Trace-Based Workload Generation and Execution

Yannis Sfakianakis[1,2(✉)], Eleni Kanellou[1], Manolis Marazakis[1], and Angelos Bilas[1,2]

[1] Institute of Computer Science, Foundation for Research and Technology – Hellas (FORTH), Heraklion, Greece
{jsfakian,kanellou,maraz,bilas}@ics.forth.gr

[2] Department of Computer Science, University of Crete, Heraklion, Greece

Abstract. Although major cloud providers have captured and published workload executions in the form of traces, it is not clear how to use them for workload generation on a wide range of existing platforms. A methodological challenge that remains is to generate and execute realistic datacenter workloads on any infrastructure, using information from available traces. In this paper, we propose *Tracie*, a methodology addressing this challenge, and introduce the tool supporting its implementation. We present all the necessary steps starting from a trace up to workload execution: analysis of datacenter traces, extraction of parameters, application selection, and scaling of a workload to match the capabilities of the underlying infrastructure. Our evaluation validates that *Tracie* can generate executable workloads that closely resemble their trace-based counterparts. For validation, we correlate the recorded system metrics of a trace against the actual execution. We find that the average system metrics of synthetic workloads differ at most 5% compared to the trace and that they are highly correlated at 70% on average.

Keywords: Trace · Workload · Simulation · Cloud computing · Benchmarking

1 Introduction

The generation and execution of realistic cloud workloads constitute a significant methodological challenge in cloud computing. There have been two main directions towards addressing this challenge: (1) benchmark suites with popular cloud applications [10,19] and (2) simulators that rely on workload descriptions extracted from publicly available datacenter traces [2]. The first approach has the advantage that one can execute the workload on an existing platform, making it amenable to parameter tuning for specific purposes. However, the correlation of the resulting workload to the available datacenter traces is unclear and generally remains the responsibility of the user to validate and demonstrate it. The second approach directly connects to datacenter workloads; however, given only a trace, it is not possible to execute on an existing platform, limiting its applicability.

© Springer Nature Switzerland AG 2021

This paper presents a robust and practical methodology, *Tracie*, to generate executable workloads from datacenter traces and sets of sample applications. The resulting workloads match key statistical characteristics derived from the respective traces. Towards this goal, we first determine which trace parameters characterize the trace and we include them in the generated actual executions. Parameter selection is not straightforward because each trace contains hundreds of parameters, such as job start time, number of tasks per job, etc. Second, we develop models of these parameters for the generation of the datacenter workload. Finally, we define a trace-specific application pool for the execution of the generated workloads. Workload execution is a real challenge because traces do not contain all the necessary variables to execute a workload. For example, we do not know the application types that were executed during trace creation.

To determine which variables of the trace are significant, we organize them into three categories: (1) variables that are *inherent* to the workload, for example, the arrival time of an application; i.e. workload variables, (2) variables that are *induced* to the workload, for example, the CPI of an application; i.e. execution variables, and (3) variables that refer to the infrastructure, for example, number of cores of server in the infrastructure; i.e. infrastructure variables. Then, we select the workload variables for the workload generation and the execution variables for the workload execution. Variables concerning the infrastructure are not crucial for describing the workload; hence, we omit them from further consideration. We model the workload as a random set of instances of the selected variables. Hence, we consider that each variable follows a different probability distribution function (PDF) and we estimate each PDF by processing the trace.

For workload execution, it is not feasible to precisely select the types of applications and the corresponding data, since this information is not part of the trace. Realistically, the best approximation we can achieve is to choose applications with similar micro-architectural characteristics, e.g. cycles per instruction (CPI). Towards this objective, we first estimate the PDF of each micro-architectural parameter available in the trace. We then measure the micro-architectural characteristics of several applications, executing them individually. Finally, we select from the available pool of application kernels the subset that best matches the micro-architectural characteristics extracted from the trace.

Overall, *Tracie* provides the methodology for the generation, execution, and validation of datacenter workloads resembling the source traces. Unlike prior related work, *Tracie* manages to generate realistic workloads on a wide range of existing platforms, starting from an analysis of a datacenter trace and leading up to the actual execution and monitoring of units of work selected from a pool of available application kernels. For validation, we define a method to statistically compare how close the execution of a workload generated by *Tracie* matches the original one captured in the trace. This is essential because the workload execution is affected by execution variables not included in the trace, e.g. interference of co-located applications and the dependencies of tasks belonging to an application.

In our evaluation, we use three types of containerized applications: (1) services that execute for a long time, e.g. a web server, (2) client applications that emulate the incoming requests of a service, e.g. the `ab` benchmark [1] issuing requests to a web server, and (3) batch applications that perform a specific amount of work, e.g. running a machine learning algorithm. Our evaluation shows that our PDF estimations of workload and execution characteristics match closely the histograms of these characteristics as captured in the traces. Correlation is 75% on average for workload parameters and 70% on average for micro-architectural parameters of applications.

The main contributions of this paper are as follows:

- Extraction of common parameters for datacenter traces that unify the description of traces from different cloud providers.
- A method for selecting a representative subset of parameters from a pool of available application kernels to match micro-architectural characteristics of applications used by traces.
- A method to validate that workload generated by *Tracie* match the execution of the captured workload trace, particularly in terms of key micro-architectural characteristics.
- A practical and easy to use workload generator that can generate executable workloads starting from a set of applications, or kernels.

2 Trace Analysis

We summarize the notions used in a trace as follows:

- A *task* is an indivisible unit of work that executes on a single processing unit. Task duration may vary significantly across tasks, from milliseconds to hours.
- A *job* is a set of tasks. For instance, in a web server, each user request is a job and consists of many tasks. Different Spark jobs in a Spark application appear in the trace as individual jobs.
- An *application* is a set of jobs that execute in batch mode, i.e. we are interested in the completion time of the full application and not individual jobs or tasks.
- A *service* is a set of user-facing jobs, i.e. we are interested in the completion time of individual jobs, as well as the job rate. Typically, services are assumed to run continuously.
- A *workload* is a set of applications and services.

Starting with the traces Google 2011 [29], Alibaba [24], and Google 2019 [30], we summarize the main events captured as follows:

- *Job events* represent changes in the state of a job, e.g. when a job is submitted or begins execution.
- *Task events* represent changes in the state of a task, similar to jobs. Task events may also contain constraints, e.g. when a task should (not) run on a specific server or task affinity with data.

- *Machine events* represent changes in the hardware or the software of the infrastructure, e.g. when a server is added, a kernel is updated, or a server fails. Machine, events, may also contain machine attributes, e.g. the amount of DRAM available in a server.
- *Resource events* represent the resources reserved or used by jobs and tasks within the interval, e.g. average CPU utilization over 10s. We exclude from this category events that refer to cumulative machine use, and instead, we include these in machine events.

Job, task, machine, and resource reservation events are point events, whereas resource usage events are periodic and refer to intervals. The above categorization, which is the default in traces, mixes inherent workload events with events that depend on the infrastructure and the scheduler. In the context of this paper, such categorization does not help because workload events originate exclusively from users, while the rest depend on the executing environment. Apart from that, there is information in the trace that does not add value to a workload generator, e.g., the user's username that submitted a job. Therefore, propose a different categorization for the trace that is more suitable for our goal:

- *Workload events*, about *inherent* workload characteristics. They include the submission times of jobs and tasks.
- *Execution events*, with information about the *induced* execution in a specific environment. They include the schedule time and the finish time of a job/task.

2.1 Workload and Execution Events

Next, we show how to select the workload and execution events, focusing on the Google'11 trace for illustration purposes. We follow the same procedure for all traces, and summarize our findings in Table 1.

Table 1. Selected events of Google'11, Alibaba, and Google'19 traces.

Category	Type	Information
Workload	Job submitted	[time, job-id, sched class]
Execution	Job scheduled, finished	[time, job-id, priority]
	Job usage	[start, end, job-id, task-id, resources, metrics]
	Task submitted, scheduled, finished	[time, job-id, task-id, resources]

Job submit events are affected neither by the underlying infrastructure nor by the job/task scheduler. They originate from user requests and are static to recorded workloads. Therefore, we consider the job submit as workload events, while the rest as execution events. We choose only the events in the common path of a workload execution from the rest events, i.e. job "schedule, finish" and task “submit, schedule, finish” events. Additionally, we select information about the event time, and the type of the jobs, either batch or UF, for the submit events. For the schedule and finish events, we select information about the event time, the resource usage, and the system metrics they cause. We omit the events that concern failures or kill events, as they are specific to the recorded workload.

3 *Tracie* Model

We model a workload as a set of running tasks with certain parameters. To simplify the modeling procedure and without loss of generality, we consider that a workload is a mix of independent, batch (abbreviated as B, below), and user-facing jobs (abbreviated as UF), originating either from applications or services. We also assume that all tasks in each job are identical, therefore, tasks have the same duration and execute the same code. Typically jobs today, repetitively perform similar tasks. For instance, a Spark job contains tasks that perform the same computation on different partitions of a Resilient Distributed Dataset (RDD) [33]. Different jobs consist of different tasks.

We consider that the aforementioned assumptions still leave enough flexibility for expressing other types of workloads. For example, a job that contains tasks that are not identical can be modelled by *Tracie* as a collection of jobs, one for each type of task it contains. If we wish to model a job where all tasks are of a different type, we can do so by replacing it with as many one-task jobs as the different task types in the original job.

Therefore, our model consists of the following entities:

– A *workload* W is a list of jobs along with their arrival time $[Job, J_{AT}]$.
– A *job* J is a tuple $[J, J_N, J_D, B/UF, (Task, T_{AT})]$, where J is the job type, J_N is the number of job tasks, J_D is the duration of the job, B/UF indicates if a job is batch or UF, and $[Task, T_{AT}]$ is a list of task instances along with their arrival time.
– A *task* T is a tuple $[T, T_D, T_R]$, where T is the task type, T_D is the duration of the task, and T_R are the resource allocation requests.

Table 2 summarizes the parameters we use in our model and their correspondence to trace events. $J_{AT}[UF]$ and $J_{AT}[B]$ parameters correspond to the timestamp of a submit job event of UF and batch jobs. $J_N[UF]$ and $J_N[B]$ parameters correspond to the number of finish events of tasks belonging to the same UF and batch job. $T_{AT}[UF]$ and $T_{AT}[B]$ parameters are the timestamps of a submit UF and batch task event. Finally, $T_D[UF]$, $T_D[B]$, $J_D[UF]$, and $J_D[B]$ are calculated as the difference in the timestamp of schedule and finish events for UF and batch tasks and jobs respectively.

Table 2. Parameter definition and PDF estimation for Google 11, Alibaba, and Google 19 traces.

	Params	Desc	Trace event	Google'11	Alibaba	Google'19
Workload	$J_{AT}[UF]$	UF job arrival	Timestamp of submit job event	$Chi^2(0.3, -1.5e^{-10}, 1.3e^{15})$	$N(1.3e^{12},\ 7.2e^{11})$	$B(0.9, 1.1, -8.14e^{-11}, 2.5e^{12})$
	$J_{AT}[B]$	Batch job arrival		$R(2, 1.3e^{12}, 1.3e^{12})$	$R(713, 80, 2.4e^{4})$	$B(0.4, 4565, -7.4e^{-11}, 1.7e^{5})$
	$J_N[UF]$	UF job task cnt	Task finish event of the same job	$T(0.2, -1.5e^{5}, 6.6e^{7})$	$Exp(-2.5e^{12}, 2e^{12})$	$N(-5e^{11}, 7.64e^{11})$
	$J_N[B]$	Batch job task cnt		$T(0.3, 1, 1.5e^{-20})$	$KDE(N, 2)$	$KDE(N, 2)$
Execution	$T_{AT}[UF]$	UF task arrival	Timestamp of submit task event	$T(0.5, 0.006, 0.006)$	$B(212, 2.6e^{4}, -4.8e^{15}, 5.8e^{17})$	$N(6.8e^{13}, 6.8e^{14})$
	$T_{AT}[B]$	Batch task arrival		$KDE(N, 2.5)$	$KDE(B, 1.6)$	$KDE(N, 1.8)$
	$T_D[UF]$	UF task duration	Difference in the timestamp of	$KDE(N, 1.9)$	$KDE(B, 23)$	$KDE(N, 1.98)$
	$T_D[B]$	Batch task duration	schedule and finish task events	$KDE(N, 2.1)$	$KDE(B, 2.1)$	$KDE(N, 2.09)$
	$J_D[UF]$	UF job duration	Difference in the timestamp of	$B(0.3, 4e^{3}, -2.3e^{-27}, 3.7e^{3})$	$Chi^2(0.2,\ -1.1e^{-25}, 21)$	$N(1.3, 11)$
	$J_D[B]$	Batch job duration	schedule and finish job events	$T(0.2, -1.9e^{5}, 6.8e^{6})$	$B(8506, 16.6, -2.2e^{17}, 2.2e^{17})$	$N(-1.9e^{13}, 5.1e^{14})$

Next, we use the traces to extract appropriate values for each parameter. We model parameters as independent random variables. For each trace, we extract the histograms of the events that correspond to our parameters. Then, we identify the PDFs that best fit the histogram in whole or piece-wise and we use these PDFs as the value distributions of our model parameters. Table 2 summarizes the PDF that corresponds to each model parameter for each trace.

Depending on the histogram, we follow two different methods. If the histogram matches a common probability distribution, such as Normal, R, Chi-squared, T, Beta, Log-normal, Gamma, F, Exponential, Cauchy, Laplace, log-gamma, Chi, we apply Parametric Density Estimation (PDE) [14,31] to calculate its specific parameters, such as mean value and variance. To figure out which PDF to use, we perform multiple PDE tests with different PDF types and select the best fitting PDF that exhibits the minimum distance (least squares) with the given data-set. If the minimum distance is above 0.5, we consider that the histogram does not match a single probability distribution, and we resort to a Non-parametric Density Estimation (NDE) [20] technique, Kernel Density Estimation (KDE). KDE [16] models random variables as the concatenation of multiple instances of a single PDF kernel. KDE divides the histogram into fixed-size intervals (*bandwidth*), with each interval represented by the same kernel and different kernel parameters. KDE first identifies the kernel and bandwidth from the histogram [17] and then identifies the kernel parameters similar to PDE for each interval.

For Google 2011, we find that $J_{AT}[UF]$ follows a Chi-squared distribution with parameters $(0.31, -1.5e^{-10}, 1.26e15)$. $J_{AT}[UF]$ follows a R distribution with parameters $(1.98, 1.25e^{12}, 1.25e^{12})$. $J_N[UF], J_N[B], T_{AT}[UF]$, and $J_D[B]$ follow a T-distribution with parameters $(0.16, -1.45e^{5}, 6.63e^{7})$, $(0.25, 1.0, 1.53e^{-20})$,

$(0.48, 0.006, 0.006)$, and $(0.18, -1.91e^5, 6.8e^6)$. $J_D[UF]$ follows a beta distribution with parameters $(0.18, -1.91e^5, 6.8e^6)$. Finally, for $T_{AT}[B], T_D[UF]$, and $T_D[B]$, we apply KDE because they do not map well to any of PDF types we use for PDE. We find that these parameters match a Gaussian kernel, with respective bandwidths 2.46, 1.91, and 2.1.

3.1 Workload Scaling

We intend to run the generated workloads on different setups and infrastructure sizes. Therefore, there is a need to scale the workloads to match the intended infrastructure. The model parameters and their value distributions as extracted from available traces typically refer to large scale infrastructures, with task and job durations that exceed hours or even days, which is not practical or possible to follow on research prototypes and specific research problems. Running the workload on a different infrastructure requires scaling the workload to adjust the number of jobs, job durations, job arrival times, or data-set sizes. To achieve workload scaling, we introduce the following scaling factors:

– The total number of jobs in a workload, W_N. With W_N, we control the number of jobs per server and per time unit.
– A parameter for the scaling of job arrival times, W_{SAT}. With W_{SAT}, we control how loaded the servers will be during execution.
– A parameter for the scaling of the duration of batch and UF jobs, J_{SD}. J_{SD} parameter changes the dynamics of the batch and UF jobs in a workload. This parameter is useful to investigate how the infrastructure and the system software copes with changes in the behavior of the workload. For instance, when throughput is more important than latency, i.e. when the duration of batch jobs is significantly higher than UF jobs and vice versa.

