2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-7281-9586-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCGrid51090.2021.00032

CASH: A Credit Aware Scheduling for Public Cloud Platforms

Aakash Sharma*, Saravanan Dhakshinamurthy†, George Kesidis*, Chita R. Das*
* Computer Science and Engineering, The Pennsylvania State University, † Facebook Inc.
{abs5688, gik2, das}@psu.edu, saravanand@fb.com

***Abstract*—Distributed data processing frameworks such as Hadoop, Tez, Spark, and Flink are exclusively used by public cloud tenants for executing large scale data analytics applications in various domains including but not limited to content management, financial sector, healthcare etc. These frameworks slice a job into a number of smaller tasks, which are then executed by a job scheduler on a multi-node compute cluster. While making scheduling decisions, the State-of-art schedulers employed in these frameworks assume hardware resources such as CPU, disk I/O and network I/O to offer a fixed service rate. However, in a public cloud environment, many of these resources are associated with burstable service rates. More specifically, the resources offer a guaranteed baseline service rate with an option to *burst* above their baseline rate by expending accumulated *burst credits*. Being unaware about this underlying hardware *burstability*, schedulers tend to make sub-optimal task placement decisions, thereby adversely affecting the job completion times, leading to higher deployment costs.**

In this paper, we propose CASH, a *burst credit aware* scheduler, which is cognizant about the burst credits associated with the individual hardware resources in the public cloud cluster. Through coarse grained task annotations depicting the burst credit demand of individual tasks and dynamically monitoring the credits for the underlying resources, CASH performs optimal task placement decisions. We prototype CASH on YARN, Hadoop, and Tez, and extensively evaluate it using both batch and streaming workloads. Our experimental results with CASH show CPU-credit based instances, like AWS T3, are a viable cost effective alternative when compared to self-managed offerings like Amazon EMR, for running large scale batch workloads. Furthermore, we demonstrate that CASH can accelerate streaming SQL queries on a large Hive database by up to 39.4% , leading to public cloud cost savings by up to 22%.

***Index Terms*—Public Cloud, Burst Credits, Cluster Scheduling, Parallel Data Processing, Cost Savings**

I. Introduction

Public cloud computing has become a ubiquitous part of every enterprise IT infrastructure owing to its flexibility and cost savings when compared to private data centers. Public cloud offerings include enterprise services spanning compute, storage and networking. Since many of the hardware resources are shared in a multi-tenant setting, cloud operators provide both fixed service rate and burstable service rate, in an effort to maximize their cluster utilization .

Tenants running *bursty* workloads whose service rate demand fluctuates with time can particularly benefit from the *burstable* service rate offering of the public cloud. When the service rate demand for a particular burstable resource is less than the baseline (guaranteed minimum) rate of the resource, *burst credits* accumulate in the burst bucket of the resource limited by the bucket capacity. These accumulated *credits* can later be used to *burst* above their baseline service rate up to the maximum rate allowed by the burst bucket. The expected service rate of burstable resources is determined by their baseline service rate and the accumulated *burst credits* in their associated *burst bucket*.

In terms of CPU *burstability*, cloud operators like AWS [1] and Azure [2] offer burstable Virtual Machines (VMs)[1] that have burstable CPU service rate and are less expensive than regular VMs. These VMs are resource-provisioned with the peak resource allocations as a regular VM and are attractive to the budget-conscious tenant. Along with burstable CPU services, block storage solutions such as AWS-EBS [3] and Azure premium SSD [4] also offer burstable I/O performance. In the case of AWS, tenants are provided with EBS as the general-purpose persistent block storage service across all instance offerings [5]. Besides CPU and storage, network burstability is also present across predominantly used AWS VM instances (VMs within "8xlarge") [5]. Public cloud providers expose the different forms of *burstability* through Service-level-agreements (SLAs). Specifically for CPU and storage, providers publish burst metrics which can be queried by tenants to determine the expected service rate of corresponding resources at any given time. Tenants can use the published burst credit metrics of their hardware resources to accelerate workload completion time and thereby save costs. *However, state-of-art data processing framework schedulers are not cognizant about burstability in their cluster resources and make sub-optimal scheduling decisions.*

Distributed data processing frameworks such as MapReduce [6], Spark [7], Flink [8] and Tez [9] work by slicing a particular job into various small microtasks[2] [11] and execute the tasks by using a job scheduler or an external cluster manager. A middleware cluster manager, e.g., Kubernetes [12], Mesos [13] or YARN [14], serves as a resource arbitrator, i.e., it offers resources to a framework for its task execution. The scheduling algorithms employed in cluster managers typically consider every hardware resource in the cluster in an (often

[1]VMs are herein a.k.a. nodes or instances.

[2]Such microtasks could also be executed using serverless cloud functions, which are arguably simpler to use than VMs and more readily available, but are more costly and have I/O limitations. In [10], for a partially predictable workload, we argue for the cost-effective use of serverless functions (jointly with VMs) for purposes of autoscaling.

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00032

crude) attempt to provide a fixed service rate. In the case of burstable CPU resources, having an assumption that all cores deliver the same service rate would be sub-optimal. Besides CPU, tasks may also be allocated on a VM with other critical (for the task) hardware resource, whose service rate is throttled (baseline rate) such as storage and network. While storage and network resources are typically not taken into consideration for making scheduling decisions, these tasks can potentially straggle due to a lower service rate availability than their demand. Hence, it is essential to identify and exploit this *burstability* in data processing frameworks.

In this paper, we propose a novel burst credit aware cluster scheduler – CASH – that leverages both the burst credit state associated with its hardware resources and the (only roughly disclosed) critical resource needs of the tasks it is scheduling. CASH queries the cloud provider for the burst credit state and categorizes task requests with their critical resource needs through the application frameworks. Unlike prior works like [15] (HeMT), which only consider CPU burst credit for determining the input partition size of long running data-processing jobs (unstructured input only) for the VMs of the cluster, CASH can be applied over a wide variety of workloads including batch (long running) and database query (short and structured). Also, while HeMT relies upon more complex workload profiling, CASH works with only coarse grained task categorization and can generically work for any burstable hardware resource like CPU, storage and network.

In summary, we make the following contribution in this work:

- *We conduct qualitative profiling of a job* by categorizing task requests according to their critical resource needs, i.e., CPU, storage I/O or network I/O. The application framework "annotates" task requests with their burst credit demand before sending it to the scheduler or the cluster manager.
- *CASH makes efficient use of burst credit information* associated with hardware resources of the cluster VMs. The scheduler matches burst credit annotated tasks to VMs based on decreasing priority of corresponding burst credit state.
- *We prototype CASH on YARN, Tez and Hadoop*. Specifically we modify the frameworks to associate vertices of a job's execution DAG to a critical resource. Tasks are then annotated based on their associated vertex.
- *We extensively evaluate CASH* using both batch workloads and streaming SQL queries. Our experimental analysis demonstrates that credit based burstable VMs are a viable and economical solution for executing batch based workloads. Further, our results show a job completion time reduction by a maximum of 39.4% for streaming SQL queries which translated to an overall wallclock completion time shortening by a maximum of 22%.

This paper is organized as follows. In section II, we provide some background discussion. In section III, we motivate our problem in detail, including through experimental case studies.

Type	vCPUs	Memory (GiB)	Baseline Performance/vCPU	CPU credits earned / hr
t3.large	2	8	30%	36
t3.xlarge	4	16	40%	96
t3.2xlarge	8	32	40%	192

TABLE I
AWS T3 CPU CREDITS

Our scheduling scheme is described in section IV and how it was prototyped on Tez/Hadoop-over-YARN is described in section V. The results of our experimental performance evaluations are given in section VI. In section VII we discuss related work. Finally, we summarize in section VIII and discuss future work.

II. Background

In this section, we discuss some service offerings by AWS, the world's largest public cloud provider, which are relevant to this paper. Other large public cloud providers (including Azure and GCE [16]) offer similar services. We particularly stress variable-rate CPU and storage I/O service aspects.

A. Credit Burst

1) CPU Burst: AWS burstable instances, a.k.a. T series instances, have a guaranteed baseline CPU service rate which is a fraction of an actual VM CPU core of a comparable general purpose (M series) type. These instances have a constant flow of CPU burst credits based on their size in memory. Table I provides a few AWS T3 instance sizes, their configurations and their CPU credit properties from the AWS website [17]. The baseline service rate ($\mu_{\mathrm{CPU}}^{\mathrm{baseline}}$) can be calculated from the credits earned per hour (λ_{CPU}) and the number of cores (n) of the instance as:

$$\mu_{\mathrm{CPU}}^{\mathrm{baseline}} = (\lambda_{\mathrm{CPU}}/n)/3600 \tag{1}$$

The instance acquires surplus CPU burst credits (B_{CPU}) when the instance CPU utilization rate (U_{CPU}) is less than μ_{CPU}. For newer generation burstable instances B_{CPU} is initialized to 0 at startup and may change depending upon $\mathrm{U_{CPU}}$ and calculated as:

$$B_{\mathrm{CPU}}^{\mathrm{new}} = max[0, B_{\mathrm{CPU}}^{\mathrm{old}} + (\mu_{\mathrm{CPU}}^{\mathrm{baseline}} - (U_{\mathrm{CPU}})] \tag{2}$$

Each surplus CPU credit can be used to "burst" to 100% CPU for one minute or 50% CPU for 2 minutes. Credit earn and expenditure of each instance is calculated by AWS at millisecond granularity but available at 5 minutes granularity using APIs provided by AWS.
AWS T3 also supports an unlimited credit option which prevents tenants from being throttled to baseline service rate if they run out of surplus credits. The surplus credit balance formula changes to:

$$B_{\mathrm{CPU}}^{\mathrm{new}} = B_{\mathrm{CPU}}^{\mathrm{old}} + (\mu_{\mathrm{CPU}}^{\mathrm{baseline}} - U_{\mathrm{CPU}}) \tag{3}$$

Any negative B_{CPU} is billed on a 24 hours granularity or the instance lifetime, whichever is shorter.

2) Storage Burst: AWS Elastic Block Storage (EBS) is a storage volume service that can be attached to any AWS instance. These volumes offer persistent storage and their lifetime is independent of the instance they are attached to. AWS SSD volume performance is *burstable* as well and is measured in the unit of IOPS (Input/Output per second). In case of EBS SSD, the baseline credit rate ($\mu_{\text{storage}}^{\text{baseline}}$) is simply three times the volume size in GB:

$$\mu_{\text{storage}}^{\text{baseline}} = 3\, IOPS\, per\, GB\, of\, Volume\, Size \quad (4)$$

Similar to CPU burst, whenever the storage volume performs lesser operations than its baseline rate, credits are conserved. Each accumulated surplus credit can be used by the volume to burst to the maximum performance of 3000 IOPS. Give a volume's utilization rate measured in operations performed in a second ($\text{Ops}_{\text{storage}}$), surplus disk credit ($B_{\text{disk}}$) can be calculated similarly to CPU as:

$$B_{\text{storage}}^{\text{new}} = \max[0, B_{\text{storage}}^{\text{old}} + (\mu_{\text{storage}}^{\text{baseline}} - \text{Ops}_{\text{storage}})] \quad (5)$$

3) Network Burst: Public cloud providers do not explicitly expose their network variability [18] and instead provide a rough ballpark figure of the expected network performance [5]. [18] empirically showed that AWS and GCE use obscure dual token bucket mechanisms to regulate their network performance. However, there is no publicly available API to query expected network service rate or the state of network credits for an instance.

B. Relevant AWS Services

We list two services by AWS relevant to this research in this section.

1) AWS Elastic MapReduce (EMR): AWS EMR [19] is essentially a SaaS offering by AWS which runs data processing frameworks like Hadoop, Spark, Tez, etc., along with databases such as Hive [20]. Tenants can provision EMR within minutes and run workloads to process their data without the hassle of any configuration or system administration. EMR comes with the YARN capacity scheduler as the cluster manager. Tenants choose the type of hardware they want to use to create their cluster.

2) AWS S3: AWS S3 [21] is an object based storage service offering by Amazon which offers cheap and reliable storage service. Tenants of S3 can expect to receive 99.999999999% durability for their data [21] along with flexibility to store data of any arbitrary size and access it from anywhere over the internet. More importantly, object store such as S3 provides the cheapest storage option in the public cloud. On AWS, while SSD backed volume costs $0.10 per GB and a HDD backed volume costs $0.045 per GB, S3 costs only $0.023 per GB for the first 50 TB with a $0.0004-$0.005 per API call charge to data objects [21].

III. Motivation

A. CPU Utilization in Compute Clusters

Large compute clusters commonly face low resource utilization and efficiency even after collocating online services and batch workloads. Analysis [22], [23] done on recently released Alibaba production traces [24] shows that average CPU utilization remains below 40% for about 60% of the machines and below 50% for 75% of the machines in their cluster. However, the CPU usage follows a *bursty* pattern with CPU utilization going above 60% for only about an hour in a 24 hour window for majority of the machines. This should prevent Alibaba from using fewer CPUs to run their workloads as it would violate their SLAs due to transient increases in CPU demands. Analysis of the Taoboa Hadoop cluster [25] also found CPU utilization to not exceed 40%. Even the Google trace analysis [26] reveals that CPU utilization doesn't exceed 60%, where the trace comprised of a mix of long running services, MapReduce and HPC. An additional factor which contributes to low CPU utilization in a public cloud is a preference to use cheap object based external storage which has high I/O latency [27], [28]. We also tested CPU utilization using Hibench [29] batch workloads on EMR and we discuss the observations in the sequel.

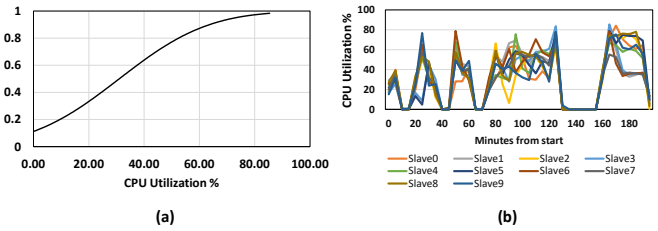

Fig. 1. CPU utilization in EMR: (a) CDF of CPU utilization, (b) CPU utilization timeline

1) CPU utilization on EMR: We observed low CPU utilization while running HiBench test workloads with a utilization average of about 30% per VM. We plot the CDF of CPU utilization sampled at 1 minute intervals for all VMs in the cluster in Figure 1(a). The low CPU utilization is primarily due to high object read/write latency of S3 [28] which will throttle any application on I/O. One might be tempted to reduce the number of VMs to improve CPU utilization, but this will further degrade I/O performance as there will be less parallelism during read/write to S3. The timeline of CPU utilization on EMR is given in Figure 1(b) and a *bursty* pattern of CPU utilization can be clearly seen.

2) CPU Burst Credits in Scheduling: Workloads that have low *average* CPU utilization can particularly benefit from the low cost offering of burstable instances. M5 (regular) instances are more than 15% pricier than T3 instances and EMR on M5 is more than 44% pricier than on (burstable) T3. Tenants can obtain significant cost savings by using credit based CPU instances over regular instances with the same peak resource allocations. For example, workloads using object stores such as S3 and MapReduce workloads are often low CPU utilization workloads and can particularly benefit from low cost, low CPU throughput instances like T3. That said, random scheduling approach on burstable instances leads to sub-optimal job execution time due to the uneven consumption of CPU credits in the nodes of the cluster. We give a timeline

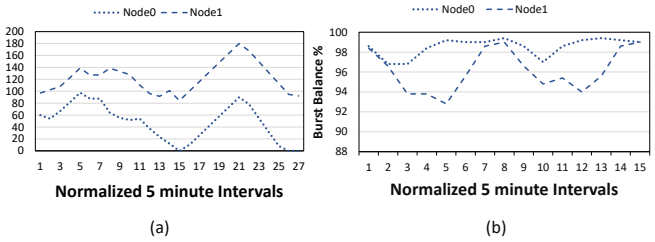

Fig. 2. Burst Credit Balance on Two Nodes: (a) CPU Burst Credits, (b) Storage Burst Credits

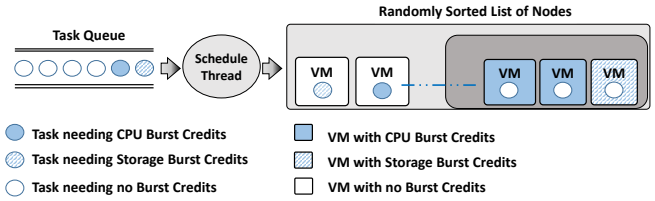

Fig. 3. Scenario depicting worst case scheduler behavior. Tasks needing burst credits are allocated to VMs without burst credits. On the other hand, tasks needing no burst credits are allocated on VMs with available burst credits.

of CPU burst credit consumption of two of our VMs from one of our experiments in figure 2(a). Tasks that are assigned on the VM with 0 credits are throttled while the other instance have a large amount of CPU credits accumulated. This is an artifact of the opaqueness of the service rate variability to the job scheduler.

B. Storage Burst Credits in Scheduling

An analysis of TPC-DS workloads in [30] shows very high I/O utilization (up to 100%) for their big-data systems based on HDFS. Anecdotal evidence also suggests that storage is typically a significant bottleneck in I/O intensive workloads. However, similarly to burstable CPUs, public cloud providers offer burstable storage volume offerings as well and as explained in section II. Hence, it is consequential to use storage burst credits while running heavy I/O workload and exploit the burst performance available to them in the cluster.

Storage I/O Burst Credits in Scheduling: We ran a two VM experiment with TPC-DS workload and observed the changes in storage burst credit state of each VM as shown in Figure 2(b). We observe a significant difference in consumption of storage burst credits between the two VMs in the cluster. While this does not cause any slowdown as both the VMs in the cluster have a full storage burst credit balance, we notice the uneven consumption in burst credits which has the potential for slowing down tasks if the storage volumes were running low on burst credits. This is similar to the problem of underlying *opaqueness* in CPU service rate in our previous experiment.

C. Problem Description

Job schedulers like YARN choose VMs for scheduling tasks in random order. A job scheduler will not differentiate between VMs which have burst credit balance and VMs which have been throttled for either CPU or storage (or network) access. In the worst case, tasks needing burst credit for a particular resource (CPU, storage, etc.) may be scheduled on VMs with insufficient credits corresponding to the tasks needs. This scenario is depicted in Figure 3. This worst case scenario will lead to significant degradation in execution time owing to tasks being scheduled on throttled VMs and thereby increase the cost of running the workload on the public cloud. Hence, there is a need to bring *cognizance* of the underlying burst credit mechanism to the job scheduler.

Suppose we need to execute N tasks t_n, $n \in \{1, ..., N\}$ on a cluster having M VMs. Also suppose there are K burstable hardware resources on each of the M VMs and that each task t_n primarily needs one bottleneck burstable hardware resource $k \in \{1, ..., K\}$ for work $w_{n,k}$. The service rate of the k^{th} hardware resource on VM $m \in \{1, ..., M\}$ is $\mu_{m,k}$, where $\mu_k^{\text{baseline}} \leq \mu_{m,k} \leq \mu_k^{\text{peak}}$, and the time needed to run task t_n on VM m is

$$\tau_n = \frac{w_{n,k}}{\mu_{m,k}}. \tag{6}$$

To elaborate equation 6 further, suppose a task is bottle necked on one hardware resource (say disk) and the resource has a certain service rate or "speed". The time spent for task execution would be the work that needs to be accomplished (say reading a few sectors from the disk) divided by the "speed" of the underlying resource. The speed or the service rate $\mu_{\text{m,k}}$ is the disk (k) speed on VM m.
Now, we calculate cumulative elapsed time for all tasks (spent using bottleneck hardware) as:

$$\tau_{\text{cum}} = \sum_{n=1}^{N} \tau_n \leq \sum_{n=1}^{N} \frac{w_{n,k}}{\mu_{\text{k}}^{\text{baseline}}}. \tag{7}$$

Consider a job which is sliced on to several tasks. In order for the job to complete, all its tasks must run to completion. Assuming such task is bottleneck on one hardware resource, we can sum equation 6 for all tasks to get a cumulative elapsed time. This cumulative elapsed time would always be less than or equal to the time it takes for all tasks to run on their respective bottleneck hardware with reduced (baseline) speeds. This is our equation 7.

