SDCBench: A Benchmark Suite for Workload Colocation and Evaluation in Datacenters

Yanan Yang[1], Xiangyu Kong[2], Laiping Zhao[1*], Yiming Li[1], Huanyu Zhang[1], Jie Li[1], Heng Qi[2], and Keqiu Li[1]

[1]Tianjin Key Lab. of Advanced Networking, College of Intelligence and Computing, Tianjin University, China.

[2]School of Computer Science and Technology, Dalian University of Technology, Dalian, China.

[*]Corresponding author. Email: laiping@tju.edu.cn

Abstract

Colocating workloads are commonly used in datacenters to improve server utilization, However,the unpredictable application performance degradation caused by the contention for shared resources makes the problem difficult and limits the efficiency of this approach. This problem has sparked research in hardware and software techniques that focus on enhancing the datacenters' isolation abilities. There is still lack of a comprehensive benchmark suite to evaluate such techniques.

To address this problem, we present SDCBench, a new benchmark suite that is specifically designed for workload colocation and characterization in datacenters. SDCBench includes 16 applications that span a wide range of cloud scenarios, which are carefully selected from the existing benchmarks using the clustering analysis method. SDCBench implements a robust statistical methodology to support workload colocation and proposes a concept of latency entropy for measuring the isolation ability of cloud systems. It enables cloud tenants to understand the performance isolation ability in datacenters and choose their best-fitted cloud services. For cloud providers, it also helps them to improve the quality of service to increase their revenues. Experimental results show that SDCBench can simulate different workload colocation scenarios by generating pressures on multidimensional resources with simple configurations. We also use SDCBench to compare the latency entropies in public cloud platforms such as Huawei Cloud, AWS Cloud and a local prototype system FlameCluser-II, the evaluation results show FlameCluser-II has the best performance isolation ability over these three cloud systems, with 0.99 of experience availability and 0.29 of latency entropy.

1 Introduction

Cloud computing has grown rapidly over the past few decades and is widely used in many application areas, such as web service, databases, big data processing and machine learning [1, 2]. Virtualization technologies allow end users to share cloud resources in the form of virtual machines (VMs) with an on-demand provision model [3]. By abstracting the underlying hardware resources, the server utilization can be improved through workload consolidation [4]. However, the contention for shared resources such as CPU, Last Level Cache (LLC) and memory bandwidth between VMs causes performance interference, especially in multitenant cloud scenarios [5–7].

Interference may result in unpredictable performance degradation for cloud services, which not only reduces the user experience but also hurts resource efficiency in datacenters [8, 9]. For example, the pressures on multidimensional resource consumption at the instruction cycle level become more intractable, cause a long tail latencies for interactive applications. Additionally, servers running latency-critical services can only operate at low utilization due to their unpredictable tail latency. The performance interference makes it difficult to utilize the spare server capacity by colocating batch applications since uncontrolled sharing of CPU cores, caches and power causes high latency degradation. As a result, the average server utilization in most datacenters is only 10%-50% [10, 11]. This leads to billions of dollars of wastage in infrastructure and energy every year [12].

To mitigate the performance interference caused by shared resource contention, many researchers seek to enhance the isolation ability in cloud systems from hardware and software approaches. The hardware methods such as Intel RDT [13] and PARD [14], provide control interfaces for partitioning hardware such as LLC ways and memory bandwidth between different colocated applications, thus reducing the contention on these microarchitecture level resources. The software methods commonly adopt mechanisms like resource overprovisioning, CPU core binding and dynamical power management to protect latency-critical services from the interference of colocated workloads [4, 15]. However, utilizing these techniques in datacenter requires new hardware support or system upgrades. Not all of the cloud providers are willing to implement such optimizations in their platforms, which also leads to service performance differences between cloud providers. The need for predictable service performance in datacenters brings new challenges and opportunities for cloud system design that seek to improve *server-level resource utilization* but do not hurt *application-level performance*.

Unfortunately, the lack of a comprehensive suite of workload colocation benchmarks makes studying this emerging problem challenging. First, it hampers research that seeks to analyze the causes of application interference. Latency-critical services (LCs) have a variety of latency requirements and microarchitectural characteristics. The performance degradation may come from the pressures on different shared resource contentions or their combinations. However, most benchmarks in prior works are not designed for evaluating the performance interference of workloads in multitenant shared cloud scenarios [1, 16–18], and they are unable to exert a wide range of pressures on the underlying hardware resources. Additionally, they only observe the service-level performance and do not support the measurement in system isolation ability. Second, most algorithm or architecture innovations in cloud systems are focused on throughput-oriented designs to provide better resource pooling or provisioning abilities [10, 19]. This constitutes a blind spot in the interference measure-

ment and explorations of new isolation techniques. For example, many scheduling frameworks adopt optimized algorithms to improve the efficiency of allocating virtual machines (VM) or jobs [20–22]. While insufficient hardware isolation mechanisms may dramatically worsen the application performance of VMs (e.g., increasing job completion time). Nevertheless, none of the existing benchmark suites support the measurement of performance isolation ability in diverse cloud scenarios.

A workload colocation benchmark can help cloud providers understand and improve the infrastructures' isolation capabilities, thereby increasing their adoption by cloud users [8]. Designing such a benchmark is challenging for several reasons. First, the applications must be carefully selected and cover a comprehensive range of domains and multidimensional resource usage behaviors. Similar applications will make the benchmark redundant and not easy to use. Second, the service performance degradation caused by interference may present in many system-level and micro-architecture metrics (e.g., tail latency, IPC). How to characterize the system uncertainty by consolidating these observation changes is a challenge. Third, it is not enough for these workloads to run individually on systems. Instead, the benchmark should support flexible mixing of workload types and intensities to adapt different application colocation requirements in datacenters.

To solve these problems, we present SDCBench, a benchmark suite for workload colocation that addresses these challenges. SDCBench includes a diverse set of LC services and BE applications, as well as a robust, validated experimental methodology that makes it easy to colocate these benchmarks on cloud systems and measure the datacenters' resource isolation abilities. Our key contributions can be summarized as follows:

- We present SDCBench, a new benchmark suite for the isolation ability measurement in multitenant shared cloud systems that covers a wide spectrum of workload diversity and characterization.
- We propose a concept of latency entropy to describe the application performance degradation arising due to resource contention in cloud systems. This enables one to quantify the system isolation ability and efficiency of hardware and software partition technologies in workload colocation scenarios.
- We implement a comprehensive evaluation framework based on SDCBench that can automatically configure, deploy, and evaluate applications on cloud platforms. The framework is open source and can be easily extended to new cloud systems.
- We demonstrate SDCBench' usability and show that it can simulate different cloud scenarios by colocating workloads with simple configurations. We also evaluate today's major cloud providers using SDCBench and present the comparison of their performance entropies.

Table 1: Comparison of existing benchmarks.

Benchmarks	# of apps.	Types	Application Domains	Service Metrics	System Metrics	Colocation Support
LINPACK [23]	2	BGs	High performance computing	JCT	GFLOPs	-
SPEC CPU [24]	8	BGs	CPU performance	JCT	GFLOPs	-
HPCC [25]	5	BGs	High performance computing	JCT	GFLOPs	-
PARSEC [26]	8	BGs	Multi-cores performance	JCT	GFLOPs	-
SPEC Cloud_IaaS [16]	2	LCs+BGs	Cloud system performance	Latency, JCT	IPC,CPU util.	Yes
YCSB [27]	5	LCs	Online services for NoSQL system	Latency	-	-
CloudSuite [1]	8	LCs+BGs	Cloud applications	Latency, JCT	-	Yes
TailBench [17]	8	LCs	Interactive services	Tail Latency	-	-
BigdataBench [18]	33	LCs+BGs	Bigdata applications	Tail Latency, JCT	-	Yes
PerfKit [28]	35+	LCs+BGs	Muitl-types applications	Tail Latency, JCT	-	Yes
uSuite [29]	4	LCs	data-intensive interactive applications	Tail Latency	-	-
DeathstartBench [30]	6	LCs	Fanout, multi-layers Microservices	Tail Latency	-	-
MLPerf [31]	12+	LCs	Machine Learning Inference	Tail Latency	-	-
ServerlessBench [32]	12+	LCs	Serverless functions	Tail Latency	-	-

2 Materials and Methods

2.1 Background and Motivation

2.1.1 Interference in Shared Cloud Scenarios

Large-scale online services such as e-commerce, search engines, online maps, social media and advertising are widespread in today's datacenters. These interactive, latency-critical services are usually scaled across thousands of servers with fanout or multitiered architecture [33]. The intermediate state and accessible data are stored in memory or flash distributedly to ensure fast response time. A large number of microservices across multiple leaf nodes may collaborate to serve a user request. As the overall latency presented to the user is determined by the slowest nodes, even small interference, queue delays or other sources of performance variations in these nodes may cause dramatic service time increases (a.k.a. tail latency). For example, the tail latency of the web search service in Google ranges from 0-500 ms, and the highest variation difference can exceed 600× [34].

The requirements for low and predictable tail latency of latency-critical applications limit server resource utilization in datacenters [4, 35]. On the one hand, the workload of interactive services varies significantly due to the diurnal patterns and unpredictable bursts in user accesses. Cloud providers have to allocate resources to these services for their peak loads. This leads to much wastage due to resource overprovisioning. On the other hand, it is difficult to utilize the spare capacity by colocating batch applications with them, as the interference from sharing CPU cores, cache and power causes high tail latency degradation and even violates the latency SLO.

Achieving high performance isolation in cloud systems has been a key challenge in improving the resource efficiency of datacenters. However, researchers have proposed many approaches to reduce the performance interference between colocated workloads in cloud systems in order to improve server utilization. The lack of a comprehensive benchmark suite hampers the researchers in this area to evaluate their new proposed methods, and for cloud users, it is also hard to help them understand their application performance in different cloud platforms. In the following, we will explain why the existing benchmarks do not address this problem.