3.2 Application Selection

In this step, we select the application types that will be used for executing the trace-based workload. We base the selection of applications on several trace events and the characteristics they represent: 1) maximum and average CPU, 2) memory, disk, and network utilization, 3) cycles per instruction and 4) memory accesses per instruction. First, we process each trace individually to generate histograms for these parameters. Then we perform dimensionality reduction, using Principal Component Analysis (PCA) [21]. This step is essential for application selection because of the large number of parameters, which complicates application type estimation. After dimensionality reduction, we perform PDF estimation, similar to our model parameters (Sect. 3). Table 3 shows the resulting parameters with their corresponding PDF type and parameters for each trace. The procedure to define the trace application pool requires access to various application types. For the purposes of this work, we select the batch applications from the Rodinia benchmark suite [9] and the TPC family [26]. Also, we select services among the following cloud services: NGINX [27], Redis [5], CouchDB [3], and Memcached [4]. However, it is not in the scope of this work to provide representative UF or batch applications. We assume that users provide suitable application pools depending on their use cases. In addition, we do not further differentiate the importance of some application types over others. Therefore, during workload execution, we uniformly select an application out of such a pool.

Table 3. Application selection features extracted for WTs.

Params	Desc	Google'11	Alibaba	Google'19
ACU	Avg. CPU	$T(0.8, 2e^{-11}, 2.7e^{-11})$	$T(1, 1e^{-11}, 3.7e^{-11})$	$T(0.3, 3, 1.3e^{-9}, 6.6e^{-11})$
MU	Avg. Mem	$Norm(0.01, 0.05)$	$B(0.5, 674, -6.3e^{-30}, 2.2)$	$B(0.05, -3.5e^{-32}, 1.2)$
MM	Max Mem	$G(0.05, -3.5e^{-32}, 1.2)$	$Exp(0.0, 4.4)$	N/A
N_{IN}	Net in	N/A	$Exp(0.03, 0.2)$	$Exp(0.5, 0.8)$
N_{OUT}	Net out	N/A	$B(0.2, 11.3, -1.2e^{-25}, 1.4)$	$B(0.5, 203, -3.5e^{-33}, 5.4)$
IO	IO	$B(0.6, 172, -6.7e^{-33}, 1.56)$	N/A	$B(0.3, 490, -9.9e^{-33}, 0.9)$
PG	Page cache	$Exp(0.02, 0.03)$	N/A	N/A
PG	Page cache	N/A	N/A	$G(0.6, -1.1e^{-31}, 0.07)$

3.3 Similarity Validation

To validate the similarity of the generated execution-based workloads to the corresponding trace characteristics, we capture system metrics from the actual execution of the generated workload and compare them to the trace events, for each model parameter. To compare the two data-sets, trace vs. measured, we use the Pearson correlation coefficient [8].

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}, \tag{1}$$

where n is the total number of samples for both data-sets, x_i is the i-th sample of the first data-set and $\overline{x}$ its corresponding mean value, y_i is the i-th sample of the second data-set and $\overline{y}$ its corresponding mean value. The range of values that r_{xy} can take is $[-1, 1]$.

For this computation, the two data-sets need to have the same size. Therefore, we randomly divide the data-set of the trace to N subsets with a size equal to the synthetic workload data-set, and calculate their Pearson correlation coefficients. N is the quotient of the size of the trace data-set to the size of the synthetic data-set. We then calculate the mean value of the resulting r_{xy} coefficients. If r_{xy} is close to 1 (or -1), then the two data-sets are highly correlated (positively or negatively). If r_{xy} is close to 0, the two data-sets are not linearly correlated.

4 Implementation

Tracie is a Python script that produces executable workloads. The user can generate a workload based on one of the trace profiles of *Tracie*. A profile mainly contains the type of PDFs and their parameters for the random variables of Table 2. Each profile is stored in a separate directory which becomes a parameter to *Tracie*, e.g. ./wlGenerator.py –profile = "Google 2011". We expect that *Tracie* will be augmented over time with additional profiles from new traces, based on our methodology of Sect. 3. After specifying a profile, users can scale the workload to match their setup with three parameters: 1) the number of jobs, 2) factor for job arrival time, and 3) factor for the job duration of batch and UF jobs. The number of jobs defines how many jobs the generator will create, e.g. ./wlGenerator.py −n = 80, will create 80 jobs. The factor for job arrival time is a number that is multiplied by the arrival time of a job, e.g. ./wlGenerator.py –wSat = 2, will double the arrival times of jobs. Finally, the factor for the duration of the jobs is a number that changes the number of tasks to change the duration of jobs, e.g. ./wlGenerator.py –jSD = 2, will double the number of job tasks. Therefore, *Tracie* allows users to execute diverse workloads of the same load factor and run a workload at different load factors (scales). The emphasis in the former is that every time the generator selects different tasks and other parameters (based on PDF profiles). In the latter case, users can fix other parameters and change the induced load. In both cases, the generated loads will run on the given infrastructure using the sample applications provided along with *Tracie*.

Tracie consists of two main modules: the *workload generator* and the *workload executor*. The workload generator receives as input the workload profile, encoded in custom data type **wlProfile**. Further inputs are J, which is the number of desired jobs in the workload. The workload generator (Pseudocode 1) produces a *job sequence* as output. This output contains a sequence of job instances, specifying the characteristics for each instance. The workload generator is used offline to produce a job trace before a test run is performed.