And finally, our scheduling objective is:

$$\min_{w} \sum_{n=1}^{N} \frac{w_{n,k}}{\mu_{m,k}}. \tag{8}$$

i.e. we want to minimize the cumulative time spent on all task execution by varying the work that is assigned on various VMs.

IV. Proposed Scheme

We propose CASH, a novel scheme which utilizes burst credits through two complimentary techniques. First, task requests are annotated in the application framework in order to attribute them to be needing burst credits. Second, we build a burst credit aware scheduler which periodically collects burst credit and usage information (CPU or storage I/O) of the cluster VMs from the public cloud and uses this information

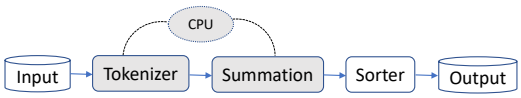

Fig. 4. A Tez OrderedWordCount DAG with CPU Annotation

for task placement. The two techniques of the scheduling scheme are discussed below.

A. Task Annotation

In general, CASH supports annotations of tasks as being intensive with respect to particular hardware resources, e.g., CPU, storage I/O, and network I/O. The annotations happen via the framework which has a good knowledge about the characteristics of its tasks. The scheme uses either CPU burst credits or storage burst credits but not both (i.e., one will be more of a bottleneck than the other). The choice of using either CPU or storage I/O burst credits is a user configuration. *These annotations happen through the framework based on the characteristics of the DAG vertex a task is associated with*. The framework follows a simple heuristic that vertices that are expected to heavily use a particular type of resource can be annotated as such. For e.g. vertices of type "shuffle" transfer a lot of data over the network and are annotated to be "network". Users also have an option to manually attach annotation to vertices or override the framework attached annotation. Essentially, users are free to mix and match annotations with vertices of their choice based on the type of tasks they wish to prioritize for burst credits. We provide implementation-specific details on task annotation in section V. Schematic details in sequel.

1) CPU Annotation: CPU annotation may be used while using lower cost burstable instances. Tasks requests annotated as CPU are prioritized to be run over VMs with CPU burst credits. *The framework annotates task requests belonging to "map-like" vertices such as "map", "initialmap", "tokenizer" etc. of the DAG as CPU*. This is because generally map vertices of a DAG involve the bulk of the workload processing and so utilize CPU intensively. These CPU intensive map tasks need CPU burst credits to avoid slowdown (and heightened possibility of being deemed stragglers). Hence, task requests belonging to map vertices of a DAG are annotated for CPU credits. Users may also attach CPU annotation to vertices such as "sort" in their application code. A sample annotated orderedWordCount DAG is given in figure 4 and its shown that the framework has attached CPU annotation to map-like vertex – "Tokenizer" and "Summation".

2) Storage Annotation: DAG vertices which are the source of initial data in the DAG are generally at a greater need of storage burst credits than the rest of the vertices of the DAG. This is because a workload which is I/O intensive such as database queries will need to process a large amount of data during initial read from storage drive resulting in large disk spillage. Hence such tasks should be assigned on VMs with storage burst credits and are annotated accordingly. A sample annotated terasort DAG is given in figure 5 and its shown that the framework has attached Storage annotation to data source vertices of type Map. Again, the user may choose to annotate vertices of their choice such as "sort" to be storage annotated.

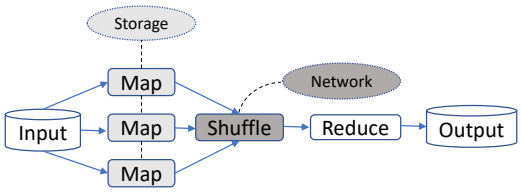

Fig. 5. A Tez TeraSort DAG with Storage and Network Annotations

3) Network Annotation: The vertices of reduce-like tasks (e.g., "reduce", "shuffle" and "collate") are typically less resource hungry compared to vertices of map-like tasks and can be assigned on VMs where CPU/storage burst has been throttled. *However, the reduce phase is generally network intensive and the framework attaches a network annotation for reduce-like tasks*. This leads to load balancing of network tasks in the cluster. AWS or other cloud providers do not expose their network credit information[3] and hence we use simple load balancing to approximate optimal network credit consumption. We distribute the network tasks "fairly" across the cluster so that no VM accumulates too many network credits compared to any other. As a result of this, we observed marked improvement in reduce task execution time with MapReduce as reduce tasks are heavily network intensive. The network annotation is attached along with CPU or storage annotation and a sample is given in figure 5. The "Reduce" vertex in the figure is not annotated as its not network heavy due to the presence of "Shuffle" vertex.

B. Credit Aware Scheduling

We modify the cluster-manager's (YARN's) scheduler to make scheduling decisions based upon the burst credit balance of the bottleneck hardware of the VMs. The bottleneck hardware is a user configuration, which is currently either CPU or storage. Figure 6 shows the main components of the proposed scheduler CASH. The optimizations are described below for two cases: scheduling burstables (AWS T3) based on CPU burst credits and scheduling regular instances (AWS M5) based on storage burst credits. In both cases, different types of tasks of a job stream are considered by YARN. Each YARN node which is a logical abstraction of a physical compute unit such as a VM, has a number of slots (each corresponding to a pre-configured vCPU or virtual core) so that a node can simultaneously execute more than one task, i.e., one task per slot. We assume the cluster manager pools all pending (annotated) tasks from all of its application frameworks into a single task queue.

At a long (one minute) time scale, YARN nodes are ordered in decreasing order of their corresponding VM CPU/storage

[3]AWS's unorthodox dual token-bucket mechanism for network I/O of its burstable instances was reverse engineered in [18].

