TraceUpscaler: Upscaling Traces to Evaluate Systems at High Load

Sultan Mahmud Sajal
sxs2561@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

Trace replay is a common approach for evaluating systems by rerunning historical traffic patterns, but it is not always possible to find suitable real-world traces at the desired level of system load. Experimenting with higher traffic loads requires *upscaling* a trace to artificially increase the load. Unfortunately, most prior research has adopted ad-hoc approaches for upscaling, and there has not been a systematic study of how the upscaling approach impacts the results. One common approach is to count the arrivals in a predefined time-interval and multiply these counts by a factor, but this requires generating new requests/jobs according to some model (e.g., a Poisson process), which may not be realistic. Another common approach is to divide all the timestamps in the trace by an upscaling factor to squeeze the requests into a shorter time period. However, this can distort temporal patterns within the input trace. This paper evaluates the pros and cons of existing trace upscaling techniques and introduces a new approach, *TraceUpscaler*, that avoids the drawbacks of existing methods. The key idea behind *TraceUpscaler* is to decouple the arrival timestamps from the request parameters/data and upscale just the arrival timestamps in a way that preserves temporal patterns within the input trace. Our work applies to open-loop traffic where requests have arrival timestamps that aren't dependent on previous request completions. We evaluate *TraceUpscaler* under multiple experimental settings using both real-world and synthetic traces. Through our study, we identify the trace characteristics that affect the quality of upscaling in existing approaches and show how *TraceUpscaler* avoids these pitfalls. We also present a case study demonstrating how inaccurate trace upscaling can lead to incorrect conclusions about a system's ability to handle high load.

EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0437-6/24/04...$15.00
https://doi.org/10.1145/3627703.3629581

CCS Concepts: • **General and reference** → **Experimentation**; **Performance**; **Measurement**.

Keywords: Analysis, Testing and Verification of Systems; Cloud Computing; Measurement, Analysis, Workloads

ACM Reference Format:
Sultan Mahmud Sajal, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. 2024. *TraceUpscaler*: Upscaling Traces to Evaluate Systems at High Load. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3627703.3629581

1 Introduction

Computer systems researchers and practitioners often need to test and experiment with their systems under realistic traffic conditions. There are two broad approaches for generating workload traffic. First, closed-loop traffic sends requests (aka jobs) to the system after previous requests have completed. This is useful for understanding the maximum throughput characteristics achievable by the system (i.e., system capacity) since completing requests faster will cause more requests to be sent to the system. Second, open-loop traffic represents requests generated from external entities (e.g., users) over time and is most applicable for user-facing systems (e.g., web servers). In these cases, throughput (i.e., completion rate) is equal to the arrival rate, assuming a stable system, so the primary performance characteristic is latency, which can significantly be affected by the arrival time and request characteristics (e.g., size, type). Significant research effort has been made to design systems to optimize latency, particularly the tail latency characteristics [2, 26, 57, 63, 79, 104]. Our work focuses on how to evaluate these open-loop scenarios to accurately portray these latency characteristics. Our research shows that failing to properly replay traffic patterns could significantly skew results by multiple orders of magnitude, leading to incorrect conclusions about a system's ability to handle various traffic conditions.

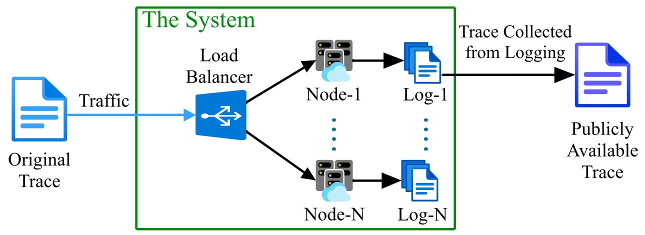


Figure 1. Typically, publicly available traces from production environments are collected from only a subset of the nodes.

One gold standard for performance evaluation is to utilize traces of production traffic to represent real-world traffic patterns. Throughout this paper, we use the term *trace* to represent a list of requests/jobs, where each request is accompanied by an arrival timestamp and other relevant request parameters. Generating experimental traffic according to a trace is known as *trace replay*, and it involves sending the requests based on the time intervals specified by the trace. Trace replay is a straightforward and simple approach for generating the traffic to a system in a realistic and reproducible manner, but it relies upon the experimenter to have accurate traces for their specific needs. Companies have made some traces publicly available [25, 46, 53, 100, 106, 108], but they are often collected from a subset of nodes in the system (Fig. 1). Hence, they do not represent the whole traffic experienced by the system (i.e., only a fraction of the load). As a result, the system load from a trace may not be high enough if the evaluation platform is larger than the portion of the system recorded by the trace.

Thus, it is often necessary to adjust the load to appropriately fit the size of the evaluation platform. We refer to modifying the load of a trace as *trace scaling*, and experimenters often invoke ad-hoc mechanisms that they tend to describe briefly and vaguely or not at all. Recently, TraceSplitter [91] has demonstrated that these common approaches can be inaccurate, but it only considers downscaling, where the desired load is lower than the load of the trace. The focus of this paper is on upscaling traces, where the desired load is higher than the load of the trace. While downscaling involves removing known, existing requests from a trace, upscaling requires adding new requests to increase the load, which necessitates fabricating requests that are representative of the input trace.

There are three predominant use cases for upscaling traces. First, one would need to upscale traces when the evaluation platform is larger than (the subset of) the production system where a trace is collected. Second, practitioners need to test hypothetical scenarios where the load has grown. For example, if the load doubles in the next year, is the system able to handle the load, or do practitioners need to address some scalability bugs, or do they need to take a step back and develop a more scalable design? One could answer these questions by capturing and upscaling a trace to test this hypothetical scenario. Third, it is helpful to characterize the performance of a system by showing how latency changes as a function of load. Latency vs. load graphs are traditionally generated by synthesizing traces at various loads (e.g., with a Poisson arrival process) and plotting the resulting latency. With upscaling, one could take a production trace and scale it to various load levels to have a more realistic characterization of the system performance, with all the bursty traffic patterns and request peculiarities of the real-world trace.

Upscaling Approaches: In practice, there are two common approaches to upscaling. First, one could divide all the timestamps by *an upscaling* factor to squeeze the requests within a shorter timespan, thus increasing the load. We call this Timespan Scaling (Tspan). Our work will show how this can distort temporal patterns within the trace, which would misrepresent performance effects. Second, one could count the number of requests within time intervals (e.g., every 60 sec [39, 77] or every 1 sec [3, 15]) and then multiply these numbers by a scaling factor. In effect, this tracks and upscales the average rate of requests over time, so we call this AverageRateScaling. Importantly, this approach involves generating requests at a higher rate, which requires some model for generating the requests. One of the more common approaches would be to use Poisson process to generate timestamps. However, it is easy to generate unrealistic traces if the time interval is too large or too small, and it is hard to gauge an appropriate time interval when upscaling a trace since there is no indication of the upscaling being realistic or not. Another choice users can make is to sample from an empirical distribution of request types/sizes/parameters. This can potentially lead to distorted caching effects, depending on how the request parameters are selected. The Repeat approach, which simply repeats requests at the same timestamps from the Input trace, also suffers from distorted caching effects due to repeating the requests.

TraceUpscaler is our new upscaling approach that maintains temporal patterns in the Input trace without altering any caching patterns. Fig. 2 illustrates our approach compared to other common approaches. To avoid altering caching behaviors, *TraceUpscaler* generates upscaled requests using the same request parameters from the Input trace in the same order. This is the same as the Tspan approach, except instead of dividing the timestamps by a scaling factor, *TraceUpscaler* uses the same arrival timestamp for a string of consecutive requests in the upscaled trace. That is, the first timestamp in the Input trace is used for multiple requests in the upscaled trace, and the second timestamp in the Input trace is used for the next set of requests in the upscaled trace. For example, Fig. 2 illustrates how all the arrivals in the trace generated by *TraceUpscaler* match the arrival times in the Input trace. This avoids distorting temporal patterns since micro-bursts and load variations in the Input trace are amplified at the same time in the upscaled trace. To the best of our knowledge, we are the first to propose this upscaling strategy. The key idea behind our novel approach is to decouple the arrival timestamps from the request data (i.e., request parameters)

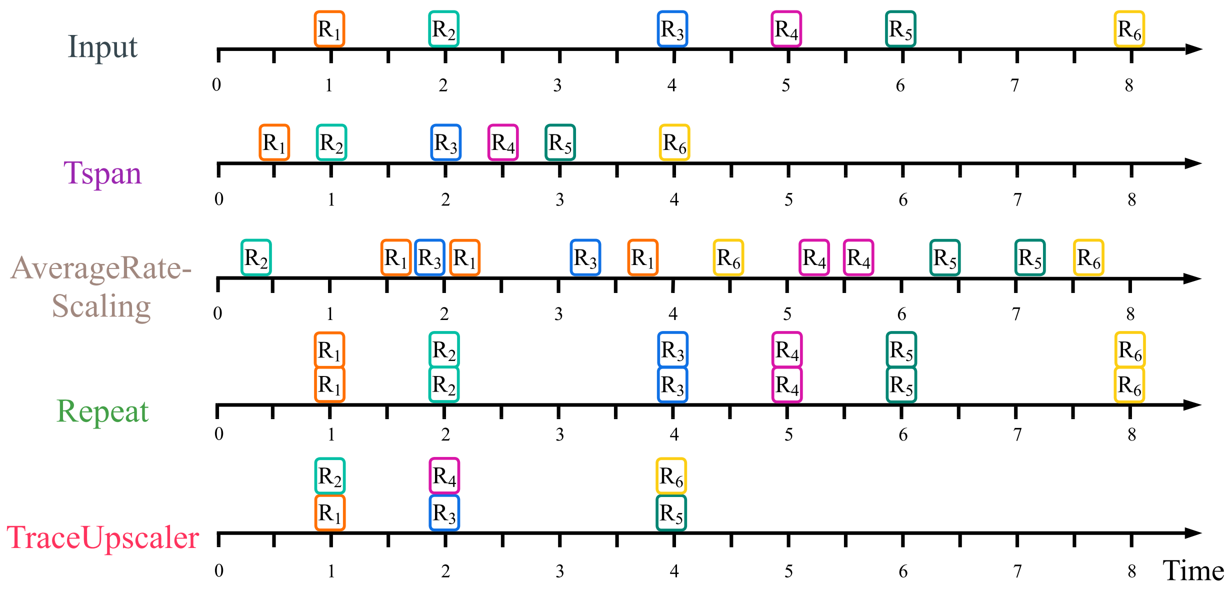


Figure 2. Illustration of upscaling techniques for scaling factor, f = 2. The requests are denoted by R_1, R_2, ..., R_6, and each request is also marked with a unique color. Tspan condenses requests into a shorter timespan, thereby increasing load at the cost of distorting temporal patterns. One side effect is the duration for Tspan (and *TraceUpscaler*) becomes shorter than that of the Input trace as more requests are needed for the higher load. AverageRateScaling generates requests at a higher arrival rate with requests sampled from the Input trace. This can possibly distort the temporal pattern and relative ordering of the requests compared to the Input trace. *TraceUpscaler* repeats timestamps similar to Repeat to maintain temporal patterns, but uses requests from the Input trace in the same order to preserve caching effects.

and repeatedly use the same arrival timestamps when upscaling. The insight behind this is twofold: (i) adhering to arrival timestamps allows us to preserve important temporal patterns from the Input trace, and (ii) preserving the relative ordering of the requests allows us to maintain the caching effects from the Input trace. Our evaluation shows that this simple and elegant approach effectively overcomes the limitations of prior approaches.

Contributions:

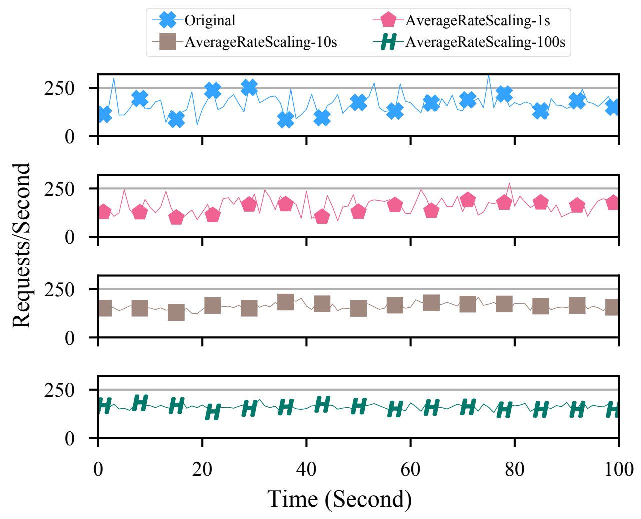
- We identify pitfalls with common upscaling approaches used in practice via evaluations with both production and synthetic traces (Sec. 5). This includes a case study (Sec. 5.5) where current upscaling techniques inaccurately portray an overload both in terms of its duration and magnitude, leading to incorrect conclusions about the system's ability to handle this overload.
- We develop *TraceUpscaler*, a novel upscaling approach that realistically maintains temporal patterns and caching effects from the Input trace, overcoming the limitations of existing approaches.
- *TraceUpscaler* is available as an open-source tool at https://github.com/smsajal/TraceUpscaler.

2 Background and Related Work

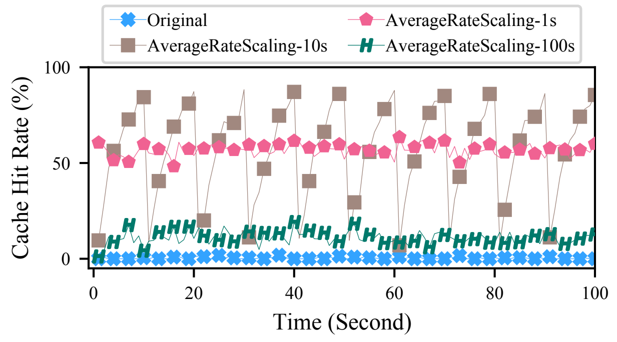
While scaling traces to achieve a desired load is a common practice in systems research, many works do not describe precisely how they perform their scaling [11, 81, 102, 103, 113, 122]. This makes it hard to reproduce results and verify the soundness of the trace scaling. For the works that do describe their scaling methodology, we can classify the approaches into three broad categories: (i) Model-Based Scaling, (ii) Timespan Scaling, and (iii) Node Removal.

2.1 Model-Based Scaling (AverageRateScaling)

One common approach for scaling traces is to generate request arrival times and parameters based on some model involving one or more "important" workload characteristics. For example, request parameters are often sampled from empirical distributions and arrival times are often based on using well-known arrival processes (e.g., Poisson process) with the arrival rates configured based on averages from the original trace. However, generating accurate models is difficult and requires significant time and effort [110]. Lublin et al. [70] have explored in detail the cases where models are preferred and important factors for developing a realistic model. There have been efforts in more accurate model-based upscaling, which include using time-series prediction techniques for generating synthetic arrival times [18, 37, 66, 67, 85, 96] and fitting arrival times into distributions so that practitioners can sample from them [55, 60]. While fitting arrival times into empirical distributions to generate statistically similar arrivals is a good choice to capture temporal patterns, it can still end up distorting the cache access patterns because of reusing existing requests to generate more requests (pitfall



(a) Effects of different time intervals in preserving short-term bursts in synthetic traces upscaled by AverageRateScaling.



(b) Difference in cache hit rate with different time intervals in synthetic traces upscaled by AverageRateScaling.

Figure 3. Effect of time interval on the quality of upscaling by AverageRateScaling. This example uses an upscaling factor f = 2 where the Input trace to AverageRateScaling is a fraction (1/2) of the Original trace.

shown in Fig. 3b). Prior research has also focused on different trace characteristics other than arrival times [71, 89, 110]. The Google trace [46, 86, 106, 111] has been used to generate models [19–21, 59, 76, 84, 93–95, 99, 120, 121] that focus on capturing key characteristics from the trace. Similarly, there are multiple models [30, 34] generated from the Azure trace [25] and other models [10, 17, 29, 42, 49, 87] generated from various other production traces [6, 27, 69]. However, experimenters trying to replay a trace typically lack both the time and expertise to generate such models just to upscale a trace for an experiment. In most cases, upscaling is orthogonal to the actual problem being solved, which leaves little room for spending effort on modeling trace characteristics to accurately scale them. As a result, inappropriate models and parameters are often chosen for simplicity, which can lead to non-representative trace scaling and wrong conclusions about system performance. By contrast, *TraceUpscaler* reuses request and arrival data from the trace itself, so it is not bound to any particular model and does not suffer from any modeling assumptions.

AverageRateScaling: In this paper, we will focus on the following model-based scaling methodology: (i) divide the original trace into fixed time intervals, (ii) calculate the average arrival rate for each time interval, (iii) scale the rates by the scaling factor, and (iv) generate a new upscaled trace where the timestamps are randomly generated from a Poisson process with the upscaled arrival rates and request parameters are randomly sampled from the empirical distributions for each time interval. We label this approach as AverageRateScaling, and this is a popular approach to upscale traces [5, 15, 28, 56, 75, 117].

There are three important aspects of this approach. First, we use a time-varying Poisson process where each time interval has a Poisson process. A Poisson process is a common, well-known arrival process that has a single arrival rate parameter that can easily be scaled, but it is not the best at representing bursty traffic. Making it time-varying helps in capturing some of the burstiness, but it is not perfect. Fitting traces to more complex models of arrival patterns is possible, but experimenters prefer simple upscaling techniques to avoid skewing results from modeling peculiarities.

Second, we empirically sample requests from each time interval from the Input trace to represent how request parameters correlate with each other over time. We have not experimented with using a different time interval for empirical request parameter distributions as that introduces an additional parameter to tune. We acknowledge that our approach is rather simple, and one can construct a more sophisticated model that can work better than this, but it would require more time, expertise, and effort from the experimenter. We hope that prior works have put sufficient thought into upscaling approaches, but the side effects of upscaling are nuanced, so a goal of this work is to demonstrate potential pitfalls from simple approaches that researchers commonly apply in practice.

Third, the choice of the time interval can cause the upscaled trace to not be representative of the original trace. If the time interval is too long, short-term bursts are eliminated by the averaging as shown in Fig. 3a. On the other hand, if the time interval is too short, requests from the empirical distribution will be reused, leading to higher hit rates than the Original trace as shown in Fig. 3b.

For a trace, the smallest time-interval possible would be the granularity of the timestamp recorded in the trace (milliseconds or seconds for typical web workloads). This would basically repeat all the requests at their exact timestamps, as many as needed to reach the desired load in the trace. We call this Repeat, and our evaluation (Sec. 5) shows that although it performs reasonably well in stateless systems, it distorts caching effects in stateful systems. Ultimately, there is no "right" time interval, and it is non-trivial to figure out what is appropriate for a given trace.

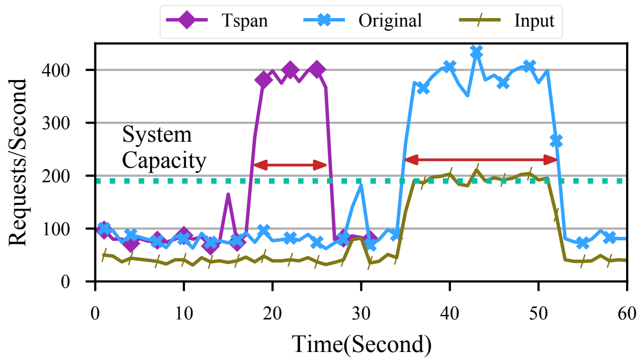


Figure 4. Impact of Tspan upscaling on temporal patterns in a trace. This example uses an Input trace derived from half of the Original trace. Tspan increases the load of the Input trace by a factor of 2 to match the load of the Original trace, but the overload duration is shorter compared to the Original trace.

2.2 Timespan Scaling (Tspan)

Another popular trace scaling approach is to scale the inter-arrival times between requests. That is, all the arrival times are divided by the scaling factor to shorten the timespan between requests in the trace [14, 52, 98, 107, 107, 112, 114], and we call this approach timespan scaling. Timespan scaling (Tspan) is simple and easy to understand, but it can distort the temporal patterns in the trace, which skews performance in unrealistic ways. For example, temporary overload periods in the input trace would be shortened in the upscaled trace, as shown in Fig. 4; this would subject the system to a shorter period of overload compared to the input trace. This can lead to misleadingly optimistic conclusions (see case study in Sec. 5.5), which can result in incorrect system management or design decisions

2.3 Node Removal

Lastly, an orthogonal approach to increasing load in the system is to instead remove nodes/machines from the system without scaling the trace [110, 119]. However, one of the use cases of upscaling is to evaluate higher loads where there may be scalability bottlenecks, and evaluating with a smaller cluster will not demonstrate the desired effect. Additionally, the granularity of increasing load is limited by the non-fractional nature of removing nodes from the system. Furthermore, practitioners also need to consider the constraints, fragmentation, and heterogeneity present in the nodes/machines for a proper node removal strategy [110], which adds complexity to the systems evaluation process.

2.4 Trace Downscaling

Downscaling traces to reduce load is a common practice among practitioners where Sampling [16, 23, 68, 72, 82, 83, 101], AverageRateScaling [3, 39, 43, 109, 118], and Tspan [13, 38, 78, 109] have been used to generated downscaled traces. A recent work, TraceSplitter [91], has explored the potential pitfalls associated with these approaches and proposed a novel downscaling technique. However, upscaling is a more difficult problem than downscaling because downscaling involves removing requests from existing traces, whereas upscaling involves introducing new requests into the trace. This is similar to how image upscaling is harder than image downscaling due to the addition/removal of data. Creating new requests while preserving the realism of the original trace is difficult and requires caution and in-depth knowledge in selecting timestamps and request characteristics appropriately, which is the goal of this work.

2.5 Other Works

Designing realistic experiments for performance evaluation of real-world computer systems has consistently garnered interest among practitioners [50, 61, 65, 90, 92, 110]. Prior works have looked at understanding computer benchmarking practices [47, 61], exploring common flaws of reporting data in experiments specific to performance evaluation in parallel computing and High Performance Computing (HPC) systems [7, 31, 33, 48, 51, 70], providing in-depth analysis of performance variability in computer systems [73], and generating synthetic testbeds for reproducible experiments [80, 116]. Recently, DITTO [64] has proposed an automated framework to clone cloud applications, capturing important application characteristics (e.g., kernel operations, application logic behavior, high-level performance metrics, and I/O and network activities) without exposing original application logic. While DITTO focuses on realistic cloning of an application, we focus on the realistic scaling of a trace that is to be replayed in the application.

There have been prior efforts in the context of storage traces, which include generating models from storage workloads [12, 40, 97, 105, 123], analyzing the challenges and design of a time accurate storage benchmark [4], developing accurate trace replay tools [1, 44, 54, 105, 115], facilitating reuse of traces collected from one type of storage to another storage system [74], and modifying traces collected from older hardware to newer hardware [22, 36, 52, 62, 74, 107, 114, 124]. These works predominantly focus on changing the request characteristics to suit various storage hardware in closed-loop workloads, whereas our work focuses on scaling the load and arrival patterns in open-loop workloads.

3 Design and Implementation

3.1 Goals and Scope

The goal of *TraceUpscaler* is to realistically increase the load of an existing trace collected from a real-world system. *TraceUpscaler* generates an upscaled trace based on an input trace and upscaling factor, f $(1 < f < \infty)$[1], where f is the ratio between the desired load and the current load in the trace.

[1]Generating scaled trace with f < 1 is downscaling, which is out of scope of *TraceUpscaler*.

This is significantly affected by trace arrival patterns (i.e., arrival times) as well as how requests are generated in the upscaled trace.

The scope of *TraceUpscaler* is open-loop [45] latency-sensitive applications, which cover a broad category of cloud systems. End users submit *requests* or *jobs* to the system and expect a response within a reasonable amount of time. Our work aims to provide a realistic upscaling approach that captures the trace characteristics impacting latency, both in the mean and tail percentiles (e.g., 99th percentile). Latency is an important performance metric in these systems and can capture the aggregate impact of many effects such as caching, whether the system is overloaded, etc.

Where TraceUpscaler works: Our work focuses solely on increasing the load (i.e., arrival rate) of a trace to evaluate *what-if* scenarios where load is increased. *TraceUpscaler* is designed to preserve the input trace characteristics, so that the upscaled trace is 'similar' to the input trace but at a higher load. To preserve caching behaviors, our approach utilizes the exact same requests from the input trace in the same order. To preserve temporal access patterns, our approach uses the same timestamps from the input trace as well, but replicates them to increase load. We assume that the input trace quality is representative of what the user wants to evaluate in their experiments. In that regard, *TraceUpscaler* does not make any specific decisions for scenarios such as handling missing data, abnormalities in trace, etc. and leaves those to the discretion of the user.

Our evaluation demonstrates how *TraceUpscaler* can use a trace collected from a subset of nodes to generate an upscaled trace that exhibits similar latency characteristics to the Original trace sent to the cluster. Practitioners can also use *TraceUpscaler* to generate traffic to evaluate scenarios such as how faster machines would react under higher load, how an increased number of nodes can handle a higher load, etc.

Where TraceUpscaler does not work: *TraceUpscaler* is not designed to forecast any changes in trace characteristics that occur due to the increased traffic. For example, in some cases (e.g., social networks), the increased traffic might be caused by a hot event, translating to more users requesting the same data, leading to an increase in cache hit rate for those requests. Conversely, increased traffic can also potentially increase the working set of accessed requests, leading to a decrease in cache hit rate compared to the current trace. Our approach only focuses on preserving the caching effect, and does not consider the possible caching behavior changes that can occur with increased traffic. Hence, *TraceUpscaler* does not work when the traffic at higher load is expected to differ significantly from the input trace, and users need to model these differences for their particular workload.

TraceUpscaler may not work at very high upscaling factors. Our evaluation includes experiments with scaling factors up to 5, and we have not tested significantly higher scaling factors. When scaling factors are too high, the specific micro-bursts within the input trace will be magnified. Upscaling is also limited by the data available in the input trace, so there may not be enough requests to achieve the desired upscaling factor and upscaled trace duration.

3.2 Key Ideas

The first key idea is to frame the trace upscaling problem in terms of a trace reconstruction problem. *TraceUpscaler* constructs an upscaled trace by reusing the same timestamps and requests from the Input trace. This way, we maintain realism by not introducing any synthetic data generated by a model.

The second key idea is to separate the arrival timestamp generation from the request parameter generation. For requests in the upscaled trace, we use the existing request parameters from the Input trace in the same order to avoid distorting caching behaviors. Caching systems are predominantly affected by the order in which requests arrive rather than the exact time that they arrive. Our approach thus preserves the request ordering from the Input trace, while allowing load to be increased through the arrival timestamp generation.

3.3 Our Proposed Method: *TraceUpscaler*

Our proposed upscaling approach, *TraceUpscaler*, generates an upscaled trace where the request parameters exactly match the request parameters in the Input trace in the same order. Only the request arrival timestamps differ where the first timestamp in the Input trace is used across f consecutive requests in the upscaled trace and the second timestamp in the Input trace is used for the following f consecutive requests. Thus, we preserve the same request ordering from the Input trace[2] along with its caching characteristics while increasing the load by having f times the number of requests arriving within a given time period. Using the same timestamp for multiple upscaled requests allows us to maintain the temporal patterns from the Input trace. For example, bursts within the Input trace would appear as bursts in the upscaled trace at the same time and for the same duration. Fig. 2 shows an example of how we reconstruct an upscaled trace with an upscaling factor f = 2. Though simple, we find through our evaluation that this approach effectively overcomes some of the limitations of prior approaches. When dealing with non-integral upscaling factors (i.e., f is not an integer), *TraceUpscaler* uses a timestamp for $\lfloor f \rfloor + 1$ consecutive requests with probability $(f - \lfloor f \rfloor)$ or $\lfloor f \rfloor$ consecutive requests otherwise. This results in an upscaling factor of f on average throughout the trace. We conduct experiments in Sec. 5.3 to demonstrate that

[2]For the requests in the upscaled trace that have the same arrival timestamp, the ordering matches the Input trace so that the trace replayer will first pick the earlier request from the Input trace. As with any concurrent system, there's a chance for requests to occur out of order, but our trace ordering biases the requests to follow the order in the Input trace. We discuss the sensitivity of this in Sec. 6.

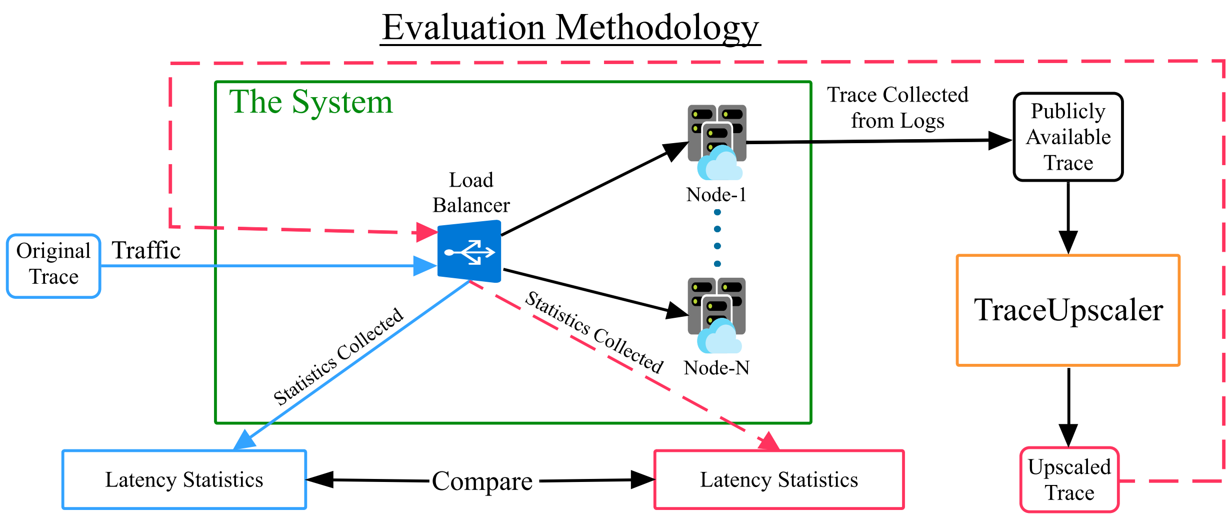


Figure 5. Evaluation methodology for comparing upscaling techniques. We replay an Original trace to collect (i) latency statistics, and (ii) an Input trace from a subset of nodes in the system. The Input trace represents what a company may publicly release, and we upscale it to match the load of the Original trace. We then replay the upscaled trace and compare its latency statistics with the latency statistics from running the Original trace.

TraceUpscaler performs well when dealing with non-integral scaling.

To develop our approach, we take inspiration from Tspan (for using only the requests from the Input trace) and Repeat (for reusing only the timestamps from the Input trace). Tspan and *TraceUpscaler* use the same requests in the same order, but the former's approach to upscaling the timestamps results in skewed temporal patterns as described in Sec. 2.2. Repeat uses the same timestamps as *TraceUpscaler*, but it also repeats the requests, which impacts caching systems since the repeated requests are easily cachable. By separately reconstructing timestamps and request data, *TraceUpscaler* produces upscaled traces that preserve the temporal and caching behaviors of the input trace, combining the best aspects of Tspan and Repeat. Our evaluation in Sec. 5 demonstrates that *TraceUpscaler* is the best upscaling approach in preserving latency characteristics.

One important caveat of using *TraceUpscaler* (and Tspan) is that they require a longer input trace duration than the output upscaled trace. For example, producing a 5-minute upscaled trace with 2× load would require a 10 minute or longer input trace. Since production systems can often collect traces over a long period of time, often multiple days [25, 46, 86, 106, 111], we do not expect this limitation to create a serious barrier against using *TraceUpscaler* in practice.

4 Evaluation Methodology

One of the most challenging aspects of evaluating trace upscaling is determining whether the upscaling was done realistically. Fig. 5 shows the design of our evaluation methodology. To evaluate realism, we need some ground truth trace that we can compare upscaled traces against, and as Fig. 5 shows, we use an *Original* trace as the ground truth. We replay this trace in the system and collect log and latency statistics from the system. From the logs, we identify the traffic experienced by each node in the system. We then take the traffic experienced by a subset of the nodes and upscale them with an *upscaling factor* (f) to match the load in the *Original* trace. We then replay the upscaled trace in the system and record the latency statistics from the system as before. Finally, we compare the latencies from the *Original* and upscaled traces to determine how closely they match. We determine the realism of upscaling by seeing how closely the latency from an upscaled trace matches that from the *Original* trace. A good upscaling technique should mimic the performance of the *Original* trace even though the Input trace to the upscaler is only a subset of the *Original* trace.

4.1 Metrics

Our main performance metric is latency as it is of primary interest in the context of latency-sensitive open-loop applications. We compare the latency distributions between the Original and upscaled traces to see which upscaling approach matches the Original trace behavior most closely. For our system with caching, we also look at the Cache Hit Rate as a means of explaining the latency characteristics. All metrics reported are averages of five runs (unless otherwise specified) and error bands demonstrate standard errors. Most of our experiments use a 2× scaling factor by default, and we explore other scaling factors in Sec. 5.3 and Sec. 5.4.

4.2 Comparison Approaches

This section describes the comparison upscaling approaches, which are depicted in Fig. 2.

Repeat: In this approach, we repeat all requests and arrival timestamps by the *upscaling* factor. Possible variations of this approach include adding small offsets (random, deterministic, etc.) to the timestamps. From our initial evaluations, we have found that this does not substantially impact the performance of Repeat. Hence, we decided to focus on the simplest version with repeated timestamps/requests.
AverageRateScaling: This upscaling technique is described in Sec. 2.1, and we select a time interval of 10s to not be too large or too small. There could be a better time interval, but analyzing the input trace to fine tune the time interval is beyond the scope of this paper.
Tspan: In this approach, we divide all arrival timestamps by the *upscaling* factor, as described in Sec. 2.2.
Fold: In this approach, we split the whole trace into f consecutive parts, and then we *overlap* these parts to create an upscaled trace. For example, we can upscale by a factor of 2 by overlapping the first half of the trace with the second half of the trace. This intuitive method seems promising in theory, but suffers from averaging out micro-bursts in the trace because the micro-bursts within each part do not overlap exactly, and we demonstrate this in Sec. 5.1.1 and Sec. 5.5. This may be desirable if one wanted to experiment with the effects of overlapping trace segments, but our goal is to mimic the behavior seen in the Input trace exactly as-is.

4.3 Applications and Cluster Hardware

We conduct our evaluations across two different web application systems — one stateful (DeathStarBench) and one stateless (MediaWiki) — as representatives of real-world applications. These are run on the nodes (i.e., VMs) in Fig. 5. We develop a simple trace replayer as the client application that generates web requests to the applications and measures performance statistics. Both the applications and client program are deployed in Azure using *Standard Ds v5* series VMs (Tbl. 1) running Ubuntu 18.04.

VM Size	vCPU	Memory (GiB)
Standard_D2s_v5	2	8
Standard_D4s_v5	4	16
Standard_D8s_v5	8	32

Table 1. VM types used in experiments.

DeathStarBench: We use the social network application from DeathStarBench [35], a modern distributed end-to-end benchmark with 30 microservices (e.g., application logic, web servers, databases). We use the *read_home_timeline* workload, where the homepage is populated by 64 posts from a user's social network timeline. We populate the social network with the Reed98 [88] dataset, which creates a total of 962 users in the system, and we insert 250 random posts per user, creating a total of 240,500 posts in the system. We add caching support through a lightweight HTTP reverse proxy cache, Varnish [41], which is placed in front of the whole DeathStarBench application. We use this setup as an example of a stateful application in our evaluation.

For DeathStarBench, we use two *D4s* nodes for the majority of the experiments where we upscale one of the node's traffic by a factor of 2. We balance the load between the two nodes using an Nginx load balancer employing the Weighted Round-Robin load balancing policy. For the Non-Integral Scaling experiments (Sec. 5.3) and Extreme Scaling experiments (Sec. 5.4), we experiment with other upscaling factors besides 2 by adjusting the weights to send different proportions of traffic (e.g., 2:1 ratio) to the nodes. For example, the trace collected from the node receiving 2/3 of the requests is used for the 1.5× upscaling experiment, while the trace collected from the node receiving 1/3 of the requests is used for the 3× upscaling experiment. Since these nodes receive different proportions of traffic, we naturally vary the node size using combinations of *D8s*, *D4s*, and *D2s* VM types. Across the experiments, the DeathStarBench setup experienced a peak rate of 352 requests/second with an average between 116.8-172.75 requests/second. Each request accesses 64 posts where each post contains 256+ characters.
MediaWiki: We deploy MediaWiki [8], which is a multi-tier web application used to run Wikipedia. Our workload targets the stateless web application tier. Each request accesses a webpage with the response being the webpage content.

For MediaWiki, we use 16 application nodes, each being deployed in *D4s* VM types. We balance the load between these nodes using an Nginx load balancer employing the Least Connections load balancing policy. To avoid bottlenecks in the database layer, we create 3 database instances, each being deployed in a *D8s* VM type. Each database hosts the data for 5-6 MediaWiki nodes, and we use a read-only workload, so we do not synchronize the databases. In our experiments, the complete MediaWiki setup experienced a peak arrival rate of 1397 requests/second with an average 383 requests/second.
Trace Replayer: Our trace replayer is responsible for generating the traffic to the applications. It is implemented as a multi-threaded application written in Java to send HTTP requests according to the trace supplied to it. The client application is highly configurable to mimic real-world multi-client traffic in accordance to the trace supplied to it. The client does not require many resources and is deployed on one *D2s* VM. It is not a bottleneck at the scale of our system (2-16 nodes).

4.4 Traces

Our evaluation uses traces collected from a real-world production system (Microsoft OneRF) and synthetically generated traces. We use arrival timestamps from the Microsoft OneRF traces to demonstrate how real-world arrival patterns are impacted by *TraceUpscaler* and current upscaling approaches. The synthetic traces help us explain the reasons behind the shortcomings of current approaches.

Microsoft OneRF traces: The Microsoft OneRF traces [9] were collected from a datacenter on the US East Coast in February 2018. OneRF is a common webpage rendering framework used to serve a wide-range of Microsoft's storefront properties including news (msn.com) and online retail software stores (microsoft.com, xbox.com). This production trace collects high-level web requests from users arriving at OneRF, which are served by more than 20 different backend systems, such as product catalogues, recommender systems, user entitlement systems, etc. The trace contains the arrival time of requests at millisecond granularity.

To create the Original trace (Fig. 5), we use arrival timestamps from the Microsoft OneRF trace with request parameters from the benchmarks/applications as described in Sec. 4.3 since the OneRF trace does not have request parameter data relevant to our evaluation applications (e.g., no request size information). We then run the Original trace in each application and collect traces from a subset of nodes to represent the Input trace. Each upscaling approach uses only the subset of data in the Input trace to generate an upscaled trace, and we compare the performance when running the upscaled trace and Original trace on the same system.

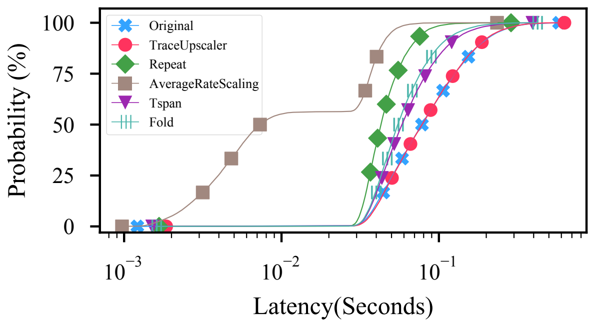
In our experiments, we pick 15 minute traces from the different backends that have a suitable load for our cluster (i.e., Original trace load is neither too high nor too low). Since Tspan and *TraceUpscaler* require a higher number of input requests to generate an upscaled trace, the upscaled traces from these policies are much shorter in duration than the Input trace. Thus to evaluate all policies for the same duration, we run each of the upscaled traces for 5 minutes. These durations are too short to evaluate long-term caching effects, and we only evaluate short-term (minutes) caching effects where Repeat and AverageRateScaling fail.

Synthetic traces: We synthesize bursty and non-bursty arrival patterns to isolate trace characteristics that impact the quality of the upscaled traces. We generate the arrival times in the trace by using a Markov Modulated Poisson Process (MMPP) [32], which randomly switches between multiple Poisson Processes. By controlling the transitions between the Poisson processes, we can create bursty and non-bursty traces, which serve the purpose of our investigation. The request parameters are randomly generated for each evaluation application as described in Sec. 4.3.

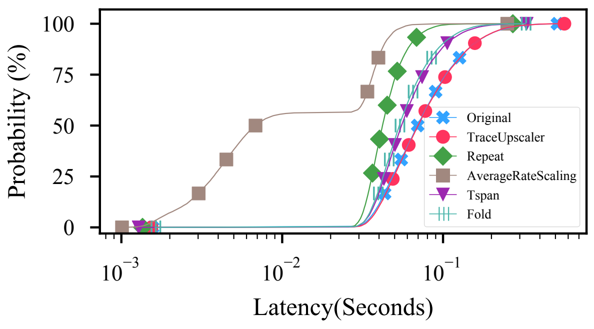
5 Experimental Results

5.1 Results using Arrival Times from Production Traces

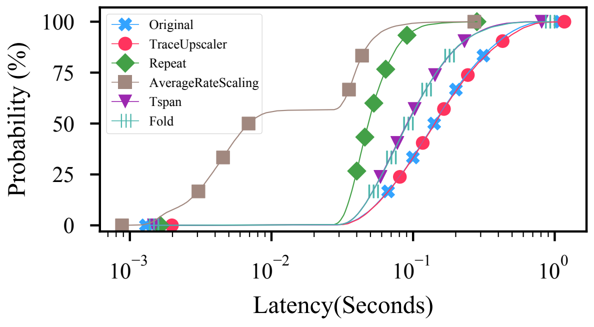
5.1.1 Results from Stateful Setup. Fig. 6 shows results from experiments in the stateful DeathStarBench setup. We create 3 different traces using arrival timestamps from the Microsoft OneRF trace. Across all three cases, *TraceUpscaler* is the closest to Original (ground truth), so it is the most accurate trace upscaling approach in representing latency characteristics. Repeat performs poorly since repeated requests can



(a) Microsoft OneRF Trace: Case 1



(b) Microsoft OneRF Trace: Case 2



(c) Microsoft OneRF Trace: Case 3

Figure 6. Comparison of different upscaling techniques in the stateful system (DeathStarBench system with an added frontend cache — details in Sec. 5.1.1), using traces with arrival timestamps from the Microsoft OneRF trace. The closer the latency for an upscaling technique is to Original, the better it is in realistic upscaling.

directly return data from the cache, hence bypassing the performance impacts experienced by the Original traffic from generating the dynamic webpage content. AverageRateScaling performs poorly since short term bursts are smoothed out when computing the average rates. Furthermore, requests can be repeated when request parameters are sampled from the empirical request distribution, which leads to higher cache hit rates compared to the Original trace. This also contributes

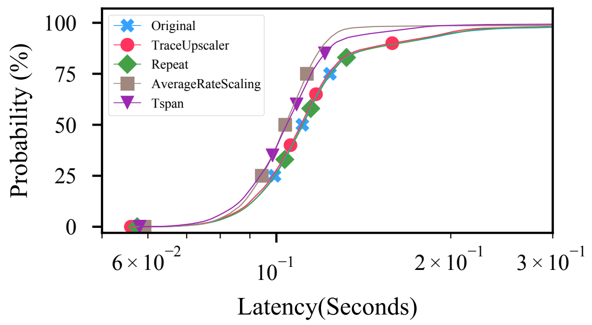


Figure 7. Comparison of different upscaling techniques in the MediaWiki system where the stateless application tier is the limiting resource (details in Sec. 5.1.2), using a trace with arrival timestamps from the Microsoft OneRF trace. The closer the latency for an upscaling technique is to Original, the better it is in realistic upscaling.

to the consistently lower latency of AverageRateScaling compared to the Original trace. Tspan is closer to Original than Repeat and AverageRateScaling because it does not distort the caching effects, but it still suffers from failing to preserve short term burst and overload characteristics in the trace. We explore this effect in greater detail with synthetic traces in Sec. 5.2. Lastly, Fold is similar to Tspan since it overlaps bursty and non-bursty periods, resulting in less severe burstiness.

5.1.2 Results from Stateless Setup. To explore the effects without a cache, we next conduct experiments in the stateless MediaWiki application using a trace with arrival timestamps from the Microsoft OneRF trace. The results from that experiment are shown in Fig. 7. Once again, *TraceUpscaler* closely matches the latency characteristics of Original, but this time, Repeat also works well. This is because repeating requests introduces an appropriate amount of work at an appropriate time. So surprisingly, even a simple upscaling policy that repeats requests can work well assuming the application is agnostic to whether a request is new or repeated (i.e., stateless systems). As before, AverageRateScaling performs poorly due to smoothing out the bursts, and Tspan skews the temporal burst and overload patterns, both of which change the latency characteristics.

5.2 Results using Synthetic Arrival Times

We synthetically generate traces in this section to explore how trace characteristics and system attributes impact the accuracy of upscaling techniques. In our investigation, we identify two characteristics that significantly affect upscaling techniques: (i) short-term burstiness, and (ii) request caching in the system.

5.2.1 Impact of Burstiness. Short-term burstiness can cause temporary strain in the system. Depending on the duration and load of the short-term burst, it can temporarily overload the system and impact latency, especially at high percentiles.

We generate and use two different traces with short-term burstiness in DeathStarBench and MediaWiki, and their results are shown in Fig. 8a and Fig. 8b, respectively. AverageRateScaling fails to faithfully recreate short-term burstiness in the trace, as it averages out the bursts in the trace. This results in lower latency than the Original trace.

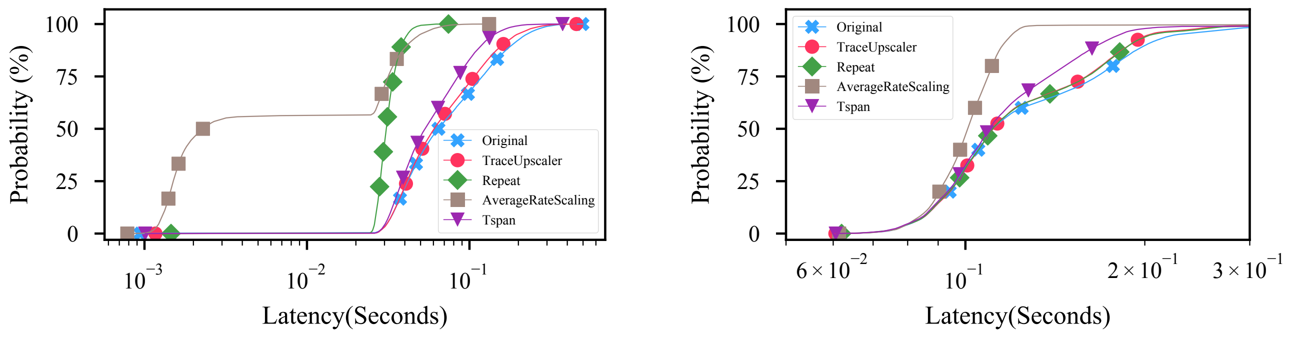
Tspan distorts the temporal pattern of the trace by shrinking the burst times in the trace. To understand this effect, Fig. 9 shows the queue length characteristics in the Original trace and how Tspan fails to replicate that whereas *TraceUpscaler* succeeds in replicating the queueing behavior. Due to the shortening of overload periods in Tspan, the requests do not spend much time in the queue. Subsequently, the queue length over time is much smaller in Tspan than in Original. As a result, latencies with Tspan are significantly lower than Original. *TraceUpscaler* preserves the overload characteristics in the trace, which translates to matching the queue length of the Original trace. To confirm this effect, Fig. 10a shows the results from a synthetic trace generated from a Poisson arrival process, which is far less bursty.[3] Tspan is much closer to Original than in Fig. 8 since the burstiness is significantly lower, but Tspan is still not as good as *TraceUpscaler*, because Tspan distorts the short term temporal pattern in the Original trace, while *TraceUpscaler* maintains that.

The major pitfall of the Repeat policy is in failing to realistically represent the cache usage patterns. In systems with a cache (Fig. 8a), the repeated requests get cached and significantly distort latency characteristics. In stateless systems (Fig. 8b), Repeat performs closely to Original indicating that repeating the timestamps is a good way to maintain temporal patterns.

Our approach, *TraceUpscaler*, accurately upscales timestamps by repeating timestamps similarly to Repeat, but it does not reuse requests and follows the same request ordering as in the Input trace to maintain caching behaviors, similar to Tspan. This results in producing the best quality upscaled traces in both stateful and stateless systems.

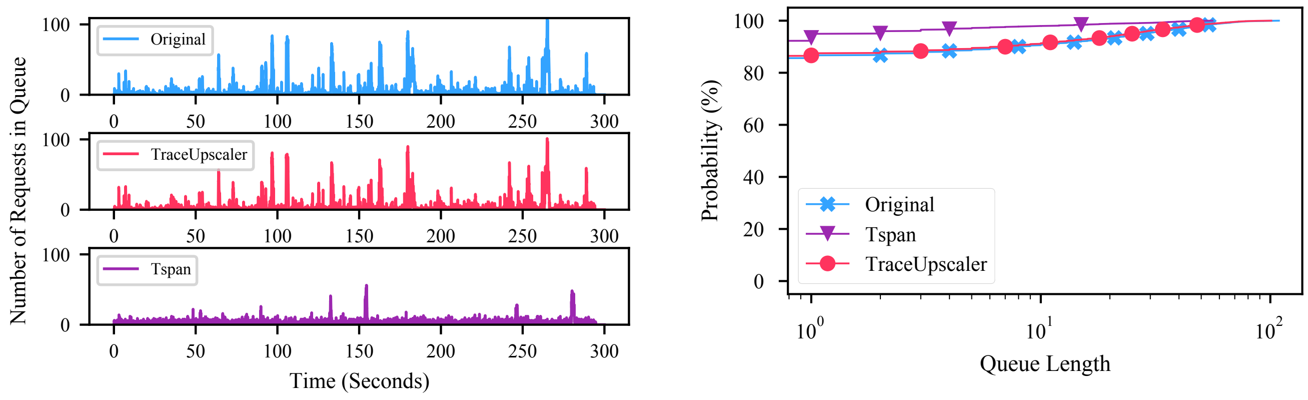
5.2.2 Impact of Caching. To show the impact of caching in isolation, we create a synthetic trace with a Poisson arrival process, which is far less bursty than the traces described in Sec. 4.4. Since we want to demonstrate the impact of caching, we use the trace in our DeathStarBench setup (Sec. 4.3), and collect the cache hit statistics from frontend Varnish cache to see how those are preserved in different upscaling techniques. As shown in Fig. 10b, Repeat gets a consistent cache hit rate of around 50% due to exactly repeating each request twice (due to $2\times$ upscaling). Similarly, since AverageRateScaling randomly samples requests from each time interval, we can see the requests getting repeated

[3]Even though less bursty, it still has some small bursts.



(a) Synthetic bursty trace: DeathStarBench (stateful) **(b)** Synthetic bursty trace: MediaWiki (stateless)

Figure 8. Comparison of different upscaling techniques when handling bursty synthetic traces (details in Sec. 5.2.1).



(a) Queue Length over Time **(b)** CDF of Queue Length

Figure 9. Queue length from the Fig. 6c experiment. We can see from Fig. 9a that Tspan fails to recreate the longer queues that develop throughout the Original trace, whereas *TraceUpscaler* succeeds in doing so. The shorter burst periods in Tspan results in shorter queues in the system. This lowers the queueing times, thus distorting the latency characteristics of the upscaled trace. Fig. 9b shows that across the distribution of queue lengths, Tspan is missing the high queue lengths seen in the Original trace and replicated in *TraceUpscaler*, which explains why the latency from Tspan is consistently lower than the Original trace.

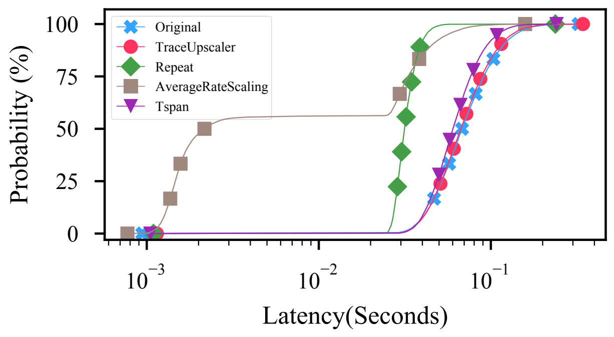
and growing in cache hit rate until the next time interval begins. These phenomena explain the much lower latency from Repeat and AverageRateScaling compared to the Original trace, as shown in Fig. 10a. On the other hand, Tspan and *TraceUpscaler* both capture the cache access pattern of the original trace properly.

5.3 Non-integral Scaling

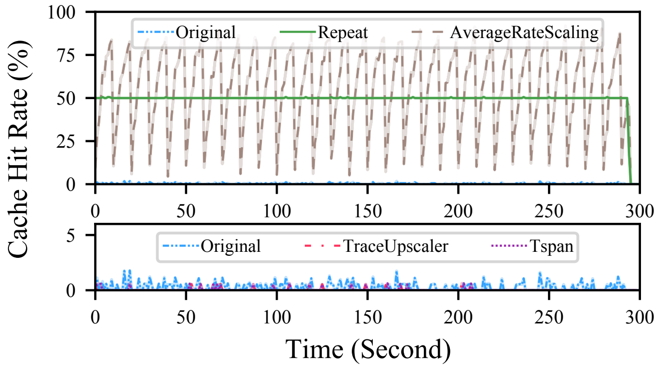
We now consider cases where the upscaling factor is not an integer, i.e., fractional numbers. We evaluate two upscaling factors (f = 1.25 and f = 1.5) in the DeathStarBench setup. Fig. 11 shows results from a trace with arrival timestamps from real-world traces (details in Sec. 4.4). Fig. 12 shows results from a bursty trace with synthetic arrival timestamps. From both sets of results along with the earlier f = 2 results, we can see that *TraceUpscaler* does better than the baselines in preserving latency properties. Of note, the Repeat approach varies the most between scaling factors, which is due to the fraction of repeated requests. With f = 1.25, only a quarter of requests are repeated, so there is a smaller impact from caching than the f = 1.5 and f = 2 cases.

5.4 Extreme Scaling

We next investigate the efficacy of our upscaling approach in scenarios where the upscaling factor is much higher than 2. We evaluate two upscaling factors (f = 3 and f = 5) in the DeathStarBench setup. Fig. 13 shows results from a trace with arrival timestamps from real-world traces (details in Sec. 4.4). Fig. 14 shows results from a bursty trace with synthetic arrival timestamps. We can see that even when upscaling by a very large factor, *TraceUpscaler* does a reasonable job of matching latency characteristics of the Original trace, outperforming all the baselines. However, upscaling itself is a process that generates hypothetical traces with higher loads, so it is not expected to be 100% realistic. Our results show that *TraceUpscaler* does a reasonable job at preserving trace characteristics even with extreme upscaling factors of 5, but such extreme

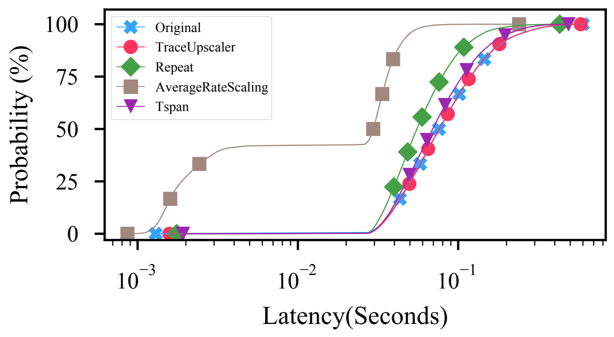


(a) Latency

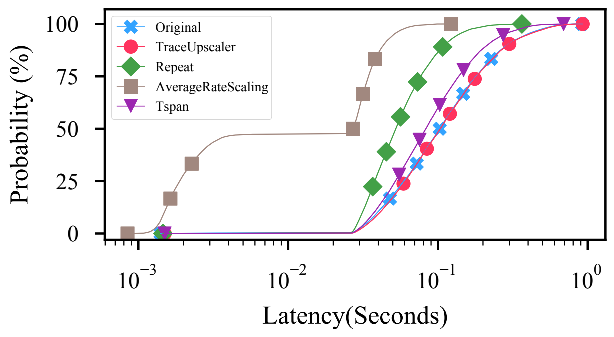


(b) Cache hit rate (%) over time.

Figure 10. Comparison of different upscaling techniques in the stateful DeathStarBench setup using a trace without significant burstiness (Poisson process at a fixed rate). This result demonstrates the impact on cache access characteristics due to different upscaling techniques (details in Sec. 5.2.2).

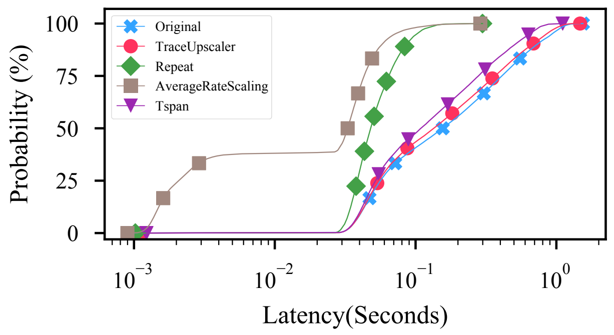


(a) upscaling factor, f = 1.25

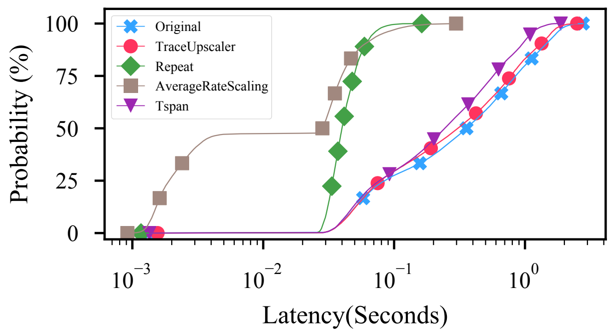


(b) upscaling factor, f = 1.5

Figure 11. Comparison of different upscaling techniques when upscaling by a non-integral upscaling factor in the stateful DeathStarBench setup using a trace with arrival timestamps from the Microsoft OneRF trace (details in Sec. 5.3).



(a) upscaling factor, f = 1.25



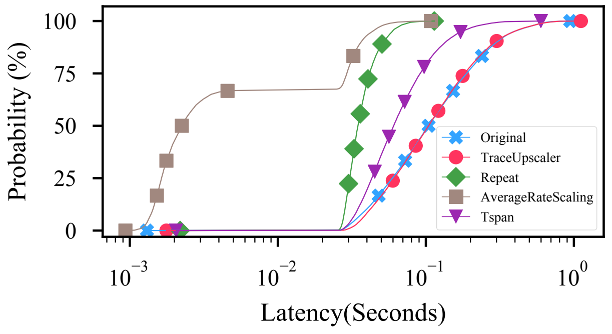
(b) upscaling factor, f = 1.5

Figure 12. Comparison of different upscaling techniques when upscaling by a non-integral upscaling factor in the stateful DeathStarBench setup using a trace with synthetic arrival timestamps (details in Sec. 5.3).

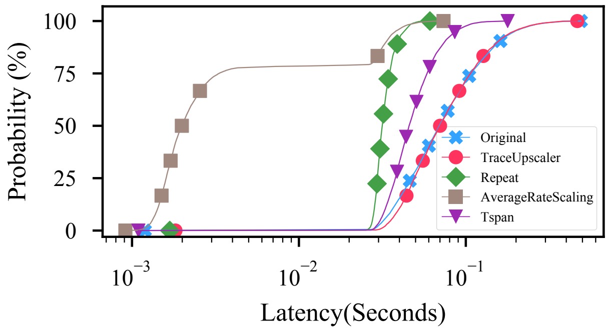
upscaling should be treated with caution. We have not evaluated how *TraceUpscaler* performs with higher upscaling factors, and do not make any claims about the performance of *TraceUpscaler* under those scenarios.

5.5 Representing Overloads

The previous results demonstrate the efficacy of *TraceUpscaler* in upscaling traces while preserving latency characteristics, but how does inaccurately representing latency impact experiments and system decisions? We conduct a case study

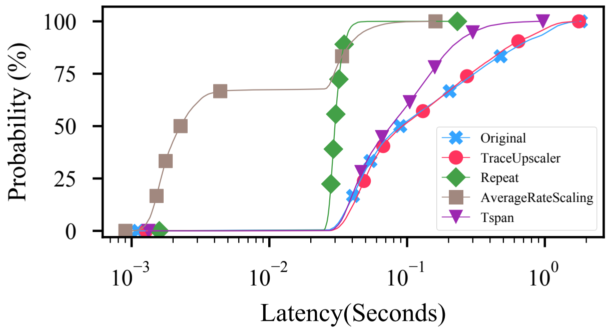


(a) upscaling factor, f = 3

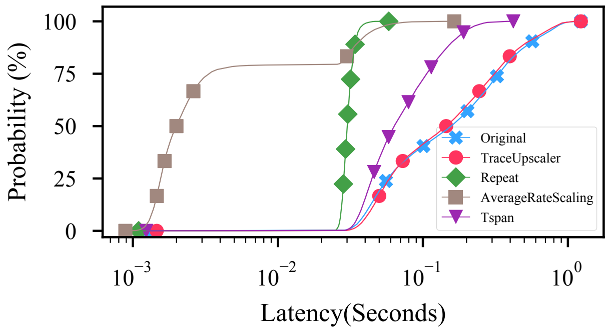


(b) upscaling factor, f = 5

Figure 13. Comparison of different upscaling techniques when upscaling by an extreme upscaling factor (>2) in the stateful DeathStarBench setup using a trace with arrival timestamps from the Microsoft OneRF trace (details in Sec. 5.4).



(a) upscaling factor, f = 3



(b) upscaling factor, f = 5

Figure 14. Comparison of different upscaling techniques when upscaling by an extreme upscaling factor (>2) in the stateful DeathStarBench setup using a trace with synthetic arrival timestamps (details in Sec. 5.4).

where we select an Original trace that temporarily overloads the system. Ideally, the upscaling should produce an upscaled trace that illustrates the same temporary overload so that the experimenter would conclude that the system cannot handle the high load in the Original and upscaled traces.

Fig. 15 shows the latency over time where the load is temporarily too high for our experimental DeathStarBench setup (i.e., overloads our experimental cluster). We create the trace for this experiment using arrival timestamps from a Microsoft OneRF trace exhibiting temporary elevated load. We see that only *TraceUpscaler* can recreate the overload in the system just like in the Original trace. Repeat and AverageRateScaling fail to represent the overload due to increased cache access and suppressing short-term bursts, respectively. Tspan somewhat preserves the overload, although it has been squeezed to a shorter duration (and occurs earlier) in the upscaled trace. Fold exhibits an overload at the same time as the Original trace, but the magnitude of the overload is reduced since the folding process overlaps the temporary overload time period with a non-overloaded period.

Due to these inaccuracies, the upscaling technique could lead practitioners into a false sense of security. With AverageRateScaling and Repeat, one might think the system is fully capable of handling the high load without any issue, when in fact it can be severely overloaded. Furthermore, practitioners may invest in expensive caching solutions to handle the upscaled traces from Repeat and AverageRateScaling when in practice there may not be as much repetition in the workload. With Tspan and Fold, one might detect the presence of an overload, but might be misled into the magnitude and duration of overloads. While this can lead the practitioner toward the actual bottleneck in the system, they might still be lacking information about correct amount of resources needed to resolve the bottleneck. The goal of our work is to raise awareness for potential pitfalls that may occur when upscaling traces and introduce a new approach, *TraceUpscaler*, that does a much better job at preserving the trace characteristics when upscaling.

6 Discussion

Evaluation on different systems: For our evaluation, we intentionally upscaled traces on the same system as the Original

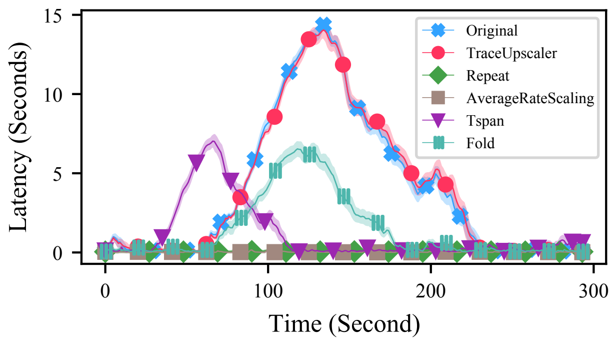


Figure 15. Comparison of different upscaling techniques in recreating temporary overload characteristics in the stateful DeathStarBench setup using a trace with arrival timestamps from the Microsoft OneRF trace (details in Sec. 5.5).

trace so that we have a baseline to compare our results against. Our evaluation methodology is designed to eliminate confounding factors and focus solely on the upscaling changes to the trace itself. In practice, traces are sometimes run on different systems and applications than the Original system, and handling these differences is a complementary problem. For example, DITTO [64] addresses the problem of creating a synthetic application to mimic an Original application. By contrast, our focus is on how we can manipulate the traces to increase the load in a realistic manner. This can be used to increase the load to account for system differences such as more hardware resources, newer hardware generations, different system designs/architectures, etc. The performance of these different systems will naturally be different from the Original system, and the goal of our work is to justify that these performance differences are not due to peculiarities in the upscaling process.

Preserving latency characteristics: Trace upscaling is a subjective process whereby one is synthesizing a *what-if* scenario to evaluate performance under a higher rate of incoming traffic. In this work, we focus on preserving latency characteristics (including at high tail percentiles) for open-loop systems because latency is the primary performance metric in these use cases and also serves as a good proxy metric for representing the behaviors of a system. There have been works that scale traces in closed-loop systems [74], but directly transferring solutions developed for closed-loop systems for trace upscaling in open-loop systems can lead to failure in capturing tail latency characteristics [24, 58]. Arrival timestamps are only relevant for open-loop systems, and our research shows that the way that timestamps are upscaled can significantly impact performance. We also consider cache hit rate, and in practice, there may be other metrics of interest. Maintaining multiple trace characteristics during the upscaling process is ideal, but it may not always be possible and is beyond the scope of our work.

High upscaling factors: Our results evaluate *TraceUpscaler* across a range of upscaling factors including non-integral and aggressive (up to 5×) factors, but upscaling is fundamentally limited by the input data, and thus it cannot be expected to always work for high upscaling factors. As an analogy, the quality of upscaling an image or video is limited by the input data. We recommend experimenters to use caution when using high upscaling factors and be cognizant of the contexts of their experiments/studies in relation to the upscaling approach to avoid the pitfalls described in this work.

Sensitivity of TraceUpscaler to timestamp alignment: In addition to repeating timestamps, we have also experimented with tweaking *TraceUpscaler* to use different types of timestamp offsets (e.g., fixed offsets, random offsets, randomly between successive requests, uniformly spaced between successive requests), but the results did not show any noticeable difference. Our results indicate that *TraceUpscaler* works well when the relative order of requests is preserved. Hence, we opt to repeat requests at the exact same time, as it is the simplest approach, preserves the original timestamps, and does not introduce bias from the practitioner in terms of choosing timestamp offsets. Similar effects were observed from the performance of Repeat.

Deployment of TraceUpscaler: We have designed *TraceUpscaler* to work as an offline stand-alone tool that upscales the collected traces to generate upscaled traces to be used for trace replay. Practitioners can use *TraceUpscaler* to generate appropriately upscaled traces spanning any number of nodes. *TraceUpscaler* does not need to be deployed in an existing cluster/system since it only operates on the trace data. However, users can easily integrate it as part of their experimental load generator/replayer to generate upscaled load representative of an already existing trace.

7 Conclusion

This paper conducts the first study on upscaling traces to increase load in latency-sensitive open-loop applications. We motivate the need for accurate upscaling techniques and raise awareness for how current practices are inadequate in representing latency characteristics in upscaled traces. We address these pitfalls by introducing a novel upscaling technique, *TraceUpscaler*, that realistically upscales traces while preserving temporal patterns, caching-related effects, and latency characteristics. Through extensive evaluation, we demonstrate how *TraceUpscaler* outperforms existing approaches in upscaling realistically. *TraceUpscaler* is available as an open source tool to help the community conduct more realistic experiments when upscaling traces.

Acknowledgments

We thank our shepherd John Wilkes and the anonymous reviewers who provided constructive and helpful feedback. We also thank Avimita Chatterjee for her help with the DeathStar setup. This research was supported in part by National Science Foundation grant 2122155.

References

[1] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. 2020. Reanimator: Versatile high-fidelity storage-system tracing and replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference*. 61–74.

[2] Abubakr O Al-Abbasi, Vaneet Aggarwal, and Tian Lan. 2019. TTLoC: Taming tail latency for erasure-coded cloud storage systems. *IEEE Transactions on Network and Service Management* 16, 4 (2019), 1609–1623.

[3] 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.

[4] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. 2004. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*.

[5] Martin F Arlitt and Carey L Williamson. 1996. *Web server workload characterization: The search for invariants (extended version)*. Technical Report. Citeseer.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[7] David H Bailey. 1991. Twelve ways to fool the masses when giving performance results on parallel computers. In *Supercomputing Review*. 54–55.

[8] Daniel J Barrett. 2008. *MediaWiki: Wikipedia and beyond*. O'Reilly Media, Inc.

[9] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching–Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 195–212.

[10] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 376–391.

[11] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*. 257–272.

[12] Jaki Bhimani, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, and Vijay Balakrishnan. 2020. Auto-tuning parameters for emerging multi-stream flash-based storage drives through new I/O pattern generations. *IEEE Trans. Comput.* 71, 2 (2020), 309–322.

[13] 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 (HotCloud'09)*. USENIX Association.

[14] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael I Jordan, and David A Patterson. 2009. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. 1–6.

[15] Kirill L Bogdanov, Waleed Reda, Gerald Q Maguire Jr, Dejan Kostić, and Marco Canini. 2018. Fast and accurate load balancing for geo-distributed storage systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 386–400.

[16] 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.

[17] 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.

[18] Defu Cao, Yujing Wang, Juanyong Duan, Ce Zhang, Xia Zhu, Conguri Huang, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, et al. 2021. Spectral temporal graph neural network for multivariate time-series forecasting.

[19] Marcus Carvalho, Francisco Brasileiro, Raquel Lopes, Giovanni Farias, Alessandro Fook, João Mafra, and Daniel 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)*. IEEE, 160–169.

[20] Marcus Carvalho, Daniel Menasce, and Francisco 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)*. IEEE, 82–90.

[21] Marcus Carvalho, Daniel A Menascé, and Francisco Brasileiro. 2017. Capacity planning for IaaS cloud providers offering multiple service classes. *Future Generation Computer Systems* 77, 97–111.

[22] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM Sigmod Record* 39, 3, 5–10.

[23] Zheyi Chen, Jia Hu, Geyong Min, Albert Y 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* 31, 4, 923–934.

[24] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. *ACM SIGARCH Computer Architecture News* 41, 3, 308–319.

[25] 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.

[26] Yong Cui, Ningwei Dai, Zeqi Lai, Minming Li, Zhenhua Li, Yuming Hu, Kui Ren, and Yuchi Chen. 2019. Tailcutter: Wisely cutting tail latency in cloud cdns under cost constraints. *IEEE/ACM Transactions on Networking* 27, 4, 1612–1628.

[27] Peter Danzig, Jeff Mogul, Vern Paxson, and Mike Schwartz. 2000. The internet traffic archive. *URL: http://ita. ee. lbl. gov*.

[28] Anuroop Desu, Udaya Puvvadi, Tyler Stachecki, Sagar Vishwakarma, Sadegh Khalili, Kanad Ghose, and Bahgat G Sammakia. 2021. Latency-Aware Dynamic Server and Cooling Capacity Provisioner for Data Centers. In *Proceedings of the ACM Symposium on Cloud Computing*. 335–349.

[29] Diego Didona and Willy Zwaenepoel. 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 79–94.

[30] 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.

[31] Dror G Feitelson. 1996. Packing schemes for gang scheduling. In *workshop on job scheduling strategies for parallel processing*. Springer, 89–110.

[32] Wolfgang Fischer and Kathleen Meier-Hellstern. 1993. The Markov-modulated Poisson process (MMPP) cookbook. *Performance evaluation* 18, 2, 149–171.

[33] Eitan Frachtenberg, Dror G Feitelson, Juan Fernandez, and Fabrizio Petrini. 2003. Parallel job scheduling under dynamic workloads. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA. Revised Paper 9*. Springer, 208–227.

[34] 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.

[35] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[36] Gregory R. Ganger and Yale N. Patt. 1998. Using system-level models to evaluate I/O subsystem designs. *IEEE Trans. Comput.* 47, 6, 667–678.

[37] Jan Gasthaus, Konstantinos Benidis, Yuyang Wang, Syama Sundar Rangapuram, David Salinas, Valentin Flunkert, and Tim Januschowski. 2019. Probabilistic forecasting with spline quantile function RNNs. In *The 22nd international conference on artificial intelligence and statistics*. PMLR, 1901–1910.

[38] 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

[39] 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.

[40] Raúl Gracia-Tinedo, Danny Harnik, Dalit Naor, Dmitry Sotnikov, Sivan Toledo, and Aviad Zuck. 2015. SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 317–330.

[41] Pablo Graziano. 2013. Speed up your web site with Varnish. *Linux Journal* 2013, 227, 4.

[42] 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.

[43] 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.

[44] Alireza Haghdoost, Weiping He, Jerry Fredin, and David HC Du. 2017. On the Accuracy and Scalability of Intensive I/O Workload Replay. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 315–328.

[45] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

[46] Joseph L. Hellerstein. 2010. Google Cluster Data. http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[47] Roger W Hockney. 1996. *The science of computer benchmarking*. SIAM.

[48] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[49] 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.

[50] Raj Jain. 1990. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

[51] Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riordan. 1997. Modeling of workload in MPPs. In *Job Scheduling Strategies for Parallel Processing: IPPS'97 Processing Workshop Geneva, Switzerland, April 5, 1997 Proceedings 3*. Springer, 95–116.

[52] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. 2014. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*. 61–74.

[53] 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, 22495–22508.

[54] Nikolai Joukov, Timothy Wong, and Erez Zadok. 2005. Accurate and Efficient Replaying of File System Traces. In *FAST*, Vol. 5. 25–25.

[55] Da-Cheng Juan, Lei Li, Huan-Kai Peng, Diana Marculescu, and Christos Faloutsos. 2014. Beyond poisson: Modeling inter-arrival time of requests in a datacenter. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 198–209.

[56] Gueyoung Jung, Matti A Hiltunen, Kaustubh R Joshi, Richard D Schlichting, and Calton Pu. 2010. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE, 62–73.

[57] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. SelfTune: Tuning Cluster Managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1097–1114.

[58] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. *ACM SIGPLAN Notices* 49, 4, 729–742.

[59] 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* 9, 4, 1305–1322.

[60] Furkan Koltuk and Ece Güran Schmidt. 2020. A Novel Method for the Synthetic Generation of Non-IID Workloads for Cloud Data Centers. In *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–6.

[61] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2020. *Systems Benchmarking*. Springer.

[62] Miryeong Kwon, Jie Zhang, Gyuyoung Park, Wonil Choi, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. 2017. TraceTracker: Hardware/software co-evaluation for large-scale I/O workload reconstruction. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–96.

[63] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 981–992.

[64] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. 2023. Ditto: End-to-End Application Cloning for Networked Cloud Services. In *Proceedings of*

the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 222–236.

[65] Dennis KJ Lin, Timothy W Simpson, and Wei Chen. 2001. Sampling strategies for computer experiments: design and analysis. *International Journal of Reliability and applications* 2, 3, 209–240.

[66] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. 2019. Generating high-fidelity, synthetic time series datasets with doppelganger. *arXiv preprint arXiv:1909.13403*.

[67] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. 2020. Using GANs for sharing networked time series data: Challenges, initial promise, and open questions. In *Proceedings of the ACM Internet Measurement Conference*. 464–483.

[68] 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.

[69] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2884–2892.

[70] Uri Lublin and Dror G Feitelson. 2003. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel and Distrib. Comput.* 63, 11, 1105–1122.

[71] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[72] 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.

[73] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 409–425.

[74] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio César López-Hernández, James Hendricks, Gregory R. Ganger, and David R. O'Hallaron. 2007. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*. USENIX Association, San Jose, CA.

[75] Haibo Mi, Huaimin Wang, Gang Yin, Yangfan Zhou, Dianxi Shi, and Lin Yuan. 2010. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *2010 IEEE International Conference on Services Computing*. IEEE, 514–521.

[76] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. 2010. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review* 37, 4, 34–41.

[77] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. 2021. Mu: an efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*. 168–181.

[78] 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.

[79] Usama Naseer and Theophilus A Benson. 2022. Configanator: A Data-driven Approach to Improving CDN Performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1135–1158.

[80] Lucas Nussbaum. 2017. Testbeds support for reproducible research. In *Proceedings of the reproducibility workshop*. 24–26.

[81] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. 1999. Flash: an efficient and portable web server. In *USENIX Annual Technical Conference, General Track*. 199–212.

[82] Fanny Pascual and Krzysztof Rzadca. 2018. Colocating tasks in data centers using a side-effects performance model. *European Journal of Operational Research* 268, 2, 450–462.

[83] Daniel Perez, Kin K Leung, et al. 2020. Fast-Fourier-Forecasting Resource Utilisation in Distributed Systems. *arXiv preprint arXiv:2001.04281*.

[84] 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, 2046–2059.

[85] Kashif Rasul, Abdul-Saboor Sheikh, Ingmar Schuster, Urs Bergmann, and Roland Vollgraf. 2020. Multivariate probabilistic time series forecasting via conditioned normalizing flows. *arXiv preprint arXiv:2002.06103*.

[86] 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.

[87] 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, 447–473.

[88] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[89] Anirudh Sabnis and Ramesh K Sitaraman. 2022. JEDI: model-driven trace generation for cache simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 679–693.

[90] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. 1989. Design and analysis of computer experiments. *Statistical science* 4, 4, 409–423.

[91] Sultan Mahmud Sajal, Rubaba Hasan, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. 2021. TraceSplitter: a new paradigm for downscaling traces. In *EuroSys*. 606–619.

[92] Thomas J Santner, Brian J Williams, William I Notz, and Brain J Williams. 2003. *The design and analysis of computer experiments*. Vol. 1. Springer.

[93] Stefano Sebastio, Michele Amoretti, Alberto Lluch Lafuente, and Antonio Scala. 2018. A holistic approach for collaborative workload execution in volunteer clouds. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28, 2, 1–27.

[94] 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*. 105–114.

[95] Stefano Sebastio and Giorgio Gnecco. 2018. A green policy to schedule tasks in a distributed cloud. *Optimization Letters* 12, 7, 1535–1551.

[96] Rajat Sen, Hsiang-Fu Yu, and Inderjit Dhillon. 2019. Think globally, act locally: A deep neural network approach to high-dimensional time series forecasting. *arXiv preprint arXiv:1905.03806*.

[97] Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon. 2013. IO workload characterization revisited: A data-mining approach. *IEEE Trans. Comput.* 63, 12, 3026–3038.

[98] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. Snf: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 296–310.

[99] Alina Sîrbu and Ozalp Babaoglu. 2015. Towards data-driven autonomics in data centers. In *2015 International Conference on Cloud*

and Autonomic Computing. IEEE, 45–56.

[100] SNIA. 2023. SNIA IOTTA trace repository. http://iotta.snia.org/. Accessed: Jan 10, 2023.

[101] 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.

[102] 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.

[103] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 570–584.

[104] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 513–527.

[105] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. 2012. Extracting flexible, replayable models from large block traces. In *FAST*, Vol. 12. 22.

[106] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *EuroSys'20*. Heraklion, Crete.

[107] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *FAST*, Vol. 11. 163–176.

[108] 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.

[109] 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

[110] Abhishek Verma, Madhukar Korupolu, and John Wilkes. 2014. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 48–56.

[111] 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.

[112] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 443–457.

[113] 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*. 246–258.

[114] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter Reiher, and Geoff Kuenning. 2007. PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage (TOS)* 3, 3, 13–es.

[115] Zev Weiss, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2013. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 373–387.

[116] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 36, SI, 255–270.

[117] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating Serverless Computing by Harvesting Idle Resources. In *Proceedings of the ACM Web Conference 2022*. 1741–1751.

[118] 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.

[119] Qi Zhang, Joseph Hellerstein, and Raouf Boutaba. 2011. Characterizing Task Usage Shapes in Google Compute Clusters. In *Proceedings of the 5th International Workshop on Large Scale Distributed Systems and Middleware*.

[120] Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L Hellerstein. 2013. Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 510–519.

[121] Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L Hellerstein. 2014. Dynamic heterogeneity-aware resource provisioning in the cloud. *IEEE transactions on cloud computing* 2, 1 (2014), 14–28.

[122] 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.

[123] Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. 2013. Cosbench: Cloud object storage benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 199–210.

[124] Ningning Zhu, Jiawu Chen, Tzi-Cker Chiueh, and Daniel Ellard. 2005. TBBT: Scalable and accurate trace replay for file server evaluation. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (2005), 392–393.

A Artifact Appendix

A.1 Abstract

Our artifact is contained within two repositories. Our public GitHub repository contains the *TraceUpscaler* code for upscaling traces. Our public Zenodo repository contains the scripts, configurations, and instructions for reproducing the results from the paper. Since we cannot release the private Microsoft OneRF trace data, we only include the synthetic trace data used for Fig. 8a and Fig. 10a. We also provide **a video** that shows how to use the *TraceUpscaler* tool to upscale traces in practice.

A.2 Description & Requirements

A.2.1 How to access.

TraceUpscaler: To access *TraceUpscaler*, a user can use either of the following options:

1. **GitHub link**:
   https://github.com/smsajal/TraceUpscaler
2. **Zenodo Link**:
   https://zenodo.org/doi/10.5281/zenodo.10042017
   (DOI: 10.5281/zenodo.10042017)

A.2.2 Hardware dependencies.

TraceUpscaler does not have any specific hardware dependencies. Reproducing our results requires using the VM types as described in Sec. 4.3.

A.2.3 Software dependencies.

TraceUpscaler: The *TraceUpscaler* software was developed using —

1. Java 17 (Amazon Corretto)
2. Apache Commons Lang 3.12.0
3. Apache Commons Math 3.6.1
4. Google Gson 2.7

Reproducing results: All of these were deployed in Azure VMs running Ubuntu 18.04.

DeathStarBench:

1. Docker 24.0.2
2. libssl-dev 1.1.1
3. libz-dev 1.2.11
4. luarocks 2.4.2
5. luasocket 3.1.0-1
6. Python 3.6.9
   a. multidict 5.2.0
   b. yarl 1.7.2
   c. typing_extensions 4.1.1
   d. async_timeout 4.0.2
   e. idna_ssl 1.1.0
   f. charset_normalizer 3.0.1
   g. aiosignal 1.2.0
   h. aiohttp 3.8.5
7. Nginx 1.25.2
8. Varnish Http Cache 6.6.1

MediaWiki:

1. MediaWiki 1.35.2
2. PHP 7.4
3. libapache2-mod-php7.4
4. php7.4-mcrypt
5. php7.4-mbstring
6. php7.4-xml
7. php7.4-mysql
8. MySQL 8.0.35
9. Apache2 Server 2.4.41
10. Nginx 1.18.0

A.2.4 Benchmarks.

We evaluate our work using the DeathStarBench and MediaWiki benchmarks as described in Sec. 4.3.

A.3 Set-up

A.3.1 *TraceUpscaler*:

Install the software dependencies and run *TraceUpscaler* according to the `README` in either the GitHub or Zenodo repository. The `README` contains details about the parameters for running *TraceUpscaler*, the trace format, etc.

A.3.2 Reproducing Results:

For setting up the DeathStarBench and MediaWiki setups, please follow the instructions in the `README` files in their respective directories in the Zenodo repository. The `README` contains details about setting up the systems across multiple VMs, important configurations, scripts and instructions for running the experiments, etc.

A.4 Evaluation workflow.

A.4.1 Major Claims.

- *(C1): TraceUpscaler outperforms all the other baselines in preserving the latency characteristics of the Original trace when upscaling. This is demonstrated by multiple experiments in Sec. 5, and we provide the Bursty synthetic trace data for reproducing the experiment in the DeathStarBench setup (results shown in Fig. 8a).*
- *(C2): Tspan does badly in handling bursts in traces, and in the absence of the bursts, it does reasonably well in upscaling. AverageRateScaling does badly due to distorting the cache access pattern present in the original trace and potential failure to capture bursts. For a stateful system such as DeathStarBench, the distorted cache access contributes more to the failure of replicating latency characteristics. Hence, even in the absence of bursts, AverageRateScaling does poorly. This is shown by the experiment in Sec. 5.2.2, and we provide the synthetic trace data for reproducing the result shown in Fig. 10a.*

A.4.2 Experiments.

Experiment (E1). : [*TraceUpscaler* Outperforms Other] [10 human-minutes + 3 compute-hour]: Runs an Original bursty synthetic trace and upscaled traces from a subtrace of the Original trace using different upscaling techniques. The expected result shows that *TraceUpscaler* most closely

matches the latency distribution of the Original trace, while the other upscaling technique traces do not. The result plot should match the trends in Fig. 8a.
[How to]

[Preparation] Please make sure you have setup the environment as described in Sec. A.3.2.

[Execution] Please run the *run_8a.sh* shell script in the *deathstar/Archive/experiments/scripts* directory. This would start an experiment that is expected to run for around 3 hours.

[Results] To create the plot, run the python file located in *deathstar/Archive/experiments/scripts/src/plotting/plot_gen.py* with the first parameter being the address of the results directory.

Experiment (E2). : [Explanation of Shortcomings of Tspan and AverageRateScaling] [10 human-minutes + 3 compute-hour]: Runs an Original non-bursty synthetic trace and upscaled traces from a subtrace of the Original trace using different upscaling techniques. The expected result shows that *TraceUpscaler* most closely matches the latency distribution of the Original trace, Tspan does reasonably well, and the other upscaling techniques do not work well. The result plot should match the trends in Fig. 10a.
[How to]

[Preparation] Please make sure you have setup the environment as described in Sec. A.3.2.

[Execution] Please run the *run_10a.sh* shell script in the *deathstar/Archive/experiments/scripts* directory. This would start an experiment that is expected to run for around 3 hours.

[Results] To create the plot, run the python file located in *deathstar/Archive/experiments/scripts/src/plotting/plot_gen.py* with the first parameter being the address of the results directory.