To calculate a job duration J_D, we create by $N = J_D/T_D$ tasks. The output of the *workload generator* is two files: (1) a sequence of jobs where each line describes a single job, i.e. $J = [J_AT, J_T, T]$ and (2) a sequence of tasks per job, where each line is list of timestamps defining their arrival time within jobs. The workload executor parses the generated job sequence and the task arrival sequence to generate and execute the tasks for each job. The tasks within a job execute the same code on the same data.

Algorithm 1. Workload generation.

```
1: procedure wlGenerator(wlProfile $WP$, int $J$, int $F$)
2:     int $jc, N := 0$                    ▷ job counter, number of tasks, respectively
3:     time $ts, t_{total}, J_D, A_D := 0$     ▷ job timestamp, total time, job duration, application duration, respectively
4:     Boolean $S$      ▷ scheduling class, may either be B, indicating batch job, or UF, indicating user-facing job
5:     appType $T$        ▷ Application name, out of pool of codes available to the tool
6:     jobTypeSeq $W$                  ▷ A sequence of jobs, the workload to output
7:     while $jc < J$ do
8:         $rand :=$ random number between 0 and 100
9:         if $rand < P_B$ then
10:            $< T, J_N, ts > :=$ generateJob($B$, $WP$, $t_{total}$)
11:        else
12:            $< T, J_N, ts > :=$ generateJob($UF$, $WP$, $t_{total}$)
13:        end if
14:        $t_{total} := ts$
15:        $W.append(< jc, T, J_N, F, ts >)$
16:    end while
17:    return $W$
18: end procedure
```

Workload execution

```
19: function executeWorkload(Boolean $S$, wlProfile $WP$, time $t_{total}$)
20:     if $S = batch$ then
21:         $J_D = WP.J_D[B](WP.\lambda_B)$
22:     else
23:         $J_D = WP.J_D[UF](WP.\lambda_{UF})$
24:     end if
25:     Select an app $A$ of type $T$ and its configuration out of pool of available apps,
        determine its duration $A_D$.
26:     $J_N = J_D/T_D$
27:     $ts = t_{total} + WP.J_{AT}(WP.\lambda_{AT})$
28:     return $< J, J_N, ts >$
29: end function
```

5 Experimental Evaluation

In this section, we first evaluate our methodology for PDE, KDE, and application selection. Afterwards, we use *Tracie* to reproduce synthetic workloads based on Google and Alibaba traces. In our experiments, we use a server with three 16-core AMD Opteron 6200 64-bit processors, running at 2.1 GHz, and 48 GB of DDR-III DRAM. For storage, we use a Samsung EVO 850 128 GB SSD.

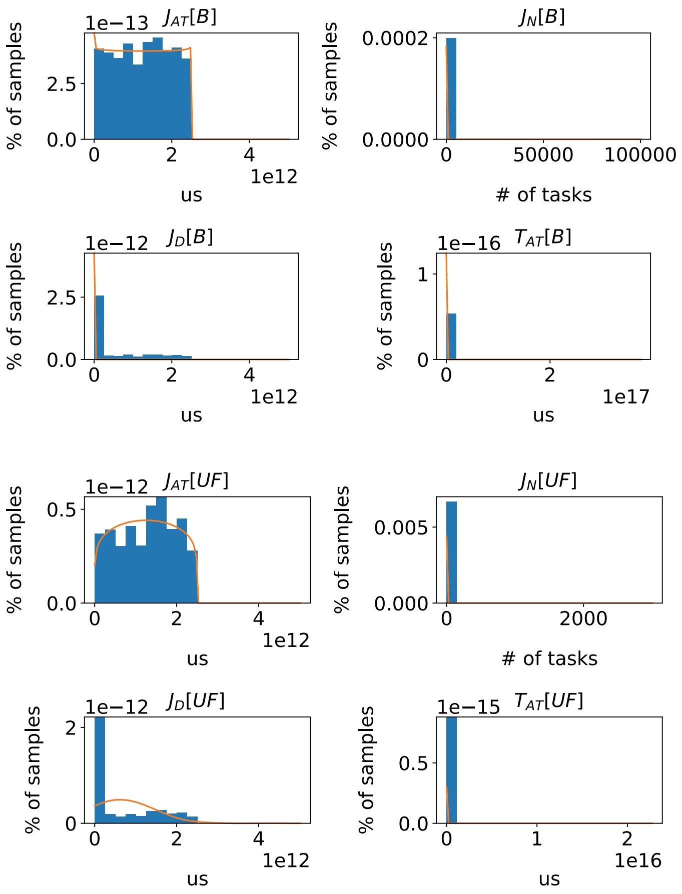


Fig. 1. Histogram and estimated PDFs using PDE for Google'11.

Table 4. Parameter similarity of PDFs and histograms (Google'11).

Parameter	$J_{AT}[B]$	$J_N[B]$	$J_D[B]$	$T_{AT}[B]$	$J_{AT}[UF]$	$J_N[UF]$	$J_D[UF]$	$T_{AT}[UF]$
Similarity (%)	94%	88%	72%	61%	91%	78%	55%	59%

5.1 Evaluating PDE

This section evaluates the accuracy of the PDFs computed by *Tracie* concerning the histograms of the corresponding parameters extracted from traces. Figure 1 compares the histograms of Google'11 parameters with the corresponding PDFs of *Tracie*. We observe that only the $J_D[UF]$ parameter is not very close to the histogram. To measure the similarity between PDFs and histograms, we compute their mean square distance. Table 4, shows the percentage of differences between histograms and PDFs. *Tracie* achieves the best distance for parameter $J_{AT}[B]$, for which the PDF and the histogram are 94% similar. The worst case is

$J_D[UF]$, for which the PDF is only 55% similar to the histogram, and on average, the PDFs are 75% similar to their histograms. Therefore, the characteristics of synthetic workloads of *Tracie* are quite close to the ones of the originating trace.

5.2 Dimensionality Reduction for Application Pools

Tracie reduces a large number of event types in each trace to a smaller number that can be used for application selection using PCA. Table 5 summarizes the importance of each event type as characterized by PCA. Event coefficients in bold indicate the most critical event types for each trace. We show with bold font the events of the traces that we select for application selection. The parameters average CPU and average memory usage are significant for all traces. Apart from that, max memory usage is critical for Google'11 and Alibaba traces while net in and net out for Alibaba and Google'19 trace. Finally, page cache usage and max IO usage are significant for Google'11 trace, while max CPU usage for Google'19. At most 6 parameters are sufficient to represent at least 75% of the micro-architectural characteristics of all traces. Cloud applications are diverse and can vary in all of these characteristics significantly. To reduce the parameter space, we focus only on the trace critical micro-architectural parameters.

Table 5. Applying PCA on execution parameters.

Parameter	avg_cpu	mem_usage	page_cache	max_mem	avg_disk_IO	disk_space	max_cpu
Google'11	**0.242**	**0.162**	**0.1**	**0.168**	0.0783	**0.095**	0.024
Alibaba	**0.176**	**0.12**	0.0718	**0.12**	0.053	0.078	0.018
Google'19	**0.136**	**0.098**	N/A	0.054	**0.099**	0.035	**0.24**
Parameter	max_IO	net_in	net_out	cpi	mai	cpu_distr	cpu_tail_distr
Google'11	**0.113**	N/A	N/A	0.005	0.018	N/A	N/A
Alibaba	0.075	**0.124**	**0.147**	0.004	0.013	N/A	N/A
Google'19	0.049	**0.107**	**0.103**	0.003	0.01	0.014	0.052

Table 6. Applying PCA on execution parameters: Importance of each parameter selected for workload execution.

Parameter	avg_cpu	mem_usage	page_cache	max_mem	avg_disk_IO	disk_space	max_cpu
Google'11	0.791	0.5	0.623	0.886	0.614	0.463	0.838
Alibaba	0.696	0.74	0.877	0.645	0.841	0.837	0.822
Parameter	max_IO	net_in	net_out	cpi	mai	cpu_distr	cpu_tail_distr
Google'11	0.5	0.655	0.613	0.706	0.531	0.77	0.734
Alibaba	0.834	0.655	0.568	0.757	0.636	0.739	0.829

5.3 Emulating the Google and the Alibaba Trace

This section generates and executes synthetic workloads of *Tracie*, based on Google and Alibaba. We then validate how representative the execution of these workloads is by comparing the system metrics of the synthetic workload execution to the ones in the trace. Figure 2 shows the results of two experiments, where we execute a workload based on the Google trace and a workload based on the Alibaba trace. By applying our validation methodology in both traces, we get a similarity coefficient among all usage parameters above 0.46 and, on average, 0.69. Table 6 shows the correlation coefficients for all parameters, for both workloads. Additionally, comparing the two synthetic workloads, we observe that the Alibaba workload is more bursty than the Google trace, which is why it results in a worse tail latency for the user-facing tasks. Moreover, we observe that the workload of the Alibaba trace has 100% load in the first 120 s and afterwards cools down, while in the Google trace load is more balanced over time.

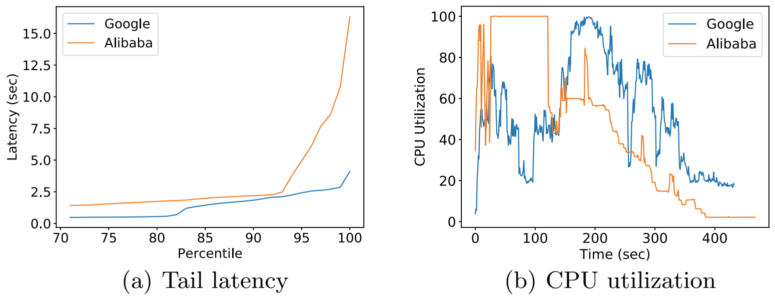


Fig. 2. Two Workloads inspired by the Google and Alibaba traces, emulating the average server in each trace. (a) Tail latency of the tasks, (b) CPU utilization.

Google: According to the Google trace analysis [28], the average server is 50% utilized, the batch to user-facing job duration ratio is 0.5, and has 38% batch jobs. We create a workload using *Tracie* that follows the statistics of the Google trace for a single server and set 35 s as the average batch job duration. To scale down the workload, we estimate the scaling factors of *Tracie*, i.e. number of jobs, job arrival time, duration ratio of batch vs. UF jobs, by calculating the corresponding average values of Google servers. *Tracie* produces a workload with 52% CPU utilization and an average job arrival time of 10 s.

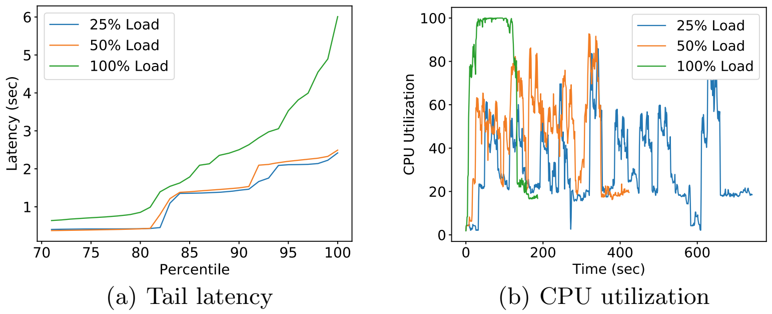
The trace released by Google has been studied extensively in several publications e.g. [6,11,12,22,23,25,28,29]). The trace has over 670,000 jobs and 25 million tasks executed over 12,500 hosts during 1-month time period [13]. Around 40% of submissions recorded are less than 10 ms after the previous submission even though the median arrival period is 900 ms. The tail of the arrival times distribution is power-law-like, though the maximum job arrival period is only 11 min. Jobs shorter than two hours account represent more than 95% of the jobs, and half of the jobs run for less than 3 min. The majority of jobs runs for less than 15 min [23].

