Heterogeneous MacroTasking (HeMT) for Parallel Processing in the Cloud

Yuquan Shan
School of EECS
Penn State University
University Park, PA
yxs182@psu.edu

George Kesidis
School of EECS
Penn State University
University Park, PA
gik2@psu.edu

Aman Jain
Microsoft
Redmond, WA
Aman.Jain@microsoft.com

Bhurvan Urgaonkar
School of EECS
Penn State University
University Park, PA
buu1@psu.edu

Jalal Khamse-Ashari
SCE Dept
Carleton University
Ottawa, Canada
jalalkhamseashari@sce.carleton.ca

Ioannis Lambadaris
SCE Dept
Carleton University
Ottawa, Canada
ioannis@sce.carleton.ca

ACM Reference Format:
Yuquan Shan, George Kesidis, Aman Jain, Bhurvan Urgaonkar, Jalal Khamse-Ashari, and Ioannis Lambadaris. 2020. Heterogeneous MacroTasking (HeMT) for Parallel Processing in the Cloud. In *Containers Workshop on Container Technologies and Container Clouds (WOC '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3429885.3429962

Abstract — Using tiny tasks (microtasks) