling factor* ($0 < f \leq 1$), which specifies the ratio of the desired average arrival rate in the target system and the average arrival rate in the original



(a) Trace collection in original system.

(b) TraceSplitter acts as a hypothetical load balancer that splits a trace into $\frac{1}{f} = 2$ downscaled traces.

Figure 2: Illustration of the key insight underlying TraceSplitter.

system. Most of our discussion is in terms of arrival rate for the sake of clarity, but one can also scale on the basis of the rate of *work* arriving to the system, which is dependent on both the arrival rate and request sizes of those arrivals. Under the assumption that the throughput of the system scales linearly with the system size (i.e., a system with 4 servers would have double the throughput of a system with 2 servers), one should select the scaling factor based on the ratio of system sizes (e.g., scaling from 4 servers to 2 would result in $f = 0.5$).

The scope of our work is in latency-sensitive systems, which encompass a broad range of cloud-based systems. Users interact with these systems by sending *requests*, which trigger work at one or more machines/nodes in the system. For example, a social network web application uses a set of web servers, caching servers, and database servers to process user requests such as posting a message or browsing the posts. As users expect timely responses, we focus on average and several latency percentiles to capture the tail effects, which are often of primary interest when using traces. It is certainly possible to imagine scenarios wherein a user would like the downscaling process to preserve other types of properties besides (or in addition to) latency-related traits. For instance, in some settings, it may be important to retain the distributions of request types or users. In general, it may not be possible to jointly preserve all specified properties since preserving some may be inherently at odds with preserving others. TraceSplitter assumes that the downscaling process is primarily affecting stateless (or mostly stateless) components of the system. For example, in a multi-tier web system, the intent of our downscaling is primarily to adjust the load on the stateless web tier. While the design of TraceSplitter may provide reasonable results with stateful systems, we have not explicitly accounted for complexities such as caching effects, dependencies between requests, data affinity, flow affinity, etc., which are beyond the scope of the current work.

Trace Format and Content: Let M denote the number of entries in the trace. Each such entry corresponds to a request serviced by the original system, with the i^{th} entry of the form (a_i, r_i, o_i). Requests are sorted by the arrival time a_i. The request size of request i is denoted as r_i, and it will be interpreted along with any opaque data o_i (e.g., request parameters for trace replay) by a user-provided service time estimation function $serviceTimeEstimator\ (r_i, o_i)$. The service time estimator translates the request size into the amount of time or work required to perform the operation, which is used by TraceSplitter to perform a more realistic downscaling. The estimator does not need to be very accurate and is mainly used to gauge the relative differences between request sizes. If an estimator is not available, the default behavior is to treat each request as having the same service time, which works reasonably well in scenarios where service times are not drastically different. Lastly, we use o_i to denote any other relevant information in the trace, which we treat as an opaque data type. TraceSplitter only focuses on the arrival time and request size information, and any other information will be copied through to the downscaled traces. Our trace format is general enough to accommodate most traces we have encountered in our literature review.

3.2 Definitions

We begin by defining a few phrases that are necessary for explaining the key TraceSplitter ideas.

Request Type: A request type represents a category of requests that an experimental setup would serve. For example, the request types serviced by a social media application might include Log In, Send Message, Send Friend Request, etc.

Request Size: The request size is any metric (e.g., payload size, packet size, CPU cycles, etc.) that has a strong correlation with processing time or resource consumption.

Service Time: The service time refers to the amount of time taken to serve the request by the server and is typically based on the request size. It excludes any other delays experienced by the request such as queueing delay. In estimating service time, we are primarily interested in relative differences due to request sizes. For example, assume that a social media application takes t time to perform Send Message with 100 bytes and $2t$ time to perform Send Message with 1000 bytes. If the service time of Send Message with 100 bytes is estimated as 1, then the service time of Send Message with 1000 bytes should be estimated as 2.

3.3 Key TraceSplitter Ideas

The core idea behind TraceSplitter is to frame the trace downscaling problem in terms of how a load balancer splits traffic. That is, we split the requests in the original trace into multiple downscaled traces in a way similar to how a load balancer splits requests across multiple servers. One benefit of this approach is that we retain each

request in the original trace in at least one of the downscaled traces; thus, we do not miss any potentially unusual requests that may expose some performance or correctness issues. This approach also does not over or under represent any of the requests. The rest of this paper aims to justify how TraceSplitter's approach is realistic for downscaling traces while maintaining performance characteristics of the original trace. Here, we present the intuition behind TraceSplitter, and extensive experimentation results are presented in Sec. 4 and Sec. 5.

The intuition behind TraceSplitter stems from the insight that many large scale systems employ load balancers to balance traffic across a cluster of servers. Fig. 2a illustrates a typical web server system with a front-end load balancer. The performance of this original system is predominantly determined by the servers behind the load balancer. For example, the average latency is roughly approximated by the average latency seen by each web server. Thus, the traffic received by the web servers is a reasonable proxy for the overall system behavior. In this sense, capturing the incoming traffic in one web server can be seen as a representative version of the downscaled traffic. Thus, the novelty of our approach is in making the connection to load-balancing techniques and using this as a framework for realistic trace downscaling. TraceSplitter works by playing the role of a hypothetical load balancer as shown in Fig. 2b. In this example, we scale an original trace by $f = 1 / 2$, which generates $\frac{1}{f} = 2$ downscaled traces. For clarity of exposition, we explain our ideas assuming $\frac{1}{f}$ is an integer; Sec. 3.4 discusses how we address non-integral scaling.

Under our load balancing framework, there are a number of load balancing policies that would translate into different trace downscaling approaches.

Random: One of the most basic load balancers is the random load balancer, which would translate to the random sampling trace scaling approach. If one were to apply this policy to the TraceSplitter framework, the main difference with traditional random sampling is that each of the requests would end up in one of the downsampled traces, whereas traditional sampling approaches would ignore/discard requests that weren't sampled.

Round Robin (RR): Round robin is another basic load balancing policy that alternates requests across the servers in a deterministic fashion. When applied to trace downscaling, RR translates to a deterministic sampling approach where every i^{th} request belongs in the $\left(i \bmod \frac{1}{f}\right)$-th downscaled trace.

Randomized Round Robin (RRR): Randomized round robin is an enhancement to the round robin policy where every "round" of requests is randomly distributed across the servers. In TraceSplitter, this policy (**TS-RRR**) corresponds to taking consecutive groups of $\frac{1}{f}$ requests (i.e., one round of requests) and randomly assigning them among the downscaled traces. RRR is a natural extension from the perspective of load balancing, but we are unaware of any work that applies RRR to trace downscaling. In our experiments, we find that TS-RRR is one of the best downscaling approaches.

Least Work Left (LWL): Least Work Left is a more sophisticated load balancing policy that uses request sizes to determine request assignment to servers. The approach works by estimating the amount of work left in each server and sending requests to the server with the least amount of work left. In TraceSplitter, this policy (**TS-LWL**) corresponds to assigning requests in the original trace to the downscaled trace with the least amount of work. That is, rather than balancing the number of requests in downscaled traces, TS-LWL balances the amount of work in each downscaled trace, which ideally would mimic the performance characteristics of a well-balanced original system. Sec. 3.4 describes how we estimate the work (i.e., service time) of requests, which only needs to be a rough approximation. Our experiments show that TS-LWL is the most realistic downscaling approach across a range of real-world and synthetic traces.

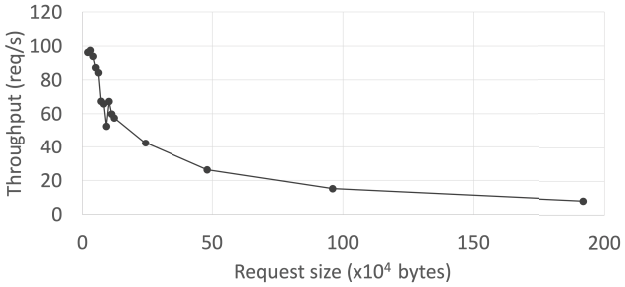


Figure 3: An example of our service time estimation technique.

3.4 Practical Considerations

Non-Integral Scaling: When the scaling is non-integral (i.e., $\frac{1}{f}$ is not an integer), our load balancing idea does not directly apply. We adapt the load balancing idea to partition the original trace into multiple downscaled traces as follows. We take the ceiling of $\frac{1}{f}$ as the number of partitions (i.e., $n = \lceil \frac{1}{f} \rceil$). Among these partitions, $(n-1)$ of them will each get f fraction of the original trace's requests, distributed according to the load balancing policy. The n^{th} partition gets the remaining requests, which is less than the desired fraction f. To compensate, we oversample requests from the other partitions. That is, we precompute a probability for when a request being added to one of the $(n-1)$ downscaled traces would be duplicated in the n^{th} trace. Sec. 4.4 demonstrates that this approach yields reasonable results.

Service Time Estimation: Ideally, the trace would contain service times measured on the original system. In practice, it may not be possible for trace providers to reveal such information for fear of disclosing proprietary design choices or violating customer privacy. Therefore, we assume a service time estimator will fill this gap. Specifically, the user-provided function $serviceTimeEstimator\ (r_i, o_i)$ estimates the time it takes to execute a request of size r_i. The estimation is only used to gauge the relative difference between request sizes. Thus, if performance scales linearly with r_i, then one could simply return r_i. However, in many systems, there are non-linearities in handling requests of various sizes. Fig. 3 shows an example of how the throughput in our system varies with the request size. We collect this data via a simple offline profiling wherein we measure the highest sustainable throughput for different request sizes. One can use such a process to collect a table of throughput values for various request sizes and then estimate service time as 1 / throughput(r_i). Since the estimator is used to quantify the relative differences between request sizes, we find a simple approach like this is sufficient. The user can also use latency in the original system (if available) or request size

as a proxy for service time estimation, although that would be less reliable than profiling. In the absence of reasonable service time estimation, the $serviceTimeEstimator\ (r_i, o_i)$ function can return a constant value, in which case TS-LWL would reduce to TS-RR or TS-RRR depending on the tie-breaking decision in TS-LWL.

4 Experimental Evaluation

4.1 Methodology

Server Application and Experimental Setup: We evaluate the efficacy of TraceSplitter in the specific context of a 3-tiered social networking application. The first tier is a replicated web server with a front-end load balancer based on Nginx [39] configured to use the *Least Connected* load balancing policy. We use Elgg [14], an open source social networking application written in PHP, for this web tier. The second tier is an in-memory Memcached [18] caching tier. The third tier is a MySQL [17] database containing the application state. Our client application for generating workloads for the server by replaying supplied traces is written in Java using the Faban [1] library. This client application employs a multi-threaded architecture wherein each thread performs the functionality of an individual user.

We deploy our entire setup in Amazon EC2 instances. Each component described above is placed in an Ubuntu-based Docker [32] container. We choose our experimental setup to ensure that no component, with the exception of the web tier, is a bottleneck. To this end, we generally use *m5.xlarge* instances (4 vCPUs, 16 GiB Memory) and ensure that the containers in an experiment are hosted in separate instances to avoid cross-component interference. We use *m5.8xlarge* instance (32 vCPUs, 64 GiB Memory) for the load balancer to ensure we are able to handle a large number of concurrent requests. In cases where our caching and database tiers need even more resources, we use the larger *m5.24xlarge* instances (96 vCPUs, 384 GiB Memory) instances [2].

Metrics: We focus on latency-based performance metrics (mean and several percentiles). We also consider the energy distance statistics metric to compare the entire latency distributions (Sec. 4.7). Unless otherwise specified, all the metrics reported are averaged over five runs and error bars show standard error. The standard errors in our graphs give an estimate of how far the sample mean is likely to be from the population mean. Simply put, large error bars indicate high variability in the results while small error bars indicate more confidence in our results. Metrics reported for partitioning-based scaling methods (TS-LWL, TS-RRR and RR) are based on results for all partitions.

Baselines: We choose four baselines to represent existing approaches. (i) For *Random* (Rand), we randomly sample requests based on the scaling factor. (ii) For *Average Rate Scaling* (AvgRate), we calculate the average arrival rate of requests within pre-determined time intervals (e.g., 10 sec) and multiply it by the scaling factor. New requests are generated by randomly picking requests within the interval until the scaled-down arrival rate is matched (i.e., equivalent to a Poisson process within an interval). The choice of time interval plays a crucial role in AvgRate, which we explore in Sec. 4.5. We acknowledge that a well-crafted workload performance model might perform well, but determining a good model for each type of workload is difficult for users, which is why we adopt a model-free approach. (iii) For *timespan scaling* (T-span), we scale the arrival time of the requests by multiplying by the scaling factor. (iv) Finally, *round robin* (RR) is a partitioning-based scaling policy described in Sec. 3.3. To accommodate non-integral scaling, TraceSplitter tweaks RR (as well as TS-LWL and TS-RRR) to oversample requests, which we evaluate in Sec. 4.4.

Original vs. Target Systems: To evaluate the difference between running in a larger original system vs. a smaller target system,we use our cluster as our *original system* and a smaller cluster as the *target system*. In most of our experiments, we reduce the number of nodes in the web tier by half and downscale the original trace by f=0.5. We provision the original system with 4 web tier nodes and the target system with 2 web-tier nodes. In both systems, we have 1 well-provisioned node for each of Memcached, MySQL, load balancer, and client. We collect metrics of interest from both clusters and use them to evaluate the efficacy of our downscaling.

In Sec. 4.4, we also show results from using f=0.75. TraceSplitter is expected to work well with any value of f, although care should be taken in selecting f. This is because trace downscaling is an inherently lossy process, and aggressive downscaling (lower value of f) may result in downscaled traces with hardly any requests. Hence, with very aggressive downscaling, the downscaled trace is likely to preserve less information from the original trace than with less aggressive downscaling. Even in that scenario, however, TraceSplitter is expected to perform better than current approaches due to it preserving all the requests across the collection of downscaled traces.

Traces: We use a combination of real-world/production and synthetic traces. Experiments with real-world traces validate the shortcomings of the state-of-the-art approaches we have pointed out previously while demonstrating the efficacy and robustness of our proposed approaches. We use synthetic traces in Sec. 4.3 to take a more in-depth look into why/when existing approaches fail and how TraceSplitter handles those cases. The synthetic traces help us explore a richer space of traces and establish that even in anomalous and extreme cases, TraceSplitter significantly outperforms existing approaches.

Summary: We present a summary of our salient findings in Tbl. 1, where the columns represent the different traces that we use and the rows represent different downscaling methods.

4.2 Results with Production Traces

We use two different production traces in our evaluation: (i) a web request trace from Microsoft's OneRF system, (ii) and a storage trace from SNIA IOTTA [3].

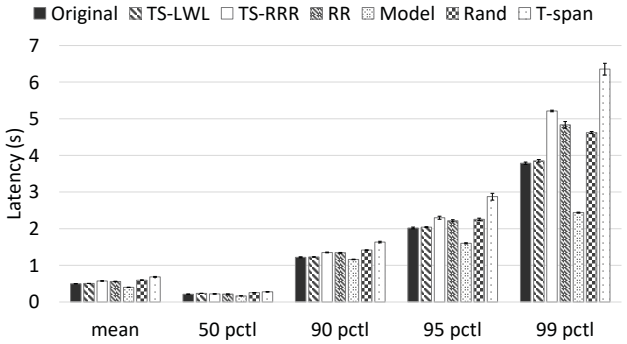
4.2.1 Microsoft Trace

The Microsoft OneRF trace was collected in February 2018 from a datacenter on the US East Coast. OneRF is a common web rendering framework that services web requests to Microsoft's storefront properties (microsoft.com, xbox.com, etc.). This production trace tracks the high-level web requests from users that arrive at OneRF, which connects to more than 20 different backend systems. For each request, the trace contains its arrival time and the identity of the backend that would service it.

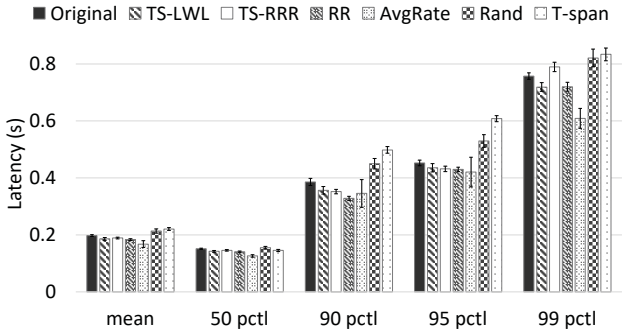
We design two different experiments that we label Case 1 and Case 2, each with about a 15 minute trace duration. Since we do not have access to the OneRF code base, for each case, we pick two backends from among the backends in the trace and associate them with two *request types* in our Elgg setup. Since the trace

Trace		Real World Trace		Synthetic Trace				
		Microsoft Trace	Storage Trace	Bursty	Temporal Pattern	Different Request Sizes	Arrival Pattern	Rare Event
Existing Approaches	Rand	✗	✓	✓	✓	✗	✓	✗
	AvgRate	✗	✗	✗	✗	✗	✓	✗
	T-span	✗	✗	✗	✗	✗	✗	✓
	RR	✗	✓	✓	✓	✗	✗	✓
Our Approaches	TS-RRR	✗	✓	✓	✓	✗	✓	✓
	TS-LWL	✓	✓	✓	✓	✓	✓	✓

Table 1: Summary of our experimental evaluation. ✓ indicates that the downscaling method closely matched the performance of the original trace while ✗ indicates that it did not.



(a) Microsoft trace: Case 1



(b) Microsoft trace: Case 2

Figure 4: Microsoft traces, see Sec. 4.2.1.

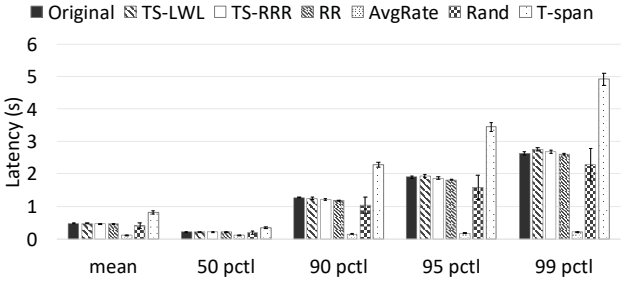


Figure 5: Storage trace, see Sec. 4.2.2.

itself does not contain complete information about request latency experienced on the original system, our focus is on reproducing important properties of the arrival process while making our own reasonable assumptions about request sizes. To this end, we choose significantly different service times between different request types and similar service times within a request type.

In Case 1, the larger request type comprises 5.7% of the requests and is on average 8.2× longer in terms of service time than the smaller request type. In Case 2, the larger request type comprises 2.8% of the requests and is on average 3.7× longer in terms of service time than the smaller request type. Case 1, therefore, has a higher skew between the two request types than in Case 2. The percentages of small/large requests come directly from the OneRF trace, and the choice of service time ratios mimics the heavy-tailed characteristics found in real world traces [5, 50]. The particular request sizes replayed are derived from the service time estimation in our system, shown in Fig. 3.

Case 1: We observe from Fig. 4a that T-span and AvgRate result in worse latencies than Original, with the gaps progressively growing at higher percentiles. In this case, the system is highly loaded especially during bursts, and T-span expands the high load bursts over a longer period of time than original, thus leading to higher tail latencies (detailed discussion in Sec. 4.3). AvgRate, on the other hand, offers lower latencies than Original due to averaging out the bursty periods (detailed discussion in Sec. 4.5). The remaining downscaling approaches more closely match Original, with TS-LWL performing the best across all percentiles.

Case 2: Case 2 has a lower skew than Case 1 between the resource needs of bigger and smaller request types, which results in the policies more closely matching the Original latencies as seen in Fig. 4b. The current downscaling approaches used in practice don't necessarily lead to the skewed results as shown in this case, but caution must be taken to appropriately construct experiments when downscaling. For example, AvgRate continues to offer slightly lower latencies as described in Case 1. Our proposed approach, TS-LWL, tends to more closely match the Original characteristics.

Overall, these results confirm that existing downscaling approaches sometimes suffer from the shortcomings we set out to address. These results also highlight how the fidelity of downscaling is intimately connected to certain workload properties (e.g., the skewness between resource needs of resource types in Cases 1 and 2), and our proposed TS-LWL approach is more robust in handling these properties, as demonstrated in Sec. 4.3.

4.2.2 Storage Trace

We also evaluate our work using publicly available SNIA IOTTA storage traces collected from enterprise storage traffic [29]. We select one of the traces that is appropriately sized for our cluster and exhibits interesting phenomena such as variation in arrival rates and burstiness. Although this trace comes from a different domain than Sec. 4.2.1, our careful approach to using the trace ensures

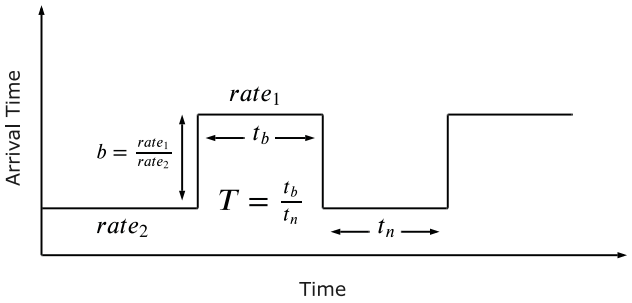


Figure 6: Parameters for synthetic trace generation.

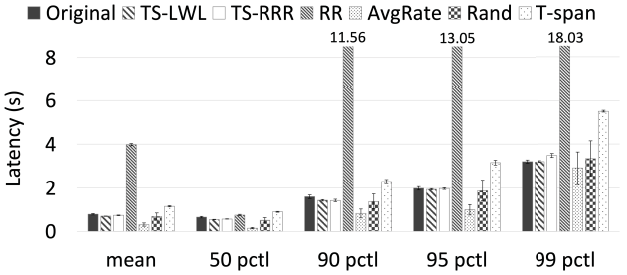


Figure 7: Combined synthetic trace, see Sec. 4.3.

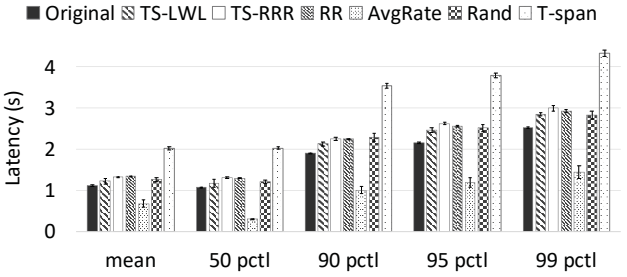


Figure 8: Effect of burstiness, see Sec. 4.3.2.

meaningful experiments. Specifically, request arrivals are the only information taken from this trace, and all requests are mapped to a single Elgg request type with a fixed request size. Thus, our focus is on the burstiness in the arrival process present in the trace.

Fig. 5 shows the latencies from this experiment. Similar to the Microsoft traces, we observe again that T-span results in higher latencies than Original while AvgRate results in lower latencies. This is due to T-span making the burst durations longer (details in Sec. 4.3.2) and AvgRate averaging out the effects of the bursts. Since all requests have the same size, TS-LWL, TS-RRR, and RR perform equally well in this experiment, as expected, with all of these policies closely matching the latency characteristics of the original system.

4.3 Synthetic Traces

In this section, we use synthetic traces to explore in detail a number of trace properties that are affected by downscaling. We identify and characterize the following properties: (i) fine-grained burstiness in the arrival of requests, captured by two parameters—burst duration and burst intensity; (ii) temporal patterns in the arrival of requests (i.e., how the arrival rate changes with time); (iii) variability in request sizes; (iv) any patterns in the order of request arrivals (e.g., periodicity); and (v) the extent and nature of rare request types present in a trace.

The experiments in this section demonstrate how these properties are affected by different state-of-the-art downscaling approaches. To accentuate these effects, we synthesize traces to amplify each of the properties individually. We also synthesize a trace that exhibits the combined effect of all the properties. Even when these properties are amplified, TraceSplitter performs well in downscaling the trace, whereas other approaches fail to realistically downscale traces with some of these properties.

Synthetic Trace Generation: The arrival process of our synthetic traces is generated as a combination of two Poisson processes with rates $rate_1$ and $rate_2$ (Fig. 6). A high value of burst intensity, $b = \frac{rate_1}{rate_2}$, helps generate a more bursty trace. We define t_b to be the duration of each burst and t_n to be the interval between two successive bursts. We also define $T = \frac{t_b}{t_n}$ to represent the ratio between the bursty and non-bursty time. We tune b, t_b and T to generate different types of bursts. For the request sizes, we generate large and small requests according to a $largeRequestFraction$ parameter (i.e., percentage of *large* requests) and a $serviceTimeRatio$ parameter (i.e., ratio of time to serve large vs. small requests). Depending on the trace, we either select the large or small request size randomly or according to a deterministic pattern (specified via $reqPattern$ parameter).

4.3.1 Combined Synthetic Trace

We start with a synthetic trace that shows the aggregate effect of all the factors listed in Sec. 4.3. The values of the trace parameters are: $b = 10$, $t_b = 4s$, $T = 0.5$, $largeRequestFraction = 5\%$, $serviceTimeRatio = 12$, $reqPattern = deterministic$. These parameters generate a trace with short high intensity bursts (due to values of b, t_b, and T) with a small fraction of very large requests (due to values of $largeRequestFraction$ and $serviceTimeRatio$) that vary in a deterministic pattern.

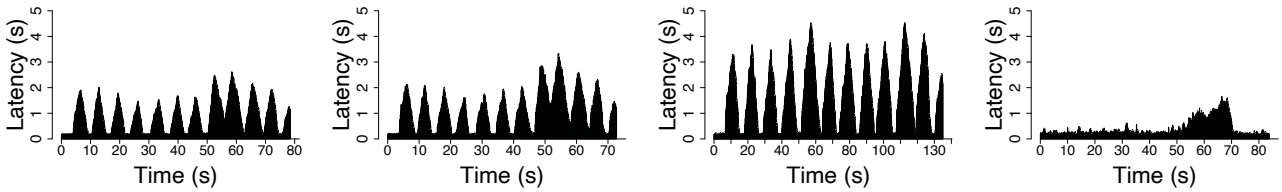
Fig. 7 shows that the mean and different percentiles of latency where RR, AvgRate, and T-span are significantly different from Original. The deterministic request size pattern causes RR to produce unbalanced downscaled traces, which leads to significantly skewed results in the downscaled experiments. Both of TS-LWL and TS-RRR mimic the latency of Original closely, with TS-LWL doing slightly better.

In the following subsections, we separate out which properties of the trace contribute to these performance behaviors for different downscaling approaches.

4.3.2 Effect of Burstiness

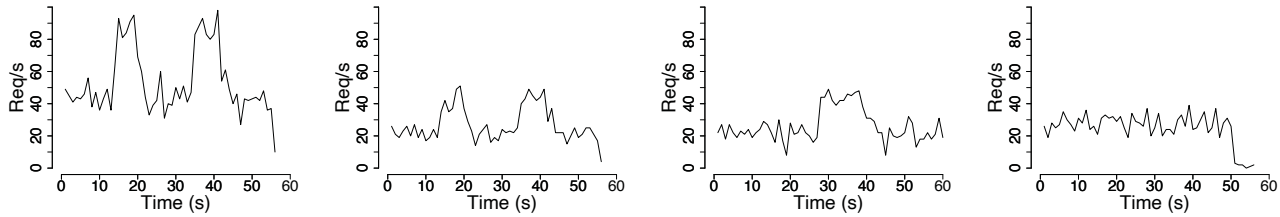
Fig. 8 shows the effect of downscaling on a bursty trace. We model the bursty trace where the values of the parameters are: $b = 5$, $t_b = 4s$, $T = 0.5$, $largeRequestFraction = 0\%$. This produces a trace that has short high intensity bursts (due to values of b, t_b, and T) and fixed request sizes.

AvgRate would have performed well (i.e., closer to Original) for bursty traces if the time-interval for averaging arrival rates is smaller than the burst duration. However, if the time-interval is longer than the burst duration (as in Fig. 8), then it can potentially average out the burst, resulting in lower latencies than in Original. This effect can be viewed by comparing the average request latency over time for Original and AvgRate in Fig. 9a and Fig. 9d respectively. TS-LWL, TS-RRR, Rand, and RR result in similar latency performance over



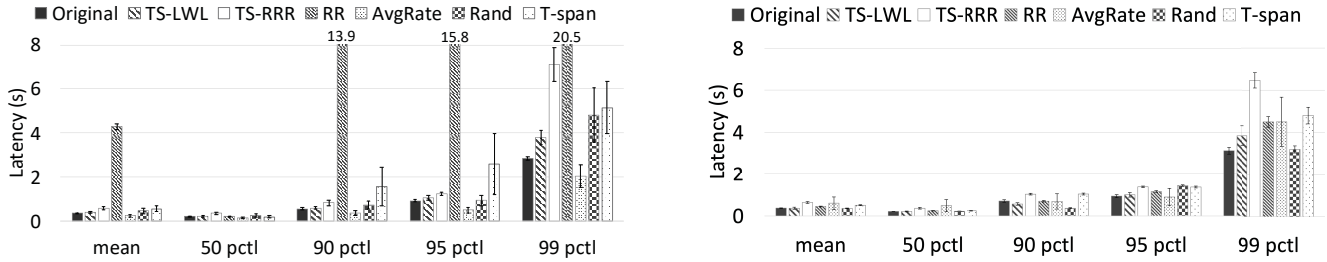
(a) Original. (b) TS-LWL. (c) T-span. (d) AvgRate.

Figure 9: Bursty trace request latencies, see Sec. 4.3.2.



(a) Original. (b) TS-LWL. (c) T-span. (d) AvgRate.

Figure 10: Effect of temporal pattern, see Sec. 4.3.3.



(a) Effect of different request sizes with deterministic pattern (b) Effect of different request sizes with random pattern

Figure 11: Effect of different request sizes, see Sec. 4.3.4.

time, closely matching Original. Thus, we only show the graph for TS-LWL (Fig. 9b) since TS-RRR, Rand, and RR look very similar.

One interesting outcome from the bursty trace is that the latencies increase for T-span as shown by comparing Fig. 9a and Fig. 9c. In T-span downscaling, we spread out the requests over a longer duration to reduce the arrival rate experienced by the system. For example, if we want to downscale a trace in half, we essentially double the interarrival time between requests. Intuitively, it seems like T-span should lower the load in half. However, the load on the system after scaling should be unchanged because while the arrival rate is halved, the number of nodes in the system is also halved. This can be problematic in bursty traces where the system is subjected to transient high load and could even be temporarily overloaded during the period of the burst. In these cases, the downscaled system may be overloaded, but with T-span, it is overloaded for a longer period of time, causing high latency for many requests for a longer period of time.

4.3.3 Effect of Temporal Pattern

We create a trace that contains a temporal pattern where arrival rates vary over time using the parameters: $b = 2$, $t_b = 12s$, $T = 1$, $largeRequestFraction = 0\%$. This creates a trace that alternates between high and low load periods (due to values of b, t_b, and T) and fixed-size requests. We show the arrival rate for every second for the first 60 seconds in Fig. 10. When the trace is scaled by T-span, as shown in Fig. 10c, we see that the temporal pattern has been stretched out where the high load and low load periods are twice as long. Depending on how the AvgRate time interval is set, AvgRate may average out the temporal pattern (Fig. 10d). Neither of these approaches accurately reflect the original temporal patterns. By contrast, TS-LWL, TS-RRR, Rand, and RR accurately downscale the trace; Fig. 10b shows the results for TS-LWL, and the other policies are similar.

4.3.4 Effect of Different Request Sizes

To show the effect of different request sizes, we create traces with a mixture of very large and small requests using the following parameters: $b = 1$, $largeRequestFraction = 5\%$, $serviceTimeRatio = 12$. This creates a trace with a high variation in request sizes (due to values of $largeRequestFraction$ and $serviceTimeRatio$) while having no burstiness or load variations. We create two such traces, one with a deterministic pattern of large requests and another with a random pattern. The point of this experiment is to demonstrate that approaches considering these differences in request sizes (i.e., TS-LWL) aren't negatively affected by request size variability. Fig. 11 shows that TS-LWL matches the latency of the original trace most closely in both traces. This is because TS-LWL is the only policy that considers request size, and since these traces have variations

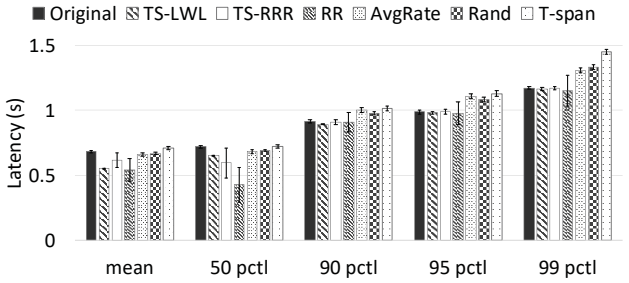


Figure 12: Effect of fixed arrival pattern, see Sec. 4.3.5.

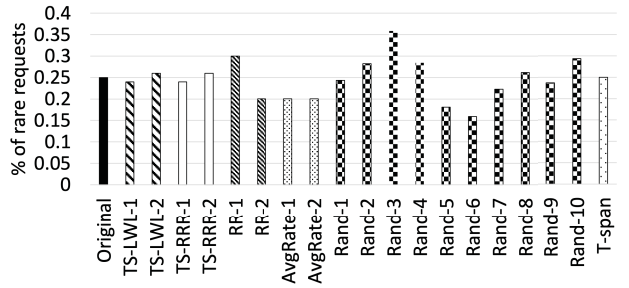


Figure 13: Rare events, see Sec. 4.3.6.

with respect to request sizes, TS-LWL outperforms the other policies. The deterministic pattern of the first trace causes RR to have a much higher latency than Original, as can be seen from Fig. 11a. This is because all the large requests in the original trace were placed in one downscaled trace. This uneven splitting of load causes the latency values to spike in one downscaled RR trace. While deterministic traffic patterns may appear in practice when users submit a sequence of related requests, we do not expect the traffic to typically be so adversarial. Fig. 11b demonstrates that RR behaves more like the other policies with traffic that has a random pattern of large requests. However, TS-LWL still matches the original latency most closely in these experiments.

4.3.5 Pattern in Arrival Process

If workloads have particular patterns in the arrival process, then they could affect deterministic partitioning policies like RR. To show that effect, we create a trace with two different request types that arrive in a deterministic pattern (e.g., one particular type of request arrives after the arrival of k requests from a different type) using the following parameters: $b = 1$, $largeRequestFraction = 10\%$, $serviceTimeRatio = 6$, $reqPattern = deterministic$. We do not introduce burstiness, and the $serviceTimeRatio$ is due to the differing service times for the request types. Because of the fixed pattern, RR ends up putting all of one request type in one downscaled trace. Since the request types take different amounts of time to execute, Fig. 12 shows that RR has a much lower latency than that of the original in the 50^{th} percentile. One other significant effect is that the two partitions of RR have significantly different performance, which does not happen for the policies of TraceSplitter.

4.3.6 Rare Events

Sometimes, preserving rare requests is important when downscaling, especially when these requests disproportionately contribute to resource usage and, therefore, performance. TraceSplitter guarantees that each request is included in at least one of the downscaled traces.

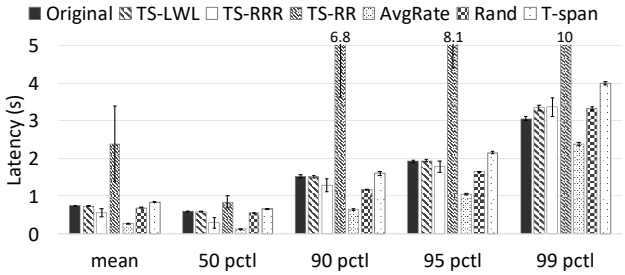


Figure 14: Non-integral downscaling, f=0.75, see Sec. 4.4.

However, because of the randomness in sampling, they could often be oversampled or undersampled in Random sampling. We consider a synthetic trace where 0.25% of the requests are large (and rare) requests and the rest are small requests.

We show the percentage of the rare requests preserved in each of the downscaled traces in Fig. 13. When we downscale the Original trace, TS-LWL, TS-RRR, and RR will preserve all of the rare requests across their partitions. T-span does not discard any requests, so it will also preserve all the rare requests. We randomly sample the Original trace 10 times and the percentage of rare requests in the downscaled trace shows variation. Random sampling drastically undersamples rare events in some cases and oversamples in other cases, as evident from Fig. 13. Although it also performs well in some cases, it can lead to misleading results if the practitioner ends up with one of the undersampled (or oversampled) downscaled traces for their experiments. Similar to Random sampling, AvgRate also picks requests randomly and will not preserve rare requests every time, which is evident in the two runs shown in the figure.

4.4 Non-Integral Scaling

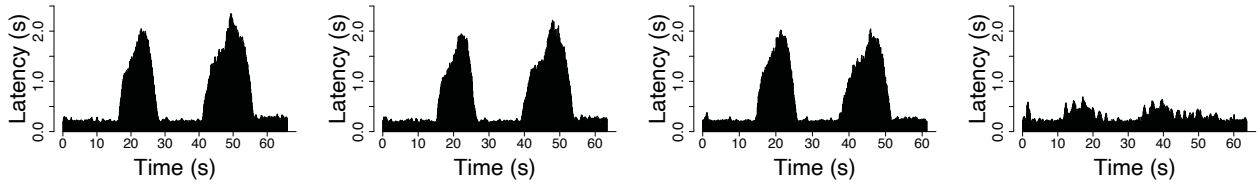
We consider a case of downscaling where f=0.75. The original trace is run in a system with 4 web servers while the downscaled traces are run in a system with 3 web servers. The workload is the same as the one described in Sec. 4.3.1. We show how various approaches compare in Figure 14. TS-LWL is able to match Original much more closely than TS-RR (which shows significant deviation and variance) as well as the baselines. TS-RR is the RR policy with TraceSplitter's approach for handling non-integral scaling.

4.5 Effect of Time Interval in AvgRate

The choice of the time interval is an important factor in AvgRate scaling. For the same original trace, we pick time intervals of 0.1 second, 1 second, and 10 seconds and compare the output latency in Fig. 15. The latency profiles of the trace downscaled by AvgRate with time intervals of 0.1 second (Fig. 15b) and 1 second (Fig. 15c) match the original trace (Fig. 15a) but that with time interval 10 seconds (Fig. 15d) fails to do so. This is because the time interval is larger than the burst length of the trace, as discussed in Sec. 4.3.2.

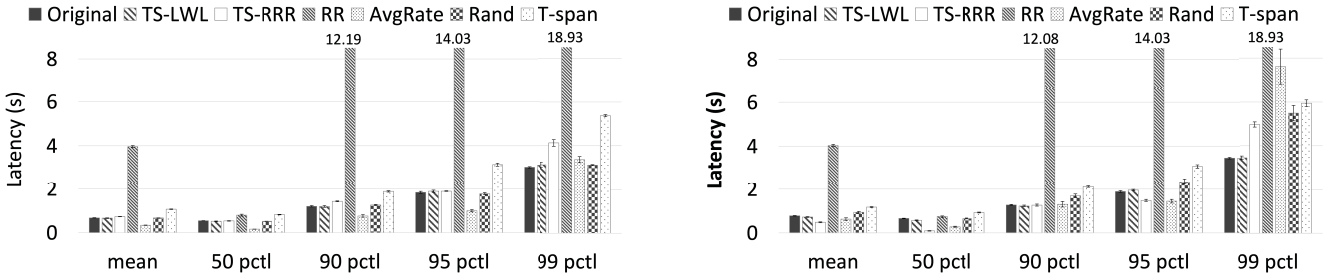
4.6 Sensitivity Study

Sensitivity to Randomness in Trace: We evaluate the sensitivity of our approach to the randomness in the synthetic trace generation to ensure our results are not specific to the exact trace generated. We create three versions of each trace using the same parameters so that they are statistically similar while not being exactly the same. As an example, Fig. 16 shows results from the other two traces that are generated from the same parameters as the trace used for Fig. 7. All



(a) Original. (b) Time interval = 0.1s. (c) Time interval = 1s. (d) Time interval = 10s.

Figure 15: Effect of time interval on AvgRate, see Sec. 4.5.



(a) Combined synthetic trace: Version-2 (b) Combined synthetic trace: Version-3

Figure 16: Different versions of combined synthetic trace

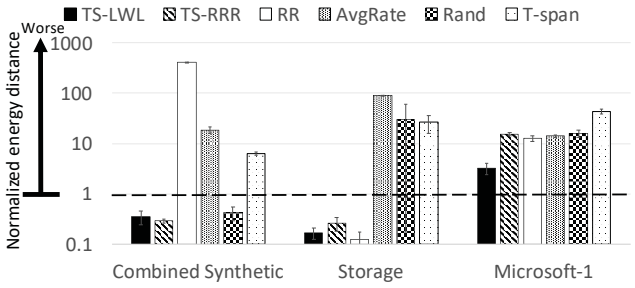


Figure 17: Energy distance

three sets of results show the same trends, so we believe our insights are not due to random chances in the trace generation.

Sensitivity to Service Time Estimation: TraceSplitter leverages a user-provided function for the service time estimation of requests in a given trace. Since TS-LWL does its partitioning based on the service time, incorrectly estimating service time could potentially affect the partitioning. As an extreme example, consider an estimator that indicates all requests have the same service time. It turns out that this is precisely the assumption in RR and TS-RRR. Hence, the performance of RR and TS-RRR gives us insight into how service time estimation affects TraceSplitter. We can see the effect of incorrect estimation of service time by observing RR and TS-RRR in Fig. 4a, Fig. 4b, Fig. 5, Fig. 12, Fig. 8, Fig. 11, and Fig. 7. They show that this incorrect estimation leads to performance degradation in RR, and incorporating randomness in that (as done in TS-RRR) improves the performance. In general, when there is high variation in request sizes, service time estimation is important.

4.7 Statistical Testing

We perform a rigorous statistical test to further evaluate how the performance from the different downscaled traces compares to the performance from the original traces. To that end, we compare the energy distance [41] between the distribution of latencies of the original trace with that of the downscaled traces. Both the original and the downscaled traces are run multiple times; they show some variability in their latencies due to the inherent noise present in any system. Let the empirical latency distribution of the original trace T be T_i and that of the downscaled trace t be t_j, where $i, j \in N_r$ = number of runs. We calculate the energy distance, $d(X, Y)$, of the latency distribution of a downscaled trace from each of the runs of the original trace, and take the minimum distance as the distance between the two: i.e., $dist_{t_j} = \min_{\forall i \in N_r} d\left(T_i, t_j\right)$. We combine the energy distance for each run of the downscaled traces by taking their mean. Since there is some variability in latencies obtained from multiple runs of the same original trace, we calculate the energy distance D_{ij} between every two runs (i^{th} and j^{th} run) of the original trace, and use the maximum D_{ij} as a normalizing factor. The normalized energy distance is then $\frac{\sum_{j \in N_r} dist_{t_j}}{N_r} \times \frac{1}{D_{ij}}$.

A value of 1 or less for the normalized energy distance would indicate a trace whose latency distribution closely matches that of the original, with equal or less variability than comparing multiple runs of the original trace against themselves. As the value increases from 1, the trace deviates from the original. We plot the normalized energy distance for our combined synthetic trace (Sec. 4.3), storage trace (Sec. 4.2.2), and Microsoft trace (case 1) (Sec. 4.2.1) in Fig. 17. We see that TS-LWL and TS-RRR perform well for the combined synthetic and storage traces. All downscaling methods perform worse than 1 for the Microsoft trace, though TS-LWL is closest to 1 among them.

It is important to note that energy distance is a very stringent measure for equivalence because it considers the entire distribution (all percentiles, not just the median, 95^{th} percentile, etc.). Since trace downscaling is an inherently lossy process, we cannot expect the downscaled traces to meet this expectation, but this experiment shows that our approach outperforms the current state-of-the-art in synthetic and real traces.

Policy	Overpro-visioning % suggested by policy	SLO violations (out of 5 trials)	Average P99 (ms)	Average web tier node count
AvgRate	0	5	18147	6.21
T-span	40	5	2757	8.56
Rand, TS-RRR	60	5	557	9.65
TS-RR, TS-LWL	70	0	425	10.34

Table 2: A sample result from our autoscaling case study.

5 Case Study: Autoscaling a Web Application

The evaluation above demonstrates TraceSplitter's efficacy at preserving the latency metrics of an application on an original system, when running downscaled traces on a smaller target system. In this section, we show that TraceSplitter can be used to inform *policy decisions* for an application, by leveraging its preservation of latency metrics. Specifically, we use TraceSplitter to optimize a policy parameter on a smaller target system w.r.t a performance SLO, and show that the optimized parameter delivers the expected performance in the original system.

The application we consider is an autoscaler that scales a web application tier based on the arrival rate of requests in the system. The autoscaler uses the Square Root Staffing rule [24] to determine the number of web tier nodes required, with an overprovisioning factor of $c > 0$: i.e., if the Square Root Staffing rule requires k nodes, the autoscaler allocates (c +1) × k nodes. Our goal is to find the minimum value of c that allows us to meet a performance SLO of 99^{th} percentile latency (P99) < 500 ms. For the web application we use Wikimedia with a MySQL database as the backend, and front it with an Nginx load balancer. The autoscaler controls the number of web-tier nodes and informs Nginx of any changes so that it load balances over the current set of nodes.

We drive the Wikimedia application with a synthetic workload that captures several of the key properties identified in our evaluation, including a diurnal pattern and periodic sharp bursts. We vary the period of the diurnal pattern and the frequency and magnitude of the bursts to generate a set of 86 different traces. For each trace, we first run it while varying the overprovisioning in increments of 0.1, i.e., $c = 0.1, 0.2$, etc., and determine the optimal value of c that meets our P99 latency SLO. We observe that some traces require very little overprovisioning or too much overprovisioning; since these traces fail to distinguish the different downscaling approaches or run into experimental setup limitations, respectively, we limit our focus to traces with optimal c ∈ [0.3, 0.8]. This results in selecting 30 of those 86 traces for further investigation.

We then downscale each trace with a scaling factor of 0.5 using each of the different downscaling approaches, and run these traces while varying c as above. It is worth noting that in this case-study, there are no fixed original and target system sizes like the ones used in Sec. 4 since the autoscaler automatically adjusts the size of the system according to the arrival rate, leading to different number of web application nodes being used for the original and downscaled trace. We then compare the optimal value of c determined by each downscaling approach with the ground truth value determined using the original system. Across all of the traces, TS-LWL performs as well or better than the other downscaling approaches, in that it finds an optimal value of c that is equal (or closest) to the optimal value in the original system. Tbl. 2 shows an example from one of the traces. In this example, only TS-LWL and TS-RR find an overprovisioning value that meets the P99 latency SLO ($c = 0.7$); the remaining approaches all underprovision.

The reason AvgRate performs poorly is because it assumes the arrival rate follows a Poisson process (similar to the assumption made by the Square Root Staffing rule), and thus fails to capture the bursts in the original trace. T-span does poorly because it stretches out the bursts, making the autoscaler think it has more time than it actually does to react to the bursts. In contrast, TS-LWL and TS-RR preserve the bursts faithfully and hence yield an overprovisioning value that is high enough to accommodate the bursts.

6 High Level Takeaway

There are a number of takeaways from our investigation into trace downscaling. T-span performs poorly because it distorts the temporal properties. A less intuitive effect of T-span is that when dealing with bursty traces, it stretches the bursts to last longer, causing the system to be overloaded for a longer period of time than the original trace. Model-based approaches are in general good candidates for downscaling, but require a lot of attention, effort, and insights from the practitioners to create an accurate model. Since the models need to be adapted for different traces and use cases depending on the metric of interest, it is difficult and time consuming to do this correctly. In many cases, since creating the model is orthogonal to the main research objective, researchers may not invest as much effort in model creation, leading to inaccurate models that might miss many key characteristics. Random sampling can potentially oversample or undersample rare events due to its inherent randomness.

To address all of these problems, we propose TraceSplitter, which can be used to preserve the arrival patterns of the requests and their performance (i.e., latency) characteristics. However, we do not claim TraceSplitter to be a silver bullet for all downscaling problems. Being an inherently lossy process, downscaling will always miss out on something in a particular downscaled trace. What TraceSplitter provides is a framework for preserving all requests if the user runs all the downscaled traces one after another. That way, one can compensate for not having a big enough experimental setup by spending more time running experiments. And if one does not have the time to do this, they can still run a random subset of the downscaled traces generated by TraceSplitter and obtain accurate results. Of the TraceSplitter policies, we find that TS-LWL and TS-RRR perform the best in preserving arrival patterns and performance characteristics, as demonstrated by our experiments and our case study.

7 Conclusion

TraceSplitter introduces a new framework for downscaling traces based on the idea of viewing the trace scaling problem as a load balancing problem. Using this idea, we implement multiple trace downscaling algorithms in TraceSplitter and perform an extensive evaluation across a wide range of real-world and synthetic traces. Our results demonstrate how current practices for downscaling traces are inaccurate for some workload patterns. For example, current practices can mask effects such as burstiness and temporal variations. Our new method (TS-LWL) achieves the best downscaling accuracy across our experiments. TraceSplitter is available as an open-source tool at https://github.com/smsajal/TraceSplitter to benefit the community and improve the standard practices in working with traces.

Acknowledgments

We thank our shepherd Jonathan Mace and the anonymous reviewers who provided constructive and helpful feedback. We also thank Ataollah Fatahi Baarzi for his help with the experimental setup. This research was supported in part by National Science Foundation grants 1717571 and 1909004.

References

[1] [n.d.]. faban.org. Accessed: Jan. 20, 2020.
[2] [n.d.]. https://aws.amazon.com/ec2/instance-types/. Accessed: May 25, 2020.
[3] [n.d.]. SNIA IOTTA trace repository. http://iotta.snia.org/. Accessed: May 15, 2020.
[4] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. 2012. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*. IEEE, 204–212.
[5] Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. [n.d.]. Waiting Game: Optimally Provisioning Fixed Resources for Cloud-enabled Schedulers. ([n. d.]).
[6] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. 2009. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing* (San Diego, California) *(HotCloud'09)*. USENIX Association, USA, Article 12, 5 pages.
[7] D. Breitgand, Z. Dubitzky, A. Epstein, O. Feder, A. Glikson, I. Shapira, and G. Toffetti. 2014. An Adaptive Utilization Accelerator for Virtualized Environments. In *2014 IEEE International Conference on Cloud Engineering*. 165–174.
[8] Binlei Cai, Rongqi Zhang, Laiping Zhao, and Keqiu Li. 2018. Less provisioning: A fine-grained resource scaling engine for long-running services with tail latency guarantees. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–11.
[9] M. Carvalho, F. Brasileiro, R. Lopes, G. Farias, A. Fook, J. Mafra, and D. Turull. 2017. Multi-dimensional Admission Control and Capacity Planning for IaaS Clouds with Multiple Service Classes. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 160–169.
[10] Marcus Carvalho, Daniel A. Menascé, and Francisco Vilar Brasileiro. 2017. Capacity planning for IaaS cloud providers offering multiple service classes. *Future Gener. Comput. Syst.* 77 (2017), 97–111.
[11] M. Carvalho, D. Menascé, and F. Brasileiro. 2015. Prediction-Based Admission Control for IaaS Clouds with Multiple Service Classes. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 82–90.
[12] Zheyi Chen, Jia Hu, Geyong Min, Albert Zomaya, and Tarek El-Ghazawi. 2019. Towards accurate prediction for high-dimensional and highly-variable cloud workloads with deep learning. *IEEE Transactions on Parallel and Distributed Systems* (2019).
[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.
[14] Cash Costello. 2012. *Elgg 1.8 social networking*. Packt Publishing Ltd.
[15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 499–510. https://www.usenix.org/conference/atc15/technical-session/presentation/delgado
[16] Ludwig Dierks, Ian Kash, and Sven Seuken. 2019. On the cluster admission problem for cloud computing. In *Proceedings of the 14th Workshop on the Economics of Networks, Systems and Computation*. 1–6.
[17] Paul DuBois and Michael Foreword By-Widenius. 1999. *MySQL*. New riders publishing.
[18] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.
[19] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2019. Stochastic resource allocation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 122–136.
[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog
[21] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*. Ieee, 9–16.
[22] Anubhav Guleria, J Lakshmi, and Chakri Padala. 2019. QuADD: QUantifying Accelerator Disaggregated Datacenter Efficiency. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 349–357.
[23] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. 2018. Elmem: Towards an elastic memcached system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 278–289.
[24] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.
[25] Joseph L. Hellerstein. 2020. Google Cluster Data. http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.
[26] Vatche Ishakian, Raymond Sweha, Jorge Londono, and Azer Bestavros. 2010. Colocation as a service: Strategic and operational services for cloud colocation. In *2010 Ninth IEEE International Symposium on Network Computing and Applications*. IEEE, 76–83.
[27] Congfeng Jiang, Guangjie Han, Jiangbin Lin, Gangyong Jia, Weisong Shi, and Jian Wan. 2019. Characteristics of co-allocated online services and batch jobs in internet data centers: a case study from Alibaba cloud. *IEEE Access* 7 (2019), 22495–22508.
[28] Ayaz Ali Khan, Muhammad Zakarya, Rajkumar Buyya, Rahim Khan, Mukhtaj Khan, and Omer Rana. 2019. An energy and performance aware consolidation technique for containerized datacenters. *IEEE Transactions on Cloud Computing* (2019).
[29] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*. 1–11.
[30] J. Liu, H. Shen, A. Sarker, and W. Chung. 2018. Leveraging Dependency in Scheduling and Preemption for High Throughput in Data-Parallel Clusters. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 359–369.
[31] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.
[32] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.
[33] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. 2010. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev.* 37, 4 (March 2010), 34–41. https://doi.org/10.1145/1773394.1773400
[34] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. 2016. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. 147–156.
[35] Fanny Pascual and Krzysztof Rzadca. 2018. Colocating tasks in data centers using a side-effects performance model. *European Journal of Operational Research* 268, 2 (2018), 450–462.
[36] M Pavlidakis, S Mavridis, Y Sfakianakis, C Symeonidou, N Chrysos, and A Bilas. [n.d.]. Flexy: Elastic Provisioning of Accelerators in MultiGPU Servers. ([n. d.]).
[37] Daniel Perez, Kin K Leung, et al. 2020. Fast-Fourier-Forecasting Resource Utilisation in Distributed Systems. *arXiv preprint arXiv:2001.04281* (2020).
[38] Safraz Rampersaud and Daniel Grosu. 2016. Sharing-aware online virtual machine packing in heterogeneous resource clouds. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2016), 2046–2059.
[39] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* 2008, 173 (2008), 2.
[40] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–13.
[41] Maria L Rizzo and Gábor J Székely. 2016. Energy distance. *wiley interdisciplinary reviews: Computational statistics* 8, 1 (2016), 27–38.
[42] Ivan Rodero, Hariharasudhan Viswanathan, Eun Kyung Lee, Marc Gamell, Dario Pompili, and Manish Parashar. 2012. Energy-efficient thermal-aware autonomic management of virtualized HPC cloud infrastructure. *Journal of Grid Computing* 10, 3 (2012), 447–473.
[43] Stefano Sebastio, Michele Amoretti, Alberto Lluch Lafuente, and Antonio Scala. 2018. A Holistic Approach for Collaborative Workload Execution in Volunteer Clouds. *ACM Trans. Model. Comput. Simul.* 28, 2, Article 14 (March 2018), 27 pages. https://doi.org/10.1145/3155336
[44] Stefano Sebastio, Michele Amoretti, and Alberto Lluch Lafuente. 2014. A Computational Field Framework for Collaborative Task Execution in Volunteer Clouds. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Hyderabad, India) *(SEAMS 2014)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/2593929.2593943
[45] Stefano Sebastio and Giorgio Gnecco. 2017. A green policy to schedule tasks in a distributed cloud. *Optimization Letters* 12 (10 2017). https://doi.org/10.1007/s11590-017-1208-8
[46] Alina Sîrbu and Ozalp Babaoglu. 2015. Towards Data-Driven Autonomics in Data Centers. In *International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE Computer Society, Cambridge, MA, USA. http://www.cs.unibo.

it/babaoglu/papers/pdf/CAC2015.pdf

[47] Kui Su, Lei Xu, Cong Chen, Wenzhi Chen, and Zonghui Wang. 2015. Affinity and conflict-aware placement of virtual machines in heterogeneous data centers. In *2015 IEEE Twelfth International Symposium on Autonomous Decentralized Systems*. IEEE, 289–294.

[48] Amoghavarsha Suresh and Anshul Gandhi. 2019. Using Variability as a Guiding Principle to Reduce Latency in Web Applications via OS Profiling. In *The World Wide Web Conference*. 1759–1770.

[49] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–14.

[50] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[51] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845. http://www.globule.org/publi/WWADH_comnet2009.html.

[52] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM Trans. Auton. Adapt. Syst.* 3, 1, Article 1 (March 2008), 39 pages. https://doi.org/10.1145/1342171.1342172

[53] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[54] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 246–258. https://doi.org/10.1145/3357223.3362728

[55] Ellen W Zegura, Mostafa H Ammar, Zongming Fei, and Samrat Bhattacharjee. 2000. Application-layer anycasting: A server selection architecture and use in a replicated web service. *IEEE/ACM Transactions on networking* 8, 4 (2000), 455–466.

[56] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. 2013. Harmony: Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 510–519.

[57] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. 2014. Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud. *IEEE Transactions on Cloud Computing* 2, 1 (2014), 14–28.

[58] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.