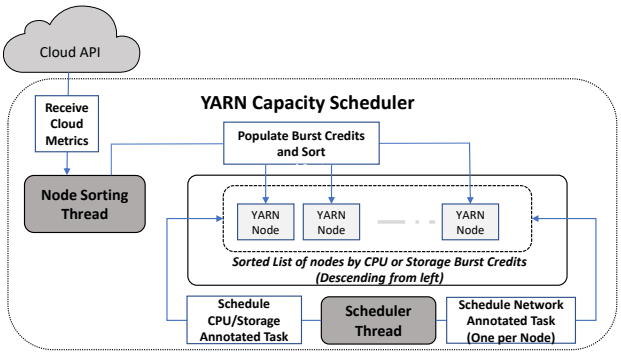

Fig. 6. A high-level overview of the proposed scheduler

burst credit balance (node sorting thread). At a short (milliseconds) time-scale, the cluster manager's scheduler (scheduler thread) first visits each node in descending order of burst credits and assigns to it as many burst (CPU or storage) annotated tasks as possible given its current number of free slots, before proceeding to the next node. This phase ends after all YARN nodes have been visited. In the second phase, non-burst intensive tasks which are annotated as network are considered by the scheduling thread. Starting from the node with the *least* burst credits, in each round at most a single free slot per node is allocated to such tasks in an effort to load balance such tasks among the nodes and reduce the risk of network congestion. Note here that network tasks are long running tasks and can easily tolerate slight delays in scheduling. In the final phase, any remaining (non-annotated) tasks are assigned to available free slots (if any) in arbitrary node-order. Algorithm 1 describes our credit based scheduling logic.

Algorithm 1: Schedule Thread of CASH

```
nodeList ← nodes in decreasing order of burst credits
sleepInterval ← Time interval in between scheduling
while Scheduler is running do
    forall node ∈ nodeList do
        tasks ← getBurstAnnotatedTasks()
        AttemptToScheduleTasks(node, tasks)
    end
    reverseList ← reverse(nodeList)
    forall node ∈ reverseList do
        tasks ← getNetworkAnnotatedTasks()
        AttemptToScheduleOneTask(node, tasks)
    end
    randomList ← shuffle(nodeList)
    forall node ∈ randomList do
        tasks ← getRemainingTasks()
        AttemptToScheduleTasks(node, tasks)
    end
    sleep(sleepInterval)
end
```

One pass of the scheduling thread is at the milliseconds time scale during which new tasks may be generated by the application frameworks and slots may become freed up. These new tasks are scheduled in the next iteration of the scheduling thread.

V. Implementation

We have developed a prototype of our burst credit aware task scheduler within the YARN capacity scheduler and conducted experiments through Tez and Hadoop frameworks. We made changes to Apache Tez to annotate task requests and leveraged the existing node label feature of Hadoop to pass on annotations to YARN. All changes were made in java. We discuss our implementation in the sequel.

A. Hadoop

Every Hadoop job can be expressed as a DAG with two vertices - "map" and "reduce". We associate burst annotation (CPU or storage based on user configuration) with map vertex and network annotation to reduce vertex. Hadoop users have an option to specify node labels for their tasks (map or reduce) which is then passed to YARN as part of task requests. We specify our annotations as labels for map and reduce tasks in user configuration. YARN recognizes these "special" labels and interprets them as task annotation for its scheduling logic. Using this existing feature, we were able to use Hadoop for CASH without any code changes.

B. Tez

Tez comes with an abstraction called vertex managers which can be used by Tez users to dynamically control the characteristics of Tez vertices, such as task parallelism. These managers are associated with vertices by Tez based on the individual vertex features. We exploit these managers to implement our annotation logic. These vertex managers annotate the task requests to YARN for their associated vertices. We modify two such vertex managers – *RootInputVertexManager* and *ShuffleVertexManager* – to implement our burst credit logic. The *RootInputVertexManager* is associated with vertices that are the source of input data in the DAG and hence can be used to annotate tasks with storage annotation. The *ShuffleVertexManager* is associated with vertices that shuffle data over the network and we attach network annotation to them. For CPU annotation, we simply infer the vertex type in Tez runtime and assign annotation if the vertex is one of preselected set of CPU intensive vertices. Tez allows its users to create their own vertex managers and associate them explicitly with the vertices of the DAG. We have added the annotation feature to the base level class of vertex manager in Tez source code allowing users to associate annotation with their user defined vertex managers. This allows users to associate any annotation with any vertex of their choice in their execution DAG.

C. YARN

We extract the burst credit balance of each VM from Amazon Cloudwatch [31] every 5 minutes and update the internal YARN node data structure for making scheduling

decisions. Cloudwatch populates burst credit balances at the smallest interval of 5 minutes. Since we do not want YARN to make scheduling decisions based on stale burst credit information, we also pull CPU utilization for each VM or storage IOPS for each storage drive from Amazon Cloudwatch every 1 minute and predict the burst balance. Prediction is made easy by the fact that AWS exposes the exact formula to calculate burst credits at any given point of time based on the instance/storage size and its CPU/storage utilization and as given in Section II. The instance type and storage size can also be retrieved from Cloudwatch. We update the internal YARN node data structure with predicted burst credit every 1 minute and actual burst credit every 5 minutes. This is done in a separate asynchronous thread inside the YARN capacity scheduler and is part of our design Figure 6.

VI. Evaluation and Results

In this section, we provide some exemplary experimental results of our prototype. We use batch based workloads for CPU burst evaluation and streaming SQL queries for storage burst. As summarised in our objective equation 8, we want to minimize the elapsed time for all tasks in the workload. We compare the cumulative task elapsed time (CPU burst) and query run time (storage burst) to stock YARN in the evaluation of our prototype. We only use time as a performance metric because we want to finally reduce the cost of running workloads on the public cloud. Details in sequel.

A. CPU Burst

We run PageRank, K-means clustering and Hive SQL aggregation from Intel HiBench test suite [29] for CPU burst evaluation. These workloads are representative of popular Hadoop batch workloads. Pagerank and K-means have an *average* CPU demand less than the baseline CPU service rate of the burstable instances that we use and SQL aggregation have a higher demand than the baseline CPU service rate. The input data for these workloads are generated synthetically through HiBench and written to AWS S3 [21]. Each HiBench workload comprises several jobs, submitted sequentially, with the input of a job being dependent on the output of the job prior to it. All jobs read their input data from S3 and write their output data to S3. We use S3 as storage due to it being the cheapest [21] and a popular storage offering for MapReduce workloads [27].

1) Experiment Design and Setup: We compare our scheme against EMR and stock YARN. Comparing against EMR removes any biases that we might inadvertently introduce into the comparison. Our EMR cluster consists of 10 EC2 M5.2xlarge instances and burstable cluster running stock YARN consists of 10 EC2 T3.2xlarge instances. We run all experiments in an orderly manner such that burst credits are made available for high CPU demanding SQL aggregation workload by the low CPU demanding workloads in the beginning. In a real world setup, this effort is not needed as the burst credits will get averaged out after the first batch run cycle. We run workloads in the order PageRank, K-means and SQL aggregation except for the worst case analysis when we run them in the order SQL aggregation, followed by PageRank and K-means.

The first two experiments – (i) **EMR**, uses general purpose instances to create the best case scenario for our evaluation and, (ii) **Ordered workload submission**, uses burstable instances with workloads submitted in an orderly manner to gauge the performance of stock YARN running over burstable hardware. We then evaluate the (iii) **Worst-case scenario** in our third experiment by running high CPU workload on the cluster with depleted burst credits and finally, evaluate (iv) **CASH** in our fourth experiment by comparing it against (i) and (ii). We also run a fifth experiment (v) **T3 unlimited**, in which burstable instances with unlimited option ON is evaluated with stock YARN.

2) Results: We report experimental results through a cumulative elapsed time comparison of the three phases a job goes through in Hadoop, namely map, shuffle and reduce. Figure 7 provides the results observed. We do not report the overall makespan of our workloads as access to AWS S3 storage varies widely over time [28]. This may be due to dynamic demand or due to an undocumented token-bucket mechanism or both. For this reason, in the experimental results that follow, we report component task execution-times rather than overall workload wall-clock (makespan) execution times. Also, considering EMR is SaaS, we expect that generally, S3 variation will be larger for non-SaaS implementations proposed herein. We discuss the results in the sequel.

*a) **Elapsed Time**:* CASH performs faster than ordered workload submission by 6% in elapsed time over all phases and degrades by about 13% when compared to EMR. However, running T3 is about 30.7% cheaper than running EMR (not shown in graphs) and hence tenants would save on billing cost by running their workloads on T3 using CASH. Running workloads over T3 by simply using stock YARN in the case of ordered workload submission leads to a degradation of about 19% compared to EMR which doesn't translate to as much cash savings as CASH. In fact in the worst case, running SQL aggregation over stock YARN caused as much as 111% degradation in cumulative elapsed time compared to EMR as shown in figure 7 (d). All tasks of this workload were throttled to baseline CPU service rate due to unavailability of any CPU burst credits. As implied by equation 7, the total elapsed time of the job is the worst case scenario when its critical burstable resource need (CPU) is throttled. Hence, its safe to assume this degradation to be the worst case upper bound. In the best case, all tasks of the job experience peak service rate which happens when workloads are run over EMR (general purpose instances). CASH's performance is near the best case scenario and therefore shields tenants from degrading near to the worst case which may be caused by the random scheduling algorithms of YARN.

*b) **CPU Utilization**:* We further assert on the fact that running workloads via CASH would yield better task execution time compared to stock YARN by looking at CPU utilization of the clusters. T3 instances running CASH show

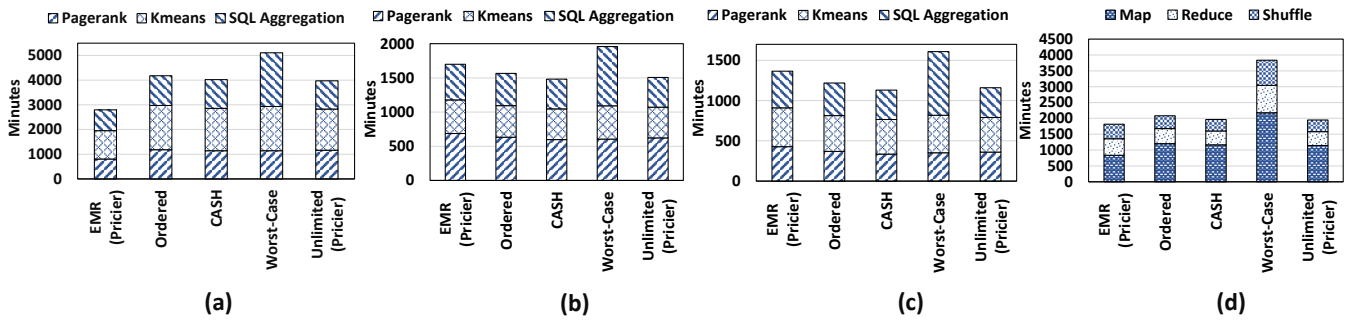

Fig. 7. Cumulative elapsed time: (a) Map, (b) Reduce, (c) Shuffle; (d) Stacked Elapsed Time: SQL aggregation

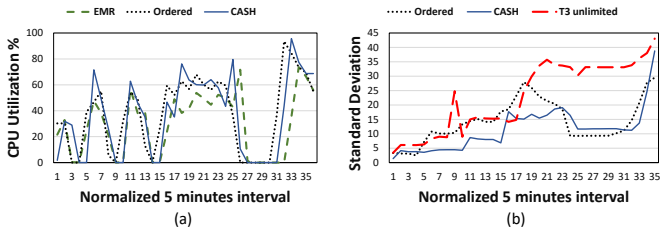

Fig. 8. (a) Average CPU Utilization Timeline, (b) Standard deviation of CPU Credit Balance

higher CPU utilization compared to ordered submission and EMR, pointing to better load balancing as shown in Figure 8(a). This will protect the tasks from being scheduled on YARN nodes which are throttled, degrading task execution time.

*c) **T3 unlimited**:* Elapsed time on T3 with unlimited option ON is about the same as CASH but the former has caveats. T3 unlimited averages CPU utilization on a per instance basis instead of the entire cluster. Tasks can be scheduled on VMs which have zero credit balance causing them to be billed for additional credits. This is possible while there are other VMs present in the cluster with surplus credits. Hence, tenants will be billed for additional credits while there are surplus credits available in the cluster. This phenomenon can be ascertained by observing the standard deviation of CPU credit balance across all VMs of the cluster in Figure 8(b). Hence, running workloads on CASH is cheaper than on T3 unlimited.

B. Storage Burst

We use hive-testbench [32] to run streaming queries for storage burst evaluation. Hive-testbench is a benchmark suite based on industry standard TPC-DS queries [30] to test database systems. We use three TPC-DS queries to be run over Hive database using Tez – query 66, query 49 and query 37. These queries are chosen such that they have high I/O requirements including the tendency for disk spillage. Input data is generated by hive-testbench and is stored in HDFS as a Hive warehouse. The application stack is Hive-2.3.6 with Tez-0.9.2 over YARN-2.8.5, the equivalent versions used in EMR.

1) Experiment Design and setup: All three TPC-DS queries are run in parallel and to avoid bias due to query caching, the feature is disabled in Hive. The VMs of the cluster are restarted in between all experiments and random data is written on the storage volumes so as to also invalidate the disk cache. We test our optimization gains by comparing execution time of individual query and wall-clock completion time of all queries against running the same queries on stock YARN keeping the cluster and the database same. The wall-clock time (makespan) is the time it takes for all three queries to have returned their output and query completion time is the time taken for individual query to return its output.

At the beginning of each experiment, we wipe out the storage credits and start from zero burst credits due to two reasons: (i) Amazon SSD volumes come with 5.4 million startup burst credits which is an unrealistic burst credit balance to expect in a long running cluster. (ii) We want to explore the scenario in which VM volumes run out of burst credits, potentially leading to task slowdown, and how this threat can be addressed.

We run our experiments on three different cluster setups which differ by the number of VMs, size of storage device and database size. We do this to gauge performance of CASH under changing task characteristics due to alterations in cluster hardware and database. For each cluster, we first run the queries over stock YARN followed by CASH so as to compare their performance. The cluster setups of the three experiments are – (i) **Two VMs**, has two EC2 M5.2xlarge instances with Hive database size of 280 GB and attached SSD volume size of 200 GB per VM; (ii) **Ten VMs**, our main experiment has ten EC2 M5.2xlarge instances with Hive database size of 1.2 TB and EBS SSD volume size of 170 GB per VM and finally we test scalability through (iii) **Twenty VMs**, by running our evaluation on twenty EC2 M5.2xlarge instances with Hive database size of 2.5 TB and EBS SSD volume size of 200 GB per VM.

2) Experimental Results:

*a) **Two VMs**:* In our first experiment running CASH over 2 VMs, we observe an average improvement of about 5% in query completion time and overall wall-clock time improvement of 4.85% compared to stock YARN. The improvements are modest as the I/O requirement of the queries on a 280 GB

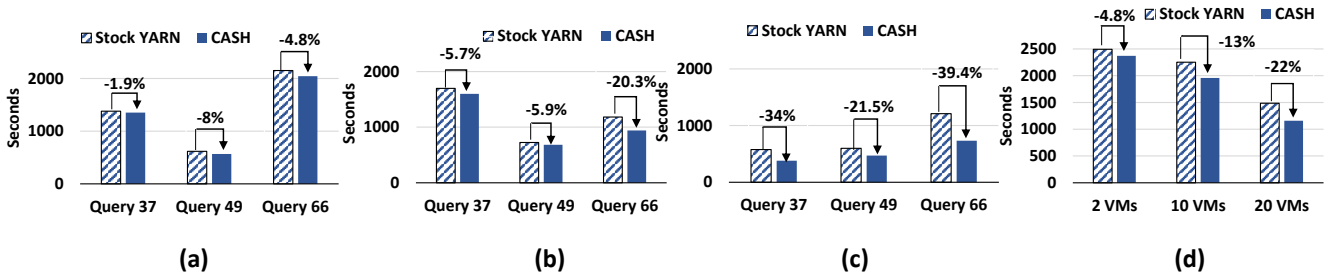

Fig. 9. Query Completion Time Comparison (a) 2 VMs, (b) 10 VMs, (c) 20 VMs; (d) Wall clock completion time for all queries

database is low and the tasks spawned for query execution needed few burst credits. Results in Figure 9 (a).

*b) **Ten VMs**:* However in our second experiment, *running CASH over 10 VMs with a 1.2 TB hive database improved average query completion time by about 10.7% and overall wall-clock time by about 13% compared to stock YARN*. Results in Figure 9 (b). An increase in the I/O requirement of the queries due to the larger size of the database and the availability of ten burstable storage volumes in the cluster (in place of two) awarded CASH more optimization opportunities. CASH was able to opportunistically schedule I/O bound tasks on VMs with higher storage burst credits leading to improved I/O throughput. This is seen in Figure 11 (a). Query 66 has a lower I/O throughput due to the query being less I/O intensive than the other two and conversely being more CPU intensive than the other two. The cluster while running CASH showed much higher average I/O utilization and a lower standard deviation of burst credits compared to stock YARN as shown in Figure 10. A lower standard deviation of burst credit balance points to better load balancing of I/O tasks in the cluster resulting in maximum burst credit utilization. We also observe a higher CPU utilization for each query (Figure 11 (b)) which is an artifact of the higher data processing throughput of the cluster.

*c) **Twenty VMs**:* We hypothesize that an increase in I/O requirement of the workload and burst credit availability in the cluster will augment CASH' benefits. In order to test this hypothesis, we ran several experiments with larger cluster sizes and we report one such experiment here. *Running CASH over a 20 VM cluster with a 2.5 TB Hive database, improved query completion time by a maximum of 39.4% and wall-clock time by 22%* as shown in Figure 9(c).

Any improvement in end-to-end wall-clock time directly translates to cost savings of equal valuation in terms of public cloud billing. Hence, our maximum wall clock time improvement of 22% directly translates to a cost savings of equal amount for tenants of the public cloud using CASH. We summarize our cost savings in Figure 9(d).

VII. Related Work

We can classify related work into two categories – Startup credit based approach and Burstable instance applications. We discuss them below.

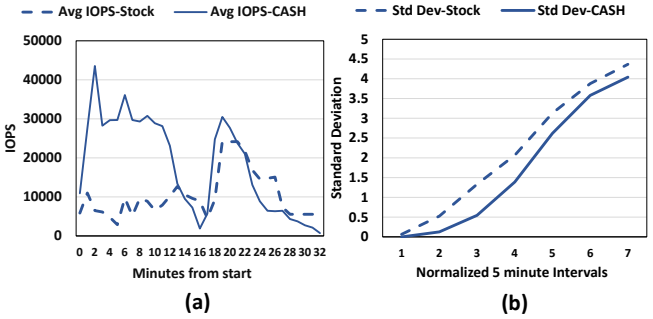

Fig. 10. (a) Avg Total IOPS, (b) Std Deviation of Burst Credits in the cluster

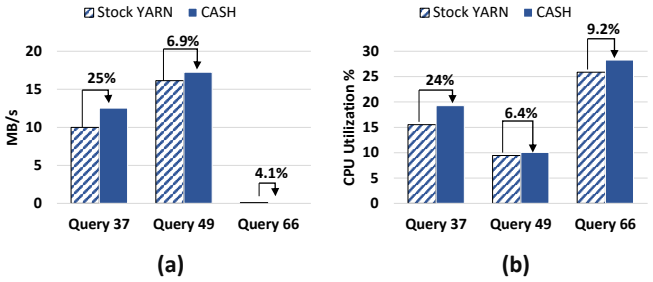

Fig. 11. (a) I/O throughput per query, (b) CPU throughput per query

1) Startup Credit based approach: Older generation AWS burstable instances (T1 and T2) came with free CPU burst "launch credits" upon instance creation. While T1 instance are no longer offered, T2 instances run on older generation hardware and are pricier than T3 instances [33]. Prior work such as [34] try to exploit the launch credits in T2 instances by delaying the launch credit consumption. However as stated in section VI, its unrealistic to rely on launch credits in a long running cluster. In [35] the authors use a similar free storage burst "launch credits" approach for AWS SSD drives. In addition to delaying the consumption of the launch credits, the authors propose to use a number of smaller volumes in place of a larger volume as the launch credits are fixed per volume. This too is impractical for a number of reasons – (i) Having multiple volumes of lower baseline service rate in place of larger volume with higher baseline rate will be detrimental once the launch credits is exhausted, (ii) Their work doesn't deal with "optimally" expending storage burst credits in a long running cluster.

2) Burstable Instance applications: Prior work such as [36] show that burstable instances can be used as a passive backup system which is also highly available. [18] characterizes the unorthodox dual token-bucket mechanism used by AWS burstables and proposes some use cases. [37] gives an empirical study on burstable instances and two more use cases. And finally [38] create a theoretical model to maximize revenue from burstable instances from a provider perspective.

VIII. Conclusions

State-of-the-art cluster schedulers do not utilize burstable hardware resources (e.g., CPU or disk I/O) efficiently in scheduling incoming tasks. Burstable resources offer a baseline service rate and a higher burst service rate depending upon their burst credit state. Tasks whose critical resource need is a burstable resource should be placed on a VM that has burst credits for the corresponding burstable resource. However, cluster schedulers are unaware of both – critical resource needs of a task and the state of burstability in their cluster. In view of this, this paper proposes a novel scheduler, CASH, for cluster VMs of a public cloud, which relies upon the knowledge of the burst credit states of the burstable resources of each VM and the estimated burst resource needs of tasks indicated through annotations. The scheduler was prototyped on YARN, Tez and Hadoop and experiments were conducted using batch and streaming workloads. With the proposed scheduler, low CPU intensive batch workloads running on T3 burstable instances were cost-effective compared to running on EMR and stock YARN. CASH was able to accelerate streaming hive queries by upto 39.4%, while saving public cloud costs by up to 22%. In our on-going work, we are experimenting with *joint* scheduling of plural credit-based resources (CPU, storage I/O and network I/O).

Acknowledgements

We are indebted to Jashwant Raj Gunasekaran for his insightful comments on several drafts of this paper. This research was partially supported by NSF grants #1763681 and #1629915. All product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] "Amazon Web Services (AWS)," https://aws.amazon.com/.
[2] "Microsoft Azure," https://azure.microsoft.com/.
[3] "Amazon Elastic Block Store: Easy to use , high perfromance block storage at any scale," https://aws.amazon.com/ebs/.
[4] "Azure Premium SSD," https://docs.microsoft.com/en-us/azure/virtual-machines/disks-typespremium-ssd.
[5] "AWS EC2 instances," https://aws.amazon.com/ec2/instance-types/.
[6] D. Jeffrey and G. Sanjay, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, Jan. 2008.
[7] "Apache Spark," https://spark.apache.org/.
[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.
[9] "Apache Tez," https://tez.apache.org/.
[10] A. Jain, A. Baarzi, N. Alfares, G. Kesidis, B. Urgaonkar, and M. Kandemir, "SplitServe: Efficient Splitting Complex Workloads across FaaS and IaaS," in *Proc. ACM/IFIP Middleware*, Dec. 2020.
[11] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly, 2016.
[12] "Kubernetes," https://kubernetes.io/.
[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proc. USENIX NSDI*, 2011.
[14] "Apache Hadoop YARN," https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html.
[15] Y. Shan, G. Kesidis, A. Jain, B. Urgaonkar, , J. Khamse-Ashari, and I. Lambadaris, "Heterogeneous MacroTasking (HeMT) for Parallel Processing in the Cloud," in *Proc. ACM/IFIP WoC)*, Dec. 2020.
[16] "Google Cloud : Cloud Computing Services," https://cloud.google.com/.
[17] "AWS T3 Instances," https://aws.amazon.com/ec2/instance-types/t3/.
[18] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis, "Using Burstable Instances in the Public Cloud: When and How?" in *Proc. ACM SIGMETRICS, Urbana-Champaign, IL*, June 2017.
[19] "Amazon EMR," https://aws.amazon.com/emr/.
[20] "Apache HIVE: Data Warehouse Software," https://hive.apache.org/.
[21] "Amazon S3 Storage Classes," https://aws.amazon.com/s3/.
[22] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *2017 IEEE Big Data*, 2017, pp. 2884–2892.
[23] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," ser. IWQoS '19. New York, NY, USA: Association for Computing Machinery, 2019.
[24] "Alibaba Trace," https://github.com/alibaba/clusterdata.
[25] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, "Workload characterization on a production hadoop cluster: A case study on taobao," in *2012 IEEE IISWC*, Nov 2012, pp. 3–13.
[26] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," ser. SoCC '12. New York, NY, USA: Association for Computing Machinery, 2012.
[27] "Nasdaq's Architecture using Amazon EMR and Amazon S3 for Ad Hoc Access to a Massive Data Set," https://aws.amazon.com/blogs/big-data/nasdaqs-architecture-using-amazon-emr-and-amazon-s3-for-ad-hoc-access-to-a-massive-data-set/.
[28] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on aws," in *2019 IEEE CLOUD*. IEEE, 2019, pp. 272–280.
[29] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," 04 2010, pp. 41 – 51.
[30] M. Poess, T. Rabl, and H.-A. Jacobsen, "Analysis of tpc-ds: The first standard benchmark for sql-based big data systems," ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 573–585.
[31] "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/, .
[32] "Hive testbench," https://github.com/hortonworks/hive-testbench.
[33] "Amazon EC2 T2 Instances: General purpose instance type that provides the ability to burst above baseline when needed," https://aws.amazon.com/ec2/instance-types/t2/.
[34] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "It's not a sprint, it's a marathon: Stretching multi-resource burstable performance in public clouds (industry track)," in *Proceedings of Middleware '19*, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 36–42.
[35] H. Park, G. R. Ganger, and G. Amvrosiadis, "More IOPS for less: Exploiting burstable storage in public clouds," in *HotCloud 20*. USENIX Association, Jul. 2020.
[36] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 620–634.
[37] P. Leitner and J. Scheuner, "Bursting with possibilities – an empirical study of credit-based bursting cloud instance types," in *2015 8th IEEE/ACM UCC*, Dec 2015, pp. 227–236.
[38] Y. Jiang, M. Shahrad, D. Wentzlaff, D. Tsang, and C. Joe-Wong, "Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization," 05 2019.