Alibaba: Correspondingly, in Alibaba trace, the average server is 40% utilized, the batch to user-facing job duration ratio is 0.05, and it contains 53% batch jobs. Similarly, we create a workload with *Tracie*, selecting 4 s for the average batch job duration. The workload results in 45% CPU utilization and average job arrival time 680 ms. In both cases, the workload generated by *Tracie* is very close to the average statistics of servers as per the original traces.

The Alibaba trace is analyzed in [24]. It contains 11089 user-facing jobs and 12951 batch jobs, which run over a time period of 12 h. This places the batch job versus the user-facing job ratio at 53.9% to 46.1%. User-facing jobs in the Alibaba trace are long-running service jobs, in this particular case spanning the entire duration of the trace. On the contrary, batch jobs in the trace are predominantly short-running, with about 90% of batch jobs running in less than 0.19 of an hour, while a total of 98.1% of batch jobs runs in less than an hour. Overall, 47.2% of total jobs are long, whether batch or user-facing.

5.4 Investigating How Scaling Affects the Tail Latency

Finally, we show that with *Tracie* we can easily explore how the workload is affected when we change one of its scaling factors. In this experiment, we produce a workload that emulates the workload of a Google server according the Google trace on average. We examine how scaling the job arrival time affects the tail latency of jobs by running the workload 3 times with a different value scaling factor of J_{AT}. Figure 3 summarizes these three runs. Figure 3(a) shows the tail latency of all the user-facing tasks starting from 70th to the 100th percentile, while Fig. 3(b) shows the corresponding CPU utilization. In the first experiment (green line), we target a load of 25% on average. In the second (orange line), we target 50%, and in the third (blue line), we target 100%. The average utilization of the system is 31%, 46%, and 95% respectively, which indicates that we can successfully control the utilization of the system with *Tracie*, by just changing



(a) Tail latency (b) CPU utilization

Fig. 3. Trade-off between CPU utilization and tail latency: (a) Tail latency of user-facing applications, for increasing system load. (b) CPU utilization.

the J_{AT} scaling factor. Moreover, the total execution time of the experiment changes almost in reverse proportion to the load of the system. The experiment finishes in 700 s for 25% load, 400s for 50% load, and 200 s for 100% load. Tail latency is not affected for the runs with 25% and 50% load. However, for the case with 100% load, tail latency suffers a 2× deterioration as we approach the 100-percentile of the tasks. It is not straightforward to increase the utilization of the system while still achieving low tail latency for the user-facing tasks.

6 Related Work

We classify prior work on workload generation in two main categories: (a) Benchmark suites with popular cloud applications to generate workloads mixes.

Benchmark Suites: BigDataBench [15] is a benchmark suite that provides benchmarks for online services, offline analytics, graph analytics, AI, data warehouse, NoSQL, and streaming. Contrary to our work, which focuses on using kernels as computation units, BigDataBench combines commonly occurring data characteristics (data motifs) with a set of micro-benchmarks to form computational building blocks. Subsequently, it uses the most common data motifs to compose complex workloads for a real-life software stack. DCMIX [32] combines a set of characteristic computation units to come up with a collection of benchmarks. DCMIX resembles *Tracie* in that it provides the functionality to execute a workload, as specified in a configuration file. However, *Tracie* also provides the capability to create arbitrary synthetic workloads, and goes one step further by profiling real-life datacenter traces to provide a collection of realistic parameter values. HiBench [19] is a benchmark suite for Hadoop. Compared to *Tracie*, BigDataBench, DCMIX, and HiBench do not consider real-life datacenter traces to provide parameter values based on realistic loads.

Workload Traces and Trace Generators: Several synthetic workload generators provide benchmarks that target specific operation aspects of a datacenter. BigBench [10] is a work that, similarly to ours, deals with the extraction of characteristics of realistic workloads and their use to generate synthetic workloads. However, BigBench focuses on a particular use case, a system handling the transactions of a big retailer, both physical and online. As such, the result is rather restricted to workloads that consist of database queries, contrary to our work, which aims at producing synthetic workloads for a more generic application domain. Some more generic efforts to produce synthetic workloads with realistic characteristics appear elsewhere in the literature. Due to the wide-spread use of this particular computing paradigm, two prominent synthetic workload generation examples, [7] and [2], deal with the synthesis of MapReduce workloads. GridMix [7] is a benchmarking tool by Apache Hadoop, which creates synthetic workloads suitable for testing Hadoop clusters. These traces are used as input to GridMix, which in turn outputs a synthetic workload. The load can be scaled up or down by adjusting the intervals between job submission. Several inherent and run-time characteristics of the jobs, such as memory usage, input data size, or job type, can be modeled. Similarly, SWIM [2] generates synthetic workloads based on the traces released by Facebook in 2009 and 2010. SWIM aims to produce workloads of shorter duration than the original traces while still maintaining their essential characteristics. CloudMix [18] is a benchmarking tool that synthesizes cloud workloads with realistic characteristics. Similar to our use of a pool of kernels, CloudMix relies on a repository of *reducible workload blocks* (RWBs), representing different mixes of assembly-level computations designed to mimic micro-architectural usage characteristics of tasks found in the Google trace. Workload scale-down is possible by reducing job and trace durations. Unlike our work, CloudMix is more concerned with reproducing the run-time behavior (in terms of micro-architectural characteristics) observed in the trace.

7 Conclusions

In this paper, we determine characteristics that describe datacenter workload traces. We use these characteristics to develop a methodology for generating synthetic workloads to execute on existing systems. We implement this methodology in *Tracie* and validate that the execution of applications in such workloads exhibits similar micro-architectural characteristics as the ones captured in the original trace. The flexibility of our methodology and *Tracie* lies in that they are not designed to mimic one particular datacenter workload. Instead, they can be configured to reproduce any desired profile, starting from a set of statistical characteristics extracted from traces. Furthermore, while we employ a specific set of application kernels as sample binaries for task execution, neither the methodology nor the supporting tools are hard-wired to this selection.

Acknowledgments. We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the EVOLVE H2020 project (Grant Agreement Nr 825061).

References

1. ab Benchmark - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html
2. Swim. https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository
3. The Apache CouchDB. https://couchdb.apache.org/
4. The Memcached I/O cache. https://memcached.org/
5. The Redis Database. https://redis.io/
6. Abdul-Rahman, O.A., Aida, K.: Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014)
7. Apache: GridMix. https://hadoop.apache.org/docs/r1.2.1/gridmix.html
8. Benesty, J., Chen, J., Huang, Y., Cohen, I.: Pearson correlation coefficient. In: Noise Reduction in Speech Processing. Springer Topics in Signal Processing, vol. 2, pp. 1–4. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00296-0_5

9. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)
10. Chen, Y., Alspaugh, S., Katz, R.: Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. arXiv preprint arXiv:1208.4174 (2012)
11. Chen, Y., Ganapathi, A.S., Griffith, R., Katz, R.H.: Analysis and lessons from a publicly available Google cluster trace. Technical report. UCB/EECS-2010-95, EECS Department, University of California, Berkeley, June 2010. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-95.html
12. Di, S., Kondo, D., Cappello, F.: Characterizing and modeling cloud applications/jobs on a Google data center. J. Supercomput. **69**, 139–160 (2014). https://doi.org/10.1007/s11227-014-1131-z
13. Di, S., Kondo, D., Cirne, W.: Characterization and comparison of cloud versus grid workloads. In: IEEE Cluster (2012)
14. Efron, B., Tibshirani, R., et al.: Using specially designed exponential families for density estimation. Ann. Stat. **24**(6), 2431–2461 (1996)
15. Gao, W., et al.: Bigdatabench: A scalable and unified big data and ai benchmark suite. arXiv preprint arXiv:1802.08254 (2018)
16. Gray, A.G., Moore, A.W.: Nonparametric density estimation: toward computational tractability. In: Proceedings of the 2003 SIAM International Conference on Data Mining, pp. 203–211. SIAM (2003)
17. Guidoum, A.C.: Kernel estimator and bandwidth selection for density and its derivatives. The Kedd package, version 1 (2015)
18. Han, R., Zong, Z., Zhang, F., Vazquez-Poletti, J.L., Jia, Z., Wang, L.: Cloudmix: generating diverse and reducible workloads for cloud systems. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD)
19. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The hibench benchmark suite: characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW). IEEE (2010)
20. Izenman, A.J.: Review papers: recent developments in nonparametric density estimation. J. Am. Stat. Assoc. **86**(413), 205–224 (1991)
21. Karamizadeh, S., Abdullah, S.M., Manaf, A.A., Zamani, M., Hooman, A.: An overview of principal component analysis. J. Signal Inf. Process. **4**(3B), 173 (2013)
22. Liu, B., Lin, Y., Chen, Y.: Quantitative workload analysis and prediction using Google cluster traces. In: 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS),pp. 935–940 (2016)
23. Liu, Z., Cho, S.: Characterizing machines and workloads on a Google cluster. In: Proceedings of the 2012 41st International Conference on Parallel Processing Workshops, ICPPW 2012, IEEE Computer Society, Washington, DC
24. Lu, C., Ye, K., Xu, G., Xu, C.Z., Bai, T.: Imbalance in the cloud: an analysis on Alibaba cluster trace. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 2884–2892. IEEE (2017)
25. Moreno, I.S., Garraghan, P., Townend, P., Xu, J.: An approach for characterizing workloads in Google cloud to derive realistic resource utilization models. In: SOSE, pp. 49–60. IEEE Computer Society (2013)
26. Nambiar, R., Wakou, N., Carman, F., Majdalany, M.: Transaction Processing Performance Council (TPC), State of the council (2010)
27. Nedelcu, C.: Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever. Packt Publishing Ltd., Birmingham (2010)

28. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing (2012)
29. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Towards understanding heterogeneous clouds at scale: Google trace analysis (2012)
30. Tirmazi, M., et al.: Borg: the next generation. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–14 (2020)
31. Varanasi, M.K., Aazhang, B.: Parametric generalized gaussian density estimation. J. Acoust. Soc. Am. **86**(4), 1404–1415 (1989)
32. Xiong, X., et al.: DCMIX: generating mixed workloads for the cloud data center. In: Zheng, C., Zhan, J. (eds.) Bench 2018. LNCS, vol. 11459, pp. 105–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32813-9_10
33. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., et al.: Spark: cluster computing with working sets. HotCloud **10**(10), 95 (2010)