2.1.2 Limitation of Existing Benchmark Suites

Prior work has proposed a variety of benchmarks, including both latency-critical services and background applications that can be deployed in cloud systems to help researchers in this area. These benchmarks fall short our needs from the standpoints of workloads diversity and performance metrics in interference measurement and studies. In the following, we compare SDCBench with some of the representative benchmarks from these aspects.

Table 1 lists the existing benchmark suites for evaluating computing system performance. LINPACK [23], SPEC CPU [24], HPCC [25] and PARSEC [26] are benchmarks designed in the evaluations of high performance computing (HPC) areas, which include several well-written programs running on computing hardware or simulators. The main focus of these applications is on the peak speed of CPU processors such as GFLOPs, which impacts the job completion time (JCT) of tested applications. Different from the SPEC CPU, the SPEC Cloud_IaaS [16] is a specially designed version for evaluating applications in cloud systems. It includes both latency-critical services and background applications in the benchmark and supports the measurement of service latency. However, SPEC Cloud_IaaS only provides two applications and is not representative of most of today's cloud scenarios.

Several benchmarks, such as YCSB [27], CloudSuite [1], TailBench [17] and uSuite [29], focus on performance measurements in cloud services. Among them, YCSB is a cloud benchmark specifically for data storage systems, which providers key-value pairs of queries from the NoSQL database. Similar to YCSB, µSuite includes four data-intensive interactive applications that are designed for the measurement in microarchitecture metrics such as system calls, context switches, and other OS overheads. CloudSuite and BigDataBench [18] provide both latency-critical and throughput-oriented applications to evaluate the microarchitectural traits that impact the performance of these services. However, their load testers adopt a "closed-loop" system and lack a rigorous latency measurement methodology. TailBench aggregates a set of interactive benchmarks and proposes a more accurate methodology to measure the tail latency. However, all of these benchmarks are designed to measure the monolithic application performance, which can not reflect the difference of performance isolation ability in cloud systems.

Other benchmarks, such as DeathstarBench [30], MLPerf [31] and ServerlessBench [32], target specific application domains in cloud datacenters. For example, DeathstarBench is a recent open-source benchmark suite for cloud and IoT microservices, which includes representative services such as social networks, video streaming, E-commerce, and swarm control services. MLPerf is an industry-academic benchmark suite for machine learning, that facilitates system-level performance measurements and comparisons on diverse software platforms, such as TensorFlow [36] and PyTorch [37] as well as hardware architectures. ServerlessBench is a benchmark designed for serverless platforms. It contains a number of multifunction applications and focuses on function composition patterns.

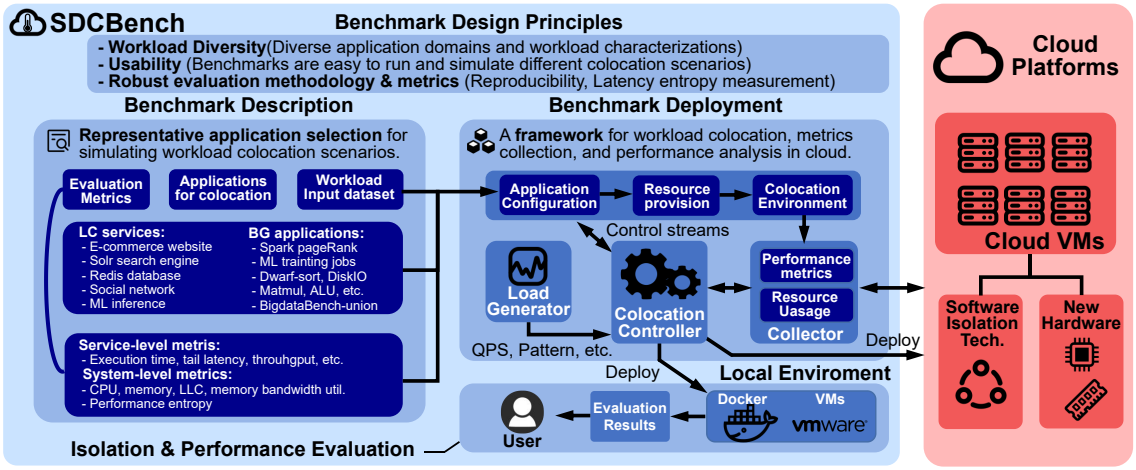


Figure 1: An overview of SDCBench.

2.1.3 Implications

The limitations of current benchmarks motivate us to design a new benchmark suite for widespread colocation scenarios in datacenters to help researchers understand the interference in cloud systems and evaluate their new proposed software and hardware techniques to improve the system efficiency. The benchmark suite should be designed with the following principles: **(i) Workload diversity**: The benchmark should include both latency-critical services and background applications from a wide range of domains in cloud systems. The applications in the benchmark should be sensitive to the pressures on multidimensional resources at the system and microarchitecture levels **(ii) Usability & Robust evaluation methodology**: The workloads in the benchmark suite should be easy to use for simulating different workload colocation scenarios in a local cluster or public cloud systems. The benchmark should also be able to provide automatic evaluation and metric collection mechanisms that cover a wide range of service latency or completion time from microseconds to tens of minutes. **(iii) Characterization & Measurement for interference**: The benchmark should be able to characterize the workload performance degradation caused by shared resources interference in cloud scenarios, and it should also provide metrics for measuring the performance isolation ability in cloud systems.

2.2 Overview of SDCBench

In this section, we present the design of SDCBench. We first give an overview of the evaluation framework of this benchmark to explain how it works in cloud systems. Then, we describe the main components in the benchmark suite, including the application selection methodology, definition of latency entropy and the implementation of the key modules of SDCBench.

Figure 1 shows the overview of SDCBench. SDCBench includes 16 latency-critical services and background applications with representative workload characterizations in multitenant shared cloud scenarios. It supports workload colocation in cloud systems based on these applications and metrics collection from service performance to microarchitectural behaviors. Different from the existing

benchmarks, SDCBench implements a robust evaluation methodology and latency entropy metric that enables it to measure the interference isolation ability between different tenants in a cloud system.

SDCBench can be deployed in a local server cluster running on a container engine or public cloud platforms built on top of VMs. We also develop an automatic software toolkit to help users easily build, deploy and evaluate SDCBench in cloud systems. The toolkit consists of three key components: *Colocation controller*, *Load generator* and *Metrics collector*. For each evaluation case in the cloud scenario, the user can choose the needed benchmarks from SDCBench and configure their resources and runtime parameters, such as query per second, request arrival pattern and peak load. The *colocation controller* reads the user's configurations, prepares the sandbox environments and runs applications by calling *load generator* to send request. During this process, *metric collector* monitors the service-level and system-level states and collects the valuable metrics, which will be analyzed by the *colocation controller* and then present the evaluation results to user.

2.3 Benchmarks Selection

2.3.1 Candidated Applications

To design SDCBench, we first select candidate applications used in cloud scenarios. The existing benchmarks have proposed a large number of applications from simple website to resource-intensive background tasks. However, some of these benchmarks are similar in their workload behaviors and microarchitecture characterization. To select representative applications, we conduct a characterization of these benchmarks using metrics describing CPU, memory behaviors and external resource requirements. The evaluation allows us to classify applications and choose a benchmark set that is representative according to the required resource consumption.

We collect more than 50 applications from existing benchmarks [1, 17, 18, 31, 32, 38]. As shown in Figure 2, these applications come from diverse cloud scenarios such as web search, video processing, machine learning and serverless computing. The performance measurement of these applications also covers a wide range of response times, from microsecond latency in interactive services to tens of minutes of completion time in background tasks. However, many of these applications have similar workload characterization of the pressures on the underlying hardware resources. For example, `Image-classify`, `Scimark` and `Alu` are compute-intensive applications that require much computing capacity for task or data processing. Similar phenomenon can be found in I/O intensive applications such as `Redis`, `Memcached` and `Media-streaming`.

To reduce the number of redundant applications and select a representative set of them in SDCBench, we characterize the applications in these benchmarks by collecting their microarchitecture resource consumption. The measured metrics include CPU, memory, LLC, memory bandwidth, network, disk I/O and IPC. These applications are deployed individually in isolated sandboxes without running other workloads aside from the server. Since latency-critical services always have varied workloads, we measure these services under different loads (10%, 50% and 100%) and aggregate the collected data as their overall metrics. For background applications, we characterize them by collecting the performance metrics under different input data sizes. The input datasets used for

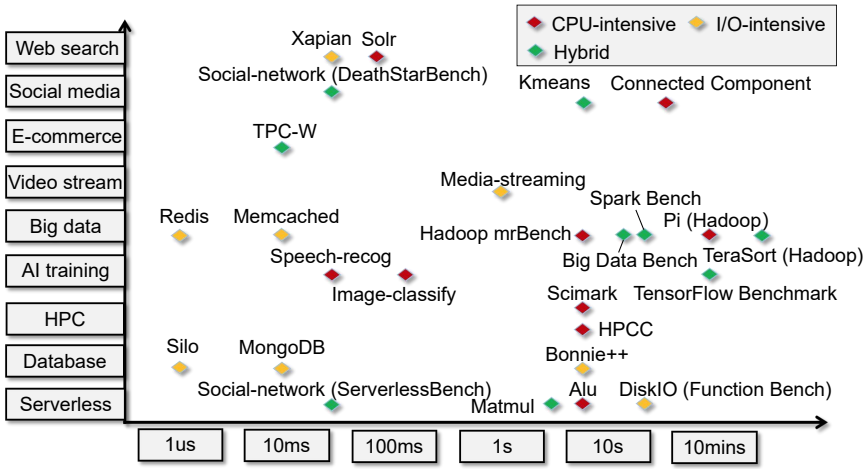


Figure 2: The applications in existing benchmarks.

Table 2: Application dataset

Dataset	Size	Type
Wikipedia index	4,300,000 literatures	Text
Amazon Movie Reviews	7,911,684 movie reviews	Text
Facebook Social Graph	4,039 nodes, 88,234 edges	Text
E-commerce transaction	242,735 rows, 6 columns	Graphs
ImageNet	14,197,122 images	Images

evaluating these applications are listed in Table 2. To estimate the measurement error from system noise, we prewarm every application for a period of time (e.g., 5 minutes) in each load and collect the average metrics from a statistical method.

We use a vector $\boldsymbol{P} = \{p_1, p_2, ..., p_n\}$ to present the profile of the application, where n is the dimension of metrics in the resource consumption characterization. For each resource, we normalize the metric value to the maximum resource capacity in the server, which is mapped in the range of 0 to 1. In this paper, we construct a 7-dimension profiling vector for each application, the profile metadata includes the consumption of CPU cores, main memory, memory bandwidth, LLC, disk I/O, network resources and a micro-architecture metric IPC. After that, we adopt the K-means algorithm [39] to cluster these applications based on their similarity to select the minimum set of representative benchmarks. We use cosine distance to define the similarity of these applications, which is commonly used for classification. Given the profile vectors of two applications, their cosine distance can be calculated as follows,

$$D(P_i, P_j) = \sqrt{\sum_{k=1}^{n} |p_{ik} - p_{jk}|^2} \tag{1}$$

where p_{ik} (p_{jk}) represents the k_{th} metric of application i (j). If the characterization of application i is close to application j, a small value of the cosine distance is derived.

Based on the application characterization, these candidated benchmarks are clustered into 6 classes of latency-critical services and 10 classes of throughput-oriented applications (see appendix A).

As the benchmarks in a same class have similar workload behaviors, we finally select one of them in each class for building the application set of SDCBench. Then selected 16 applications in SDCBench are listed in Table 3. For clarity, each number in the table is color-coded as follows: Red is ≥ 0.8, yellow is between 0.2 and 0.8, and green is ≤ 0.2. We can see that these applications exhibit a variety of resource sensitivity characteristics, and many of them can generate considerable pressure on several dimensions of resources.

Table 3: Application selection in *SDBench*.

Type	Name	Cpu	Mem	Mem I/O	LLC	IPC	Disk I/O	Network
LC	Image-classify	0.956	0.289	1.000	1.000	0.663	0.000	0.108
	Redis	0.239	0.069	0.029	0.127	0.228	1.000	0.076
	Solr	0.430	1.000	0.193	0.165	0.099	0.000	0.053
	Speech-recog	1.000	0.085	0.547	0.971	0.673	0.000	0.368
	TPC-W	0.017	0.054	0.016	0.160	0.307	0.000	0.052
	Social-network	0.288	0.062	0.061	0.143	0.208	0.000	0.007
BE	AlexNet	0.674	0.600	0.918	0.616	0.006	0.000	1.000
	ResNet20	0.126	0.461	0.163	0.173	0.424	0.000	0.017
	DecisionTree	0.082	0.234	0.061	0.115	0.911	0.078	0.027
	Alu	1.000	0.007	0.004	0.573	0.595	0.000	0.000
	PageRank	0.079	0.224	0.007	0.082	0.215	0.087	0.018
	DiskIO	0.008	0.274	1.000	0.916	0.146	1.000	0.000
	LeNet	0.221	0.209	0.279	0.236	0.494	0.000	0.822
	Dwarf-sort	0.008	0.046	0.076	0.294	0.994	0.356	0.000
	Matmul	0.757	0.518	0.959	1.000	0.348	0.000	0.000
	BigDataBench-Union	0.213	0.161	0.106	0.563	0.187	0.000	0.000

2.3.2 Application Descriptions

We now briefly describe the applications included in SDCBench.

Image-classify [40] is a deep learning serving application implemented with python. This service takes images through HTTP requests and active a ResNet model for `Image-classify`. Since ResNet is computationally intensive, the serving latency typically ranges from 10s to 1000s of milliseconds.

Redis [41] is an open-source, key-value database, and is widely used as distributed in-memory cache and message brokers. `Redis` is written in C and is highly efficient, which provides sub-millisecond response latency.

Solr [42] is an open-source enterprise-search engine written in Java, which supports full-text search, hit highlighting and real-time indexing. `Solr` is highly scalable and fault-tolerant and is widely used for enterprise search and analytics use case. The search latency of `Solr` is typically at 10s of milliseconds.

Speech-recog [43] is a speech classification inference service, which consists a `Speech-recog` model that takes in the frequency spectrum of the input speech sequence and produces the classified labels. This light-weighted model is implemented with python, which takes only 10s of milliseconds for inference.

TPC-W [44] is a web server and database performance benchmark, which is proposed by the Transaction Processing Performance Council. It defines the complete web-based shop for searching, browsing, and ordering books. The response time of such web interactions typically ranges from 10s to 100s milliseconds.

Social-network [30] is a microservice application reside in the DeathStarBench benchmark, which is an end-to-end service that implements a broadcast-style social network with unidirectional follow relationships. Since requests may be forwarded and processed by different components, the service latency typically ranges from 10s to 100s milliseconds.

DecisionTree [45] is an application in the Spark benchmark suite, which is written in Scala. The spark decision tree application is implemented with Spark mllib APIs, which supports decision trees for binary and multiclass classification and for regression. This application is highly IPC efficient.

Alu [32] is an arithmetic computation application in the serverless benchmark suite, ServerlessBench, which computes the arithmetic operation repeatedly with multiple threads. The `Alu` application is CPU intensive and requires much less memory and network resources.

PageRank [46] is a graph processing application implemented with Spark [47]. Since web pages could be numerous, the page rank computation also consumes relatively intensive resources and jobs would take at least minutes to complete.

DiskIO [48] is an application from serverless benchmark suite, FunctionBench, which performs the *dd* system command that creates a file in the */tmp/* directory of the function runtime. The `DiskIO` application consumes less CPU, memory and network resources, while imposing high pressure on the disk I/O bandwidth.

Dwarf-sort [18] is a big data sort application in the BigDataBench benchmark suite, which is implemented with Scala by sorting the Wikipedia entries by keys. This application is typically memory and cache intensive.

AlexNet, **LeNet** and **ResNet20** [49] are deep learning training applications implemented with TensorFlow. These deep learning training processes impose huge and lasting pressure on CPUs, memory, cache, and network bandwidth. Typically, `AlexNet` has the highest resource demands, while `LeNet` consumes relatively fewer resources to train the network.

MatMul [50] is a matrix multiplication application in the HPCC benchmark. Typically, the matrix multiplication operation consumes a large amount of CPU, memory, and cache resources, while generating less network pressure.

2.4 Metrics Collection

SDCBench supports the measurement of both service-level and system-level metrics in workload colocation scenarios. These service-level metrics mainly focus on application performance and are presented to the user as intuitive results. System-level metrics are collected to analyze user application runtime behavior and the impact of system isolation ability on application performance. We now discuss the detailed metrics that are measured at the service level and system level.

Service-level metrics. These metrics provide an accurate profile of application performance for users.

• **Response Time** For interactive services, SDCBench records the response time for each request

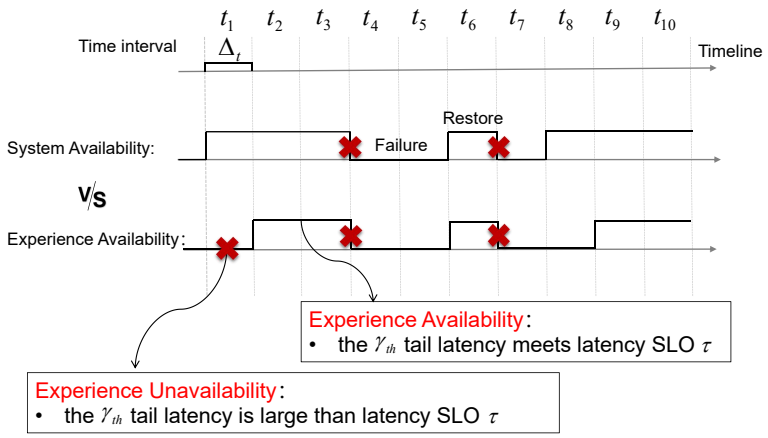


Figure 3: The definition of experience availability.

to analyze their performance changes in cloud systems. The collected metrics include the latency for a single request, the average latency and the tail latency (e.g., $90th$, $95th$ and $99th$ latency).

• **CPU Utilization**. SDCBench supports the measurement of the CPU time in the user and the kernel space spent by the application. This metric helps to determine which applications are sensitive to computational resources.

• **Memory**. The available memory size directly affects the data swapping speed for applications that need to exchange a large amount of data between the memory and disk, and insufficient memory can cause an increase in the order of magnitude latency. SDCBench supports the measurement of peak memory usage to help users understand the memory requirements of their applications.

• **Disk & Network I/O** The contention on the disk and network affects the performance of I/O-intensive applications. Previous work has shown that the average throughput of file system I/O and network operations decreases with the number of colocated applications in cloud systems that share bandwidth. SDCBench supports the measurement on the disk I/O and network usages to help users track performance variations from the contention on these two-dimensional resources.

• **Cache & Memory Bandwidth**. Latency-critical services, such as in-memory databases are more sensitive to contentions on cache and memory bandwidth. Even when adopting hardware isolation mechanisms such as CPU core binding and the *numactl* technique, contention from the shared LLC and memory bandwidth is inevitable between virtual machines in old generation servers. SDCBench provides the measurement of LLC and memory bandwidth usage to analyze the applications' performance degradation caused by microarchitecture resource contentions.

• **Cache Miss**. An insufficient cache capacity or contentions from the threads running on the same CPU socket may cause frequent cache misses for the application. SDCBench uses hardware performance counters to count cache miss events and collect LLC MPKI as a metric.

System-level metrics. This metric is used to describe the performance uncertainty of an application running on a cloud system and the impact of performance degradation on users caused by system uncertainty.

• **Experience Availability**. SDCBench supports the user-oriented service experience availability (EA) measurement (Figure 3). Prior work always focuses on system failures that cause user expe-

rience degradation, that is, when the system fails, the service becomes unavailable. However, this ignores the low latency requirement of cloud users. High tail latency also degrades user experience and even violates the users' latency service level objectives (SLOs). SDCBench introduces a new EA metric [51, 52] by combining the system availability with service tail latency, which is defined as follows,

$$EA(\tau,\gamma) = \sum_{i=1}^{m} t_i(\tau,\gamma)/m \tag{2}$$

$$t_i(\tau,\gamma) = \begin{cases} 0, & tail\_latency(\gamma_{th}) > \tau \\ 1, & tail\_latency(\gamma_{th}) \leq \tau \end{cases} \tag{3}$$

where the collected latency statistics are divided into m uniform time intervals (i.e., $\triangle_t$). For the i_{th} time interval, if the γ_{th} of tail latency $t_i(\tau,\gamma)$ meets $t_i(\tau,\gamma) \leq \tau$, where τ is the latency SLO, then it is set to 1. Otherwise, $t_i(\tau,\gamma)$ is set to 0.

• **Latency Entropy**. The concept of entropy was first introduced by the German physicist, Clausius, in 1865 and is used to describe the degree of disorder within a system [53]. Inspired by this, SDCBench first proposes the latency entropy (LE) metric for measuring the uncertainty of cloud systems. Inside a computing system, the sequence of system calls and hardware accessing events (e.g., memory access, instruction fetching and thread execution) occur in per unit time can be considered as micro-state of the system. The colocation from multiple applications makes the micro states of the system more complex, especially when shared resource contention occurs between different applications. In these scenarios, system behaviors, such as instruction fetching and execution become disorderly and unpredictable.

Unfortunately, it is difficult to measure the internal micro states of computer systems under the architecture of modern high-speed processors. To help users understand their application performance changes with observable metrics, SDCBench defines the latency entropy that describes the variations of tail latency for the measurement of system isolation ability. The latency entropy is calculated as follows,

$$LE = -\sum_{i=1}^{n} p_i \log_{p_i} \tag{4}$$

where n is the number of latency distribution states, and p_i represents the i_{th} state's probability. In practice, we divide the collected latencies into multiple fixed-length time intervals, and each of them can be seen as individual states, then the probability of one state can be approximately derived by calculating the percentage of latency samples falling into the corresponding interval. For each cloud service, latency entropy can be used to describe its performance uncertainty in cloud system, which implies that: (i) The smaller the number of latency distribution states, the smaller the latency entropy of the cloud system. (ii) The more uneven the probability of the latency distribution states, the smaller the latency entropy of the cloud system. For example, if service A has a latency state distribution with "[14,17,20],[24,25,29],[32,37]", and service B has a latency state distribution with "[19],[23,24,24,26,28,29],[31]", we could see that service B is more stable than service A, and actually, service B has smaller LE score than service A (0.81 v.s. 1.08), which is consistent with our observation.

3 Results and Discussion

3.1 System Implementation

We implement an evaluation framework based on SDCBench, which is designed to help users easily understand the isolation ability of different cloud provides, thus to evaluate their application performance in these cloud platforms. As mentioned in § 2.2, the colocation controller, load generator and metrics collector are the core modules in the framework design, which support automatic application configuration, deployment, and measurement of performance and cloud system isolation ability.

Colocation controller. The controller automatically manages all necessary steps of the workload evaluation in a cloud system. It provides application selection and parameter configuration interfaces with a visual interface for users to execute these operations. The latency-critical services and background applications in SDCBench are registered in the database, and the user can select the evaluated applications by marking their flags as executable. For latency-critical services, SDCBench supports the evaluation parameter settings, such as request arrival pattern, peak load, warm-up invocations, and the total request invocations. For background applications, SDCBench supports the settings of job execution times, task types, and input data sizes.

SDCBench supports component-level (i.e., container-level) application colocation based on docker APIs and Linux system tools. It uses *docker update* commands [54] and the *numactl* [55] tool to bind the CPU core and memory block to containers. Applications running on the same CPU socket may have contentions on the shared cache and memory bandwidth, which causes performance inference for the colocated workloads. SDCBench also provides a fine-grained resource partition mechanism for containers running within the same CPU socket, it adopts the *Intel RDT* tool [13] to set the cache ways and memory bandwidth for each container. Additionally, SDCBench uses the *qdisc* network tool [56] for allocating network bandwidth for the evaluated applications to measure their performance variations in both colocated and isolated workload scenarios.

Load generator. SDCBench implements a load generator to generate requests to the latency-critical services, which can be deployed on one or more client machines. The load generator integrates a traffic shaper, a client pool and a recorder. It creates multiple clients from the thread pool to continuously generate requests, and the traffic shaper handles these requests and sends them to the backend service following the desired workload patterns (e.g., from production traces) by inserting delays between requests before sending them out over the network. The simulated clients operate in an "open-loop" mode, where the request can be sent directly according to their desired timing characteristics without waiting for responses from previous requests. The open-loop [17, 57] setups generate sufficient workload pressures on the evaluated services and can accurately capture the queuing delays that are an important factor impacting the tail latency.

The recorder maintains a queue to store the processed requests, which is shared among simulated clients. It records the response time of all the requests sent by the clients, aggregates them and calculates statistical metrics, such as the single response time, average latency and tail latency. The measured data can be stored in a database or exported as files for users.

Metrics collector. The performance degradation caused by interference can manifest in multiple hardware resource activities. To accurately measure these system layer and microarchitecture

Table 4: Experimental testbed configuration.

Component	Specification	Component	Specification
CPU device	Intel Xeon Silver-4215	Shared LLC Size	11MB
Number of sockets	2	Memory Capacity	128GB
Processor BaseFreq.	2.50 GHz	Operating System	Ubuntu 16.04
CPU Threads	32 (16 physical cores)	SSD Capacity	960GB
Memory Bandwidth	20 GB/s	Network bandwidth	10 Gbps

level behaviors without introducing external overhead, SDCBench adopts a nonintrusive method to implement the metric collector. In the system layer, we collect the actual resource usage ratio of the measured application, which includes the number of CPU cores, memory, network, disk I/O, etc. In the microarchitecture layer, we collect branch prediction errors, cache switching, context switching, memory-level parallelism and misses per thousand instructions (MPKI). These metrics focus on the operating efficiency of the application code on the current physical hardware, such as locality and parallelism, which help us understand where the inference comes from and its impact on the application performance.

The collector runs aside the measured applications in an isolated CPU socket and adopts a multithreading technique to invoke a series of monitoring tools such as *Intel RDT*, *Perf* [58] and *Docker Stats* [54], for the metrics measurement. When all of the monitors complete, the collector formats the data and returns the results to the user. SDCBench is open source and is available at https://github.com/TankLabTJU/sdcbench/tree/sdcbench-v2.0/.

3.2 Evaluation and Methodology

SDCBench is designed to help cloud users understand the performance isolation ability of cloud systems by deploying colocated applications in cloud systems that different workloads may share the hardware resources and measuring their performance changes. In the evaluation of SDCBench, we need to answer several key questions: Are the benchmarks in SDCBench representative for the multi-tenant cloud scenarios by covering a wide range of latencies that can be measured? Is SDCBench able to observe the service performance degradation due to interference from colocated workloads? Can hardware isolation mechanisms eliminate the performance variations caused by interference? How do the major cloud service providers perform in latency entropy measurement? To answer these questions, we use SDCBench to thoroughly evaluate cloud system under different workloads. We begin with a local benchmark evaluation to verify that we cover various workload behaviors (§ 3.2.1). We analyze performance degradations of latency-critical services under different workload colocation scenarios (§ 3.2.2), and present the comparison of latency entropy in some of the existing cloud platforms (§ 3.2.3).

We evaluate SDCBench on both a local cluster and public cloud platforms. The benchmarks are implemented in C, Python 3.7 and Java. All real-system measurements reported in the evaluation were performed on servers with two Intel Xeon Silver-4215 CPUs. Table 4 shows the detailed server configurations. We run load generator on individual server to avoid interference from the deployed applications. In the testbed, we bind the applications in the second CPU socket and forbid the operation system to schedule other tasks on the CPU cores in this socket, thus to prevent

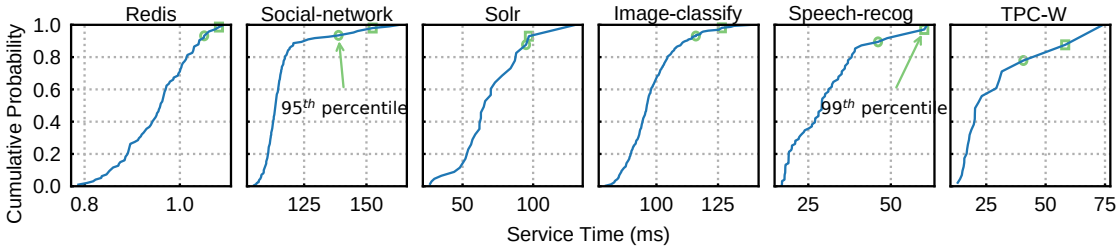


Figure 4: Cumulative distribution function (CDF) of latency for each interactive services, with latency on the x-axis and cumulative probability on the y-axis.

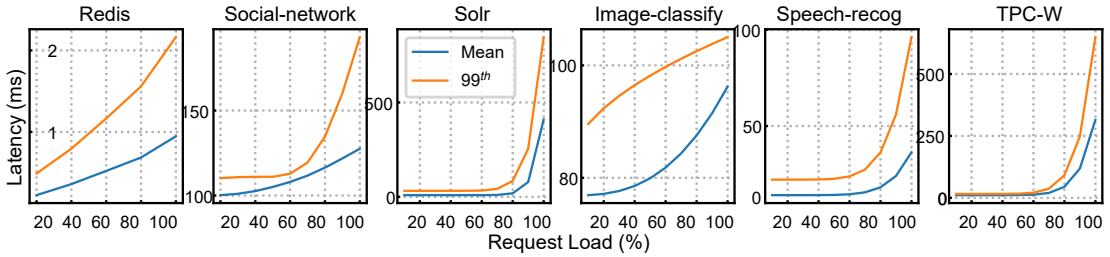


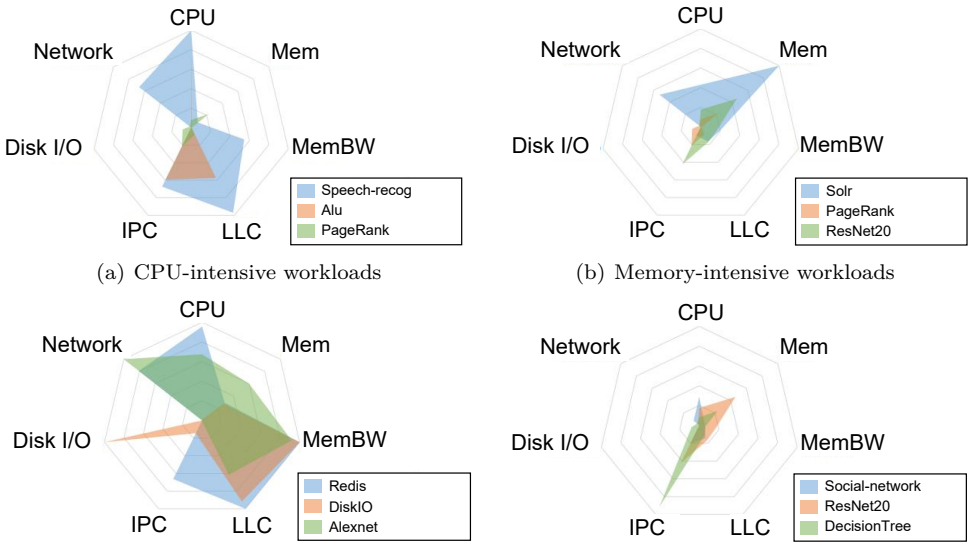
Figure 5: Mean and 99th percentile latencies for each application under different request loads.

interference from the system. We also disable the TurboBoost technique and use *cpufreq* tool to fix CPU frequency at 2.0 GHz, which can help to avoid unpredictable performance fluctuations [59]. The server and client nodes are connected via 10 Gbps, full-bisection bandwidth Ethernet.

3.2.1 Benchmark Characteristics

We now study the latency characteristics of each application, which include the average request service time and tail latency. The service time of a request measures the time the application takes to process that request, which can reflect the execution speed of the application code running on dedicated hardware. The tail latency represents the few slowest requests (e.g., the slowest 1% requests when measuring the 99th percentile latency), it is much more sensitive to small perturbations and can be used to observe the service performance fluctuations. We also study how the request arrival rate affects tail latency in these applications. All measurements in this experiment were obtained by the record collector module in the load generator. To mitigate the measurement error caused by system noise, we collect the evaluation metadata after the application running stably and each of these experiments is measured three times.

Q1: Are the benchmarks in SDCBench representative for the multi-tenant cloud scenarios by covering a wide range of latencies that can be measured? Figure 4 shows the cumulative distribution function (CDF) of request service times for each SDCBench application. Obviously, the service times vary widely across applications. Almost all `Redis` requests finish in less than 1.1 ms, and the difference between the lowest and highest service times is only 0.3 ms. But the service time of `Social-network` requests can take more than 110 ms each. Applications also vary widely in how tightly their request service times are distributed. For some applications, request service times are distributed in a fairly narrow range or have a long tail. 90% of `Social-network`



(a) CPU-intensive workloads

(b) Memory-intensive workloads

(c) Hybrid-contention workloads

(d) Symbiotic workloads

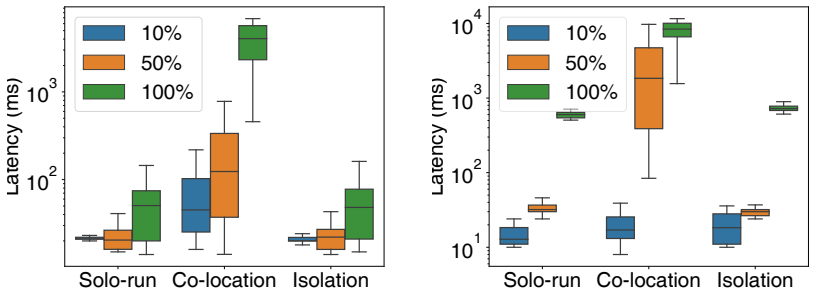
Figure 6: Four representative application suites for workload colocation and system measurement.

request times are distributed between 110 ms and 125 ms, and another 10% are distributed between 125 ms and 175 ms, accounting for 77% of the total time distribution. `Solr` requests have 1% of requests spread over 100 ms to 150 ms, accounting for one-third of the total time distribution. `Image-classify` requests have a similar trend. Other applications, such as `Speech-recog` and `TPC-W`, have their request service times fairly evenly distributed in two specific ranges.

Figure 5 shows the mean and 99th percentile latencies for each application at various request load. In these experiments, 100% of the request load represents the queries per second (QPS) limit of the application under the current configuration. At very low request load, the difference between mean and tail latencies mostly depends on the distribution of request service times. As the request load increases, both mean and tail latencies increase because of competition for resources and queue delays. However, the tail latencies of all applications except `Image-classify` increase much faster than the mean; for example, `Solr` requests have a tail latency of about 500 ms higher than the mean latency at 100% of the request load and 50 ms higher at 80% of the request load. `Redis` also has a similar trend, but the difference between the tail latency and the mean latency of `Redis` increases slowly compared to other applications. And the tail latency of `Redis` is only about 1 ms higher than the mean at 100% of the request load. The difference between the tail latency and the mean latency of `Image-classify` is also small, but its tail latency growth rate is gradually exceeded by the mean latency growth rate.

3.2.2 Interference Measurement

We next analyze the performance changes of these benchmarks in multi-tenant shared cloud scenarios. Users may deploy different workloads in their cloud virtual machines that run on a large scale of physical servers. Recent studies have shown that interactive services such as websites make up a large part of these cloud applications. At the same time, many users also use cloud VMs to



(a) CPU-intensive workloads (b) Memory-intensive workloads

(c) Hybrid-contention workloads (d) Symbiotic workloads

Figure 7: The latency distributions of latency-critical services in the four workload suites under 10%, 50% and 100% request loads. Each workload suite are evaluated in solo-run, colocation and isolation groups, respectively.

process data-intensive applications such as batch tasks. When deploying SDCBench to simulate these shared cloud scenarios, we prefer to observe the performance degradations on the colocated workloads, which means that applications running inside the cloud system cause contentions on the underlying hardware resources. This could help us to understand the sensitivity of the SDCBench applications for interference and the range of performance changes they can measure.

Based on the benchmark characterization of SDCBench in § 2.3.1, we build four workload colocation suites with different levels of competition in share resources (Figure 6). These application combinations are (i) CPU-intensive suite: This suite includes `Speech-recog`, `Alu` and `PageRank`, which are computation-intensive applications during their executions. (ii) Memory-intensive suite: This suite consists of latency-critical service `Solr`, background applications `PageRank` and DNN model training for `ResNet20`, which rely on much memory resources for execution. (iii) Hybrid contentions: This suite contains `Redis`, `DiskIO` and DNN model training for `AlexNet`, which can generate pressures on multi-dimensional resources. (vi) Symbiotic workloads: This suite includes `Social-network`, `DecisionTree` and DNN training on `ResNet20`. Unlike the other combinations, these three applications can be colocated together without significant performance interference on shared resources. We use these benchmark suites on the local cluster and evaluate their performance changes with the measurement of system isolation ability.

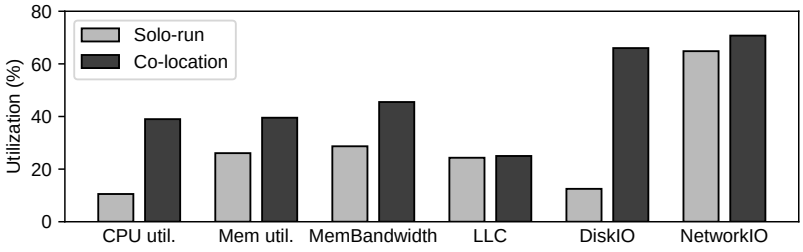
Q2: Is SDCBench able to observe the service performance degradation due to interference from colocated workloads? Figure 7 shows the latency distributions of latency-critical services in these colocation workload suites. For these workload suites of CPU, memory, and hybrid

contentions, the latency of colocation is significantly higher than that of solo-run. And it is clear that the latency distribution of above three types of workloads at various request load has become larger, which means that the performance of these services is more unstable. `Solr` is most sensitive to the interference caused by colocation with other background workloads. At 50% of the request load, the mean latency of colocated `Solr` service is about 70× that of the solo-run, and the latency distribution is also significantly larger. At 100% of the request load, the latency of `Solr` increases from about 600 ms to 9000 ms. Compared with the other two types of workload, `Redis` is less sensitive to the colocation, but the latency is still increased by 8 times at 10% of the request load.

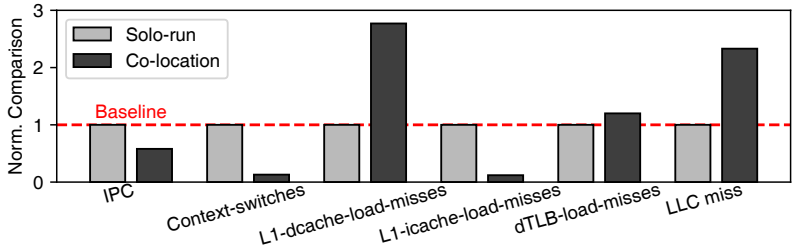
The performance degradation caused by interference is especially obvious when the online service is under high load. This is because as the load of online services increases, the demand for memory and CPU resources increases, and the competition with other background applications becomes more intense, making it more sensitive to interference. For example, while at 10% and 50% of the request load, the latency of colocated `Speech-recog` increases by 2× and 10× compared with solo-run, respectively. At 100% of the request load, the latency increases by about 100×. Compared with the previous three workloads, the distribution of latency increases slightly, but is not obvious. The mean latency of colocated social network increases about 5 ms compared with solo-run at 10% of the request load. However, the mean latency hardly changes at 100% of the request load and even slightly decreases at 50% of the request load. Therefore, when the symbiotic application is combined with other background applications, the performance does not change much compared to the solo-run, which is also in line with our expected results. This result shows that the performance degradation in the first three workloads is indeed caused by interference between applications, and it also shows that the application selection in SDCBench is scientific.

Figure 8(a) and 8(b) show an example of the comparison in system-level and microarchitecture-level metrics with hybrid-contention workloads suite. As `Redis` generates a high utilization of network and LLC resources, there is little difference in these two metrics between solo-run and colocation workloads suites. By colocating `Redis` with `DiskIO` and `AlexNet`, the CPU and memory utilizations in the system are improved by 2.7× and 13%, respectively. Meanwhile, the usage of `DiskIO` in colocation group increases by 4.28×. However, we could also see that the IPC and Context-switches metrics decrease by 42% and 87% in the colocation group since the workload interference significantly reduces the QPS of `Redis`. Additionally, the workload interference in colocation group results in higher tail latency for `Redis`. When compared with solo-run group, the cache miss rate of L1 and L2 increase by 1.7× and 1.3× in colocation group, respectively. More cache miss also results in a higher memory access rate, and we could see the memory bandwidth usage increases about 30% in the colocation group. This explains why the shared resources contention leads to performance degradation for latency-critical services.

Q3: Can hardware isolation mechanisms eliminate the performance variations caused by interference? In the evaluation of the first three workloads, application performance has been greatly improved after resource isolation. It can be seen from the figure that the performance of these three services in isolation is very close to that of solo-run. In CPU-intensive workloads, after isolation, the mean latency of `Speech-recog` at three types of request loads decreases by about 20 ms, 90 ms, and 4000 ms, respectively, which is almost the same as that of



(a) Comparison of system-level metrics



(b) Comparison of microarchitecture-level metrics

Figure 8: Comparison of the system-level (a) and microarchitecture-level (b) metrics in solo-run and colocation groups. The metrics are collected with the Memory-intensive workload suite.

Table 5: Comparison of experience availability (EA) in local cluster.

Types	**Solo-run**			**Co-location**			**Isolation**		
Loads	10%	50%	100%	10%	50%	100%	10%	50%	100%
Speech-recog	1.0	1.0	1.0	0.42	0.29	0.16	0.97	0.98	0.99
Solr	1.0	1.0	1.0	0.81	0.1	0.04	0.82	1.0	0.98
Redis	1.0	1.0	1.0	0.19	0.04	0.02	1.0	0.95	0.98
Social-network	1.0	1.0	1.0	0.89	0.83	0.57	0.95	0.98	0.97

solo-run. And the performance stability is very close to that of solo-run. It indicates that the workload colocation with resource isolation brings limited interference. For symbiotic application combinations, there is no significant change between the effect of resource isolation and colocation. We can see that the latency distribution after resource isolation is more centralized, which means that performance is more stable. However, under 100% request load, the mean latency of social network even increases by about 10 ms. This also shows that the resource isolation method can improve application performance to a certain extent in the multi-tenant cloud scenario, but it should be analyzed according to the specific application characteristics.

Table 6: Comparison of latency entropy (LE) in local cluster.

Types	**Solo-run**			**Co-location**			**Isolation**		
Loads	10%	50%	100%	10%	50%	100%	10%	50%	100%
Speech-recog	0.03	0.06	0.45	1.04	2.15	4.06	0.04	0.10	0.61
Solr	0.05	0.06	1.60	0.07	3.32	3.61	0.06	0.07	1.34
Redis	0.18	1.56	1.89	3.41	3.53	3.51	0.12	1.43	2.19
Social-network	1.75	3.01	2.77	2.53	2.84	2.90	2.10	2.69	2.66

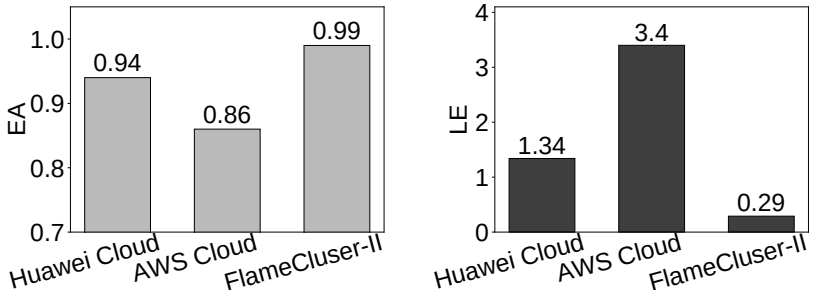
We also measure the service experience availability and latency entropy metrics of these applications in each group of experiment. Table 5 shows the comparison of service experience availability of the colocated latency-critical services, which is calculated with Equation 2 and 3. We define the latency-critical services have the best performance in solo-run group, and set the latency SLO (τ) as the value that meets $EA = 1.0$ at service 100% load. We could see that the service experience availability decreases dramatically in the first three experiment groups because of the performance interference between colocated workloads. For example, the experience availability of `Speech-recog` reduces about 60% at 10% load level in the colocation group. Moreover, its performance becomes even worse when we increase the request arrival rate, which achieves only 29% and 16% of time intervals that meet latency SLO at 50% and 100% load levels, respectively. Additionally, we could see that isolation on hardware resources significantly improve the performance availability of these latency-critical services. For `Speech-recog`, its performance availability recovers to 0.97, 0.98 and 0.99 at 10%, 50% and 100% load levels, respectively. The performance availability changes of `Solr` and `Redis` are similar to the `Speech-recog`. `Social-network` has less performance fluctuations over these evaluated applications.

We further present the comparison of latency entropy of these experiments, which are listed in Table 6. For each workload colocation suite, we record the best and worst latencies in the solo-run group, divide them into multiple latency intervals and calculate the probabilities of latency time that falls in these intervals in colocation and isolation groups, thus to derive their latency entropy measurements. We could see that `Social-network` has the highest latency entropy in solo-run group over the four workload colocation suites as it's latency that crosses multiple microservices invocations is more sensitive to the performance fluctuation. The interference caused by shared resources contention leads to an average of 13×, 4× and 2.8× of LE increasement except for the `Social-network`. By isolating the shared resources between these colocated applications, the average LE decreases about 9.7×, 4.7× and 2.8× in `Speech-recog`, `Solr` and `Redis` groups, respectively. This indicates that the interference between colocated workloads can greatly magnify the system uncertainty, leading to unpredictable performance degradation on applications running in the system. Introducing isolation mechanisms such as hardware partitions can effectively reduce the application performance fluctuation caused by system uncertainty, thus improve the user experience.

3.2.3 Case Study in Public Cloud

Q4: How do the major cloud service providers perform in the latency entropy measurement? One of the key benefits of SDCBench is to help users to understand their application performance in different public cloud systems. We seek to find the answers in some of the existing public cloud platforms through a simple case study, where we deploy SDCBench in these platforms to measure their latency entropy metrics. Specifically, our testbeds include the public cloud platforms such as Huawei Cloud, AWS Cloud and a local prototype system FlameCluser-II [60], which is built based on the new CPU architecture for Labeled von Neumann Architecture (LvNA) that supports better isolation of shared resources than the traditional x86-based CPU architectures.

Since the current version of FlameCluster-II only supports C and JAVA languages, we choose



(a) Comparsion of experience availability (b) Comparsion of latency entropy

Figure 9: The comparison of experience availability (a) and latency entropy (b) in Huawei Cloud, AWS Cloud and FlameCluster-II.

`TPC-W` and `Redis` as the evaluated latency-critical services to measure the EA and LE metrics of the three platforms. For the public cloud platforms, we deploy `TCP-W` and `Redis` in individual cloud VMs and collect their latencies at different times of the day. As the VMs of different cloud tenants may be scheduled into the same server, thus the user's application performance can be impacted by the interference from other tenants' workloads. For Flame-Cluser II, we build an 8-node FPGA cluster and deploy benchmarks across these nodes. The load generator in these experiments is deployed in an isolated server environment and communicates with the evaluated applications via network. To reduce the measurement error caused by system noise, we collect the measured data in each experiment three times, and take their statistical metrics as the evaluated results.

We collect more than 1,000,000 request latencies in each testbed and measure their EA and LE metrics. Figure 9 shows the comparison of these three platforms in average EA and LE measurement. We could see that Huawei cloud, AWS cloud and FlameCluster-II achieved 0.94, 0.86 and 0.99 of EA, respectively. This indicates that user may obtain better performance experience by deploying applications in Huawei cloud when compared with AWS cloud. For the comparison of LE, the latency entropy of the three platforms are 1.34, 3.4 and 0.29, respectively. Specifically, applications in FlameCluster-II have minimal performance fluctuations since its strong isolation in hardware from the LvNA design, which validates that hardware isolation is a good way to eliminate performance uncertainty in cloud datacenters. Additionally, the evaluation results also show that application in Huawei cloud achieves better performance isolation ability than that in AWS cloud. This may be because AWS has adopted more aggressive resource oversold policies in different cloud tenants.

4 Conclusion

We have presented SDCBench, a benchmark suite and evaluation methodology for latency entropy measurement in datacenters. SDCBench seeks to help cloud tenants and providers understand the application isolation ability in datacenter by colocating workloads and observing their performance variation in cloud systems. SDCBench includes 16 representative applications selected from today's well-known benchmarks that across a wide range of cloud scenarios. It first proposes the concept of latency entropy and implements a robust methodology to measure the performance isolation ability in datacenters. Our validation results show that SDCBench can simulate different multitenant

shared cloud systems with simple configurations, and we also present the comparison of latency entropy in today's major cloud providers by deploying SDCBench in Huawei cloud, AWS cloud and a local prototype system FlameCluster-II. The evaluation results show FlameCluster-II achieves the lowest latency entropy with 0.29 while the scores in Huawei cloud and AWS cloud are 1.34 and 3.4, respectively.

Acknowledgments

We thank our editor and anonymous reviewers for their extremely insightful comments and suggestions that have significantly improved the quality of this paper. This work is supported by the National Key Research and Development Program of China No. 2016YFB1000205; the National Natural Science Foundation of China under grant 61872265, 62141218; CCF-Huawei Populus euphratica Innovation Research Funding CCF2021-admin-270-202104.

References

[1] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds., ACM, 2012, pp. 37–48. DOI: 10.1145/2150976.2150982. [Online]. Available: https://doi.org/10.1145/2150976.2150982.

[2] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, 2017, pp. 153–167. DOI: 10.1145/3132747.3132772. [Online]. Available: https://doi.org/10.1145/3132747.3132772.

[3] J. Zhang, X. Wang, H. Huang, and S. Chen, "Clustering based virtual machines placement in distributed cloud computing," *Future Gener. Comput. Syst.*, vol. 66, pp. 1–10, 2017. DOI: 10.1016/j.future.2016.06.018. [Online]. Available: https://doi.org/10.1016/j.future.2016.06.018.

[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Improving resource efficiency at scale with heracles," *ACM Trans. Comput. Syst.*, vol. 34, no. 2, 6:1–6:33, 2016. DOI: 10.1145/2882783. [Online]. Available: https://doi.org/10.1145/2882783.

[5] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, J. S. Chase and A. E. Abbadi, Eds., ACM, 2011, p. 22. DOI: 10.1145/2038916.2038938. [Online]. Available: https://doi.org/10.1145/2038916.2038938.

[6] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, T. Conte and Y. Zhou, Eds., ACM, 2016, pp. 473–488. DOI: 10.1145/2872362.2872365. [Online]. Available: https://doi.org/10.1145/2872362.2872365.

[7] H. Yang, A. D. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, A. Mendelson, Ed., ACM, 2013, pp. 607–618. DOI: 10.1145/2485922.2485974. [Online]. Available: https://doi.org/10.1145/2485922.2485974.

[8] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013. DOI: 10.1145/2408776.2408794. [Online]. Available: https://doi.org/10.1145/2408776.2408794.

[9] Z. Xu and C. Li, "Low-entropy cloud computing systems," *SCIENTIA SINICA Informationis*, vol. 47, no. 9, pp. 1149–1163, 2017. DOI: https://doi.org/10.1360/N112017-00069.

[10] M. Tirmazi *et al.*, "Borg: The next generation," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, 2020, 30:1–30:14. DOI: 10.1145/3342195.3387517. [Online]. Available: https://doi.org/10.1145/3342195.3387517.

[11] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, ACM, 2018, pp. 347–360. DOI: 10.1145/3267809.3267830. [Online]. Available: https://doi.org/10.1145/3267809.3267830.

[12] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007. DOI: 10.1109/MC.2007.443. [Online]. Available: https://doi.org/10.1109/MC.2007.443.

[13] I. A. Papadakis, K. Nikas, V. Karakostas, G. I. Goumas, and N. Koziris, "Improving qos and utilisation in modern multi-core servers with dynamic cache partitioning," in *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017, COSH/VisorHPC@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*, C. Clauss, S. Lankes, C. Trinitis, and J. Weidendorfer, Eds., TUM Library, 2017, pp. 21–26. DOI: 10.14459/2017md1344298. [Online]. Available: https://doi.org/10.14459/2017md1344298.

[14] J. Ma *et al.*, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD)," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds., ACM, 2015, pp. 131–143. DOI: 10.1145/2694344.2694382. [Online]. Available: https://doi.org/10.1145/2694344.2694382.

[15] C. Iorgulescu *et al.*, "Perfiso: Performance isolation for commercial latency-sensitive services," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds., USENIX Association, 2018, pp. 519–532. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/iorgulescu.

[16] S. Baset, M. Silva, and N. Wakou, "SPEC cloud™ iaas 2016 benchmark," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, Eds., ACM, 2017, p. 423. DOI: 10.1145/3030207.3053675. [Online]. Available: https://doi.org/10.1145/3030207.3053675.

[17] H. Kasture and D. Sánchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*, IEEE Computer Society, 2016, pp. 3–12. DOI: 10.1109/IISWC.2016.7581261. [Online]. Available: https://doi.org/10.1109/IISWC.2016.7581261.

[18] W. Gao *et al.*, "Bigdatabench: A dwarf-based big data and AI benchmark suite," *CoRR*, vol. abs/1802.08254, 2018. arXiv: 1802.08254. [Online]. Available: http://arxiv.org/abs/1802.08254.

[19] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, IEEE Computer Society, 2014, pp. 238–247. DOI: 10.1109/ICDCS.2014.32. [Online]. Available: https://doi.org/10.1109/ICDCS.2014.32.

[20] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, IEEE, 2014, pp. 1842–1850. DOI: 10.1109/INFOCOM.2014.6848123. [Online]. Available: https://doi.org/10.1109/INFOCOM.2014.6848123.

[21] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Gener. Comput. Syst.*, vol. 28, no. 2, pp. 358–367, 2012. DOI: 10.1016/j.future.2011.07.003. [Online]. Available: https://doi.org/10.1016/j.future.2011.07.003.

[22] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: lightweight data skew mitigation in mapreduce," *IEEE Trans. Parallel Distributed Syst.*, vol. 26, no. 9, pp. 2520–2533, 2015. DOI: 10.1109/TPDS.2014.2350972. [Online]. Available: https://doi.org/10.1109/TPDS.2014.2350972.

[23] J. J. Dongarra and P. Luszczek, "LINPACK benchmark," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed., Springer, 2011, pp. 1033–1036. DOI: 10.1007/978-0-387-09766-4_155. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_155.

[24] C. D. Spradling, "SPEC CPU2006 benchmark tools," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007. DOI: 10.1145/1241601.1241625. [Online]. Available: https://doi.org/10.1145/1241601.1241625.

[25] P. Luszczek *et al.*, "S12 - the HPC challenge (HPCC) benchmark suite," in *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, ACM Press, 2006, p. 213. DOI: 10.1145/1188455.1188677. [Online]. Available: https://doi.org/10.1145/1188455.1188677.

[26] D. A. Padua, "PARSEC benchmarks," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed., Springer, 2011, p. 1464. DOI: 10.1007/978-0-387-09766-4_441. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_441.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds., ACM, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152. [Online]. Available: https://doi.org/10.1145/1807128.1807152.

[28] G. Cloud, "Perfkit," 2017. [Online]. Available: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[29] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*, IEEE Computer Society, 2018, pp. 1–12. DOI: 10.1109/IISWC.2018.8573515. [Online]. Available: https://doi.org/10.1109/IISWC.2018.8573515.

[30] Y. Gan *et al.*, "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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 3–18. DOI: 10.1145/3297858.3304013. [Online]. Available: https://doi.org/10.1145/3297858.3304013.

[31] P. Mattson *et al.*, "Mlperf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020. DOI: 10.1109/MM.2020.2974843. [Online]. Available: https://doi.org/10.1109/MM.2020.2974843.

[32] T. Yu *et al.*, "Characterizing serverless platforms with serverlessbench," in *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, 2020, pp. 30–44. DOI: 10.1145/3419111.3421280. [Online]. Available: https://doi.org/10.1145/3419111.3421280.

[33] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition* (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, 2018. DOI: 10.2200/S00874ED3V01Y201809CAC046. [Online]. Available: https://doi.org/10.2200/S00874ED3V01Y201809CAC046.

[34] D. Krushevskaja and M. Sandler, "Understanding latency variations of black box services," in *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, D. Schwabe, V. A. F. Almeida, H. Glaser, R. Baeza-Yates, and S. B. Moon, Eds., International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 703–714. DOI: 10.1145/2488388.2488450. [Online]. Available: https://doi.org/10.1145/2488388.2488450.

[35] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, 2014, pp. 127–144. DOI: 10.1145/2541940.2541941. [Online]. Available: https://doi.org/10.1145/2541940.2541941.

[36] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[37] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[38] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A spark benchmarking suite characterizing large-scale in-memory data analytics," *Clust. Comput.*, vol. 20, no. 3, pp. 2575–2589, 2017. DOI: 10.1007/s10586-016-0723-1. [Online]. Available: https://doi.org/10.1007/s10586-016-0723-1.

[39] H. Yuan and C. Wang, "A human action recognition algorithm based on semi-supervised kmeans clustering," *Trans. Edutainment*, vol. 6, pp. 227–236, 2011. DOI: 10.1007/978-3-642-22639-7_22. [Online]. Available: https://doi.org/10.1007/978-3-642-22639-7_22.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114. [Online]. Available: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[41] "Redis: An open source, in-memory data structure store," 2019. [Online]. Available: https://redis.io/.

[42] "Solr is the popular, blazing-fast, open source enterprise search platform built on apache lucene," 2019. [Online]. Available: https://www.elastic.co.

[43] L. Velikovich, I. Williams, J. Scheiner, P. S. Aleksic, P. J. Moreno, and M. Riley, "Semantic lattice processing in contextual automatic speech recognition for google assistant," in *Interspeech 2018, 19th Annual Conference of the International Speech Communication Association, Hyderabad, India, 2-6 September 2018*, B. Yegnanarayana, Ed., ISCA, 2018, pp. 2222–2226. DOI: 10.21437/Interspeech.2018-2453. [Online]. Available: https://doi.org/10.21437/Interspeech.2018-2453.

[44] D. A. Menascé, "TPC-W: A benchmark for e-commerce," *IEEE Internet Comput.*, vol. 6, no. 3, pp. 83–87, 2002. DOI: 10.1109/MIC.2002.1003136. [Online]. Available: https://doi.org/10.1109/MIC.2002.1003136.

[45] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993, ISBN: 1-55860-238-0.

[46] Y. Ding, E. Yan, A. R. Frazho, and J. Caverlee, "Pagerank for ranking authors in co-citation networks," *J. Assoc. Inf. Sci. Technol.*, vol. 60, no. 11, pp. 2229–2243, 2009. DOI: 10.1002/asi.21171. [Online]. Available: https://doi.org/10.1002/asi.21171.

[47] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, S. D. Gribble and D. Katabi, Eds., USENIX Association, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[48] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, Eds., IEEE, 2019, pp. 502–504. DOI: 10.1109/CLOUD.2019.00091. [Online]. Available: https://doi.org/10.1109/CLOUD.2019.00091.

[49] Y. Wang, G. Wei, and D. Brooks, "Benchmarking tpu, gpu, and CPU platforms for deep learning," *CoRR*, vol. abs/1907.10701, 2019. arXiv: 1907.10701. [Online]. Available: http://arxiv.org/abs/1907.10701.

[50] M. Christandl, P. Vrana, and J. Zuiddam, "Barriers for fast matrix multiplication from irreversibility," *Theory Comput.*, vol. 17, pp. 1–32, 2021. [Online]. Available: https://theoryofcomputing.org/articles/v017a002/.

[51] Y. Cao, L. Zhao, R. Zhang, Y. Yang, X. Zhou, and K. Li, "Experience-availability analysis of online cloud services using stochastic models," in *17th International IFIP TC6 Networking Conference, Networking 2018, Zurich, Switzerland, May 14-16, 2018*, C. Casetti, F. A. Kuipers, J. P. G. Sterbenz, and B. Stiller, Eds., IFIP, 2018, pp. 478–486. DOI: 10.23919/IFIPNetworking.2018.8696531. [Online]. Available: http://dl.ifip.org/db/conf/networking/networking2018/8A2-1570416570.pdf.

[52] B. Cai, R. Zhang, X. Zhou, L. Zhao, and K. Li, "Experience availability: Tail-latency oriented availability in software-defined cloud computing," *J. Comput. Sci. Technol.*, vol. 32, no. 2, pp. 250–257, 2017. DOI: 10.1007/s11390-017-1719-x. [Online]. Available: https://doi.org/10.1007/s11390-017-1719-x.

[53] H. Fuchs, M. D'Anna, and F. Corni, "Entropy and the experience of heat," *Entropy*, vol. 24, no. 5, p. 646, 2022. DOI: 10.3390/e24050646. [Online]. Available: https://doi.org/10.3390/e24050646.

[54] D. Inc, "Docker homepage.," 2019. [Online]. Available: https://www.docker.com/.

[55] "Numactl," 2019. [Online]. Available: https://github.com/numactl/numactl.

[56] M. A. Brown, "Traffic control howto," 2015. [Online]. Available: http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[57] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, IEEE Computer Society, 2016, pp. 456–468. DOI: 10.1109/ISCA.2016.47. [Online]. Available: https://doi.org/10.1109/ISCA.2016.47.

[58] "Perf tool.," 2014. [Online]. Available: https://perf.wiki.kernel.org/.

[59] S. Kanev, K. M. Hazelwood, G. Wei, and D. M. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, IEEE Computer Society, 2014, pp. 31–40. DOI: 10.1109/IISWC.2014.6983037. [Online]. Available: https://doi.org/10.1109/IISWC.2014.6983037.

[60] X. Jin *et al.*, "Qosmt: Supporting precise performance control for simultaneous multithreading architecture," in *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, R. Eigenmann, C. Ding, and S. A. McKee, Eds., ACM, 2019, pp. 206–216. DOI: 10.1145/3330345.3330364. [Online]. Available: https://doi.org/10.1145/3330345.3330364.

A Appendix

Table 7: Workload characterization and classification with K-means clustering algorithm for BG Applications.

Classes	BG Applications	CPU	Mem	Mem IO	LIC	IPC	Disk IO	Network
A	TensorFlow-AlexNet	0.674	0.600	0.918	0.616	0.006	0.000	1.000
	TensorFlow-GoogleNet	0.858	0.433	0.935	0.619	0.025	0.000	0.061
	TensorFlow-Inception3	0.801	0.859	0.955	0.621	0.114	0.000	0.055
B	TensorFlow-ResNet20	0.176	0.461	0.163	0.173	0.424	0.000	0.017
	Dwarf-wordcount	0.008	0.102	0.074	0.287	1.000	0.000	0.000
C	Spark-DecisionTree	0.082	0.234	0.061	0.115	0.911	0.078	0.027
	Scimark	0.034	0.000	0.017	0.515	1.000	0.000	0.000
	Spark-LinearRegression	0.059	0.224	0.050	0.104	0.715	0.247	0.004
	Spark-PCA	0.102	0.221	0.027	0.139	0.943	0.247	0.032
	Spark-Terasort	0.116	0.253	0.055	0.186	0.601	0.196	0.058
D	Alu	1.000	0.007	0.004	0.573	0.595	0.000	0.000
E	Spark-PageRank	0.079	0.224	0.007	0.082	0.215	0.087	0.018
	Spark-PregelOperation	0.088	0.221	0.045	0.130	0.487	0.247	0.000
	Spark-LabelPropagation	0.090	0.200	0.037	0.103	0.367	0.000	0.018
F	DiskIO	0.008	0.274	1.000	0.916	0.146	1.000	0.000
G	TensorFlow-LeNet	0.221	0.209	0.279	0.236	0.494	0.000	0.822
H	Dwarf-sort	0.008	0.046	0.076	0.294	0.994	0.356	0.000
	Spark-KMeans	0.107	0.248	0.066	0.063	0.759	0.000	0.048
	Parsec-Blackscholes	0.008	0.028	0.031	0.575	0.533	0.000	0.000
	Parsec-Canneal	0.008	0.039	0.107	0.581	0.604	0.000	0.000
	Parsec-Fluidanimate	0.008	0.024	0.051	0.587	0.538	0.000	0.000
	Parsec-Freqmine	0.008	0.027	0.030	0.580	0.302	0.000	0.000
	Parsec-streamcluster	0.008	0.005	0.095	0.597	0.060	0.000	0.000
I	Matmul	0.757	0.518	0.959	1.000	0.348	0.000	0.000
	TensorFlow-Inceotion4	0.860	1.000	0.983	0.619	0.089	0.000	0.054
	TensorFlow-VGG11	0.750	0.547	0.994	0.609	0.108	0.000	0.387
	TensorFlow-VGG16	0.718	0.595	0.984	0.608	0.127	0.000	0.204
J	BigDataBench-Union	0.213	0.161	0.106	0.563	0.187	0.000	0.000
	BigDataBench-OrderBy	0.167	0.175	0.086	0.595	0.341	0.000	0.000
	BigDataBench-CF	0.360	0.212	0.184	0.606	0.269	0.000	0.000
	BigDataBench-MD5	0.243	0.157	0.105	0.521	0.209	0.000	0.000
	BigDataBench-CC	0.129	0.143	0.084	0.592	0.143	0.000	0.000

Table 8: Workload characterization and classification with K-means clustering algorithm for latency-critical services.

Classes	LC Services	CPU	Mem	Mem10	LIC	IPC	Disk IO	Network
1	Image-classify	0.956	0.289	1.000	1.000	0.663	0.000	0.108
	ResNet-50	0.737	0.735	0.692	0.956	0.713	0.000	0.099
2	Redis	0.239	0.069	0.029	0.127	0.228	1.000	0.076
	Memcached	0.614	0.203	0.068	0.108	0.604	0.000	1.000
	Mongodb	0.170	0.101	0.014	0.028	0.010	0.637	0.019
3	Solr	0.430	1.000	0.193	0.165	0.099	0.000	‘0.053
4	Speech-recog	1.000	0.085	0.547	0.971	0.673	0.000	0.368
	Half	0.229	0.059	0.074	0.841	1.000	0.000	0.327
	SSD	0.781	0.351	0.453	0.857	0.594	0.000	0.288
	Mobilenet	0.367	0.332	0.191	0.857	0.644	0.000	0.328
	Catdog	0.371	0.094	0.109	0.713	0.842	0.000	0.205
	LSTM	0.947	0.181	0.265	0.822	0.436	0.000	0.004
5	TPC-W	0.017	0.054	0.016	0.160	0.307	0.000	0.052
	MNIST	0.048	0.012	0.018	0.472	0.287	0.000	0.012
	TextCNN	0.207	0.025	0.025	0.489	0.505	0.000	0.012
6	SocialNetwork	0.288	0.062	0.061	0.143	0.208	0.000	0.007
	Xapian	0.059	0.000	0.024	0.131	0.297	0.000	0.001
	Img-dnn	0.041	0.019	0.020	0.154	0.525	0.000	0.001
	Masstree	0.061	0.072	0.023	0.162	0.099	0.000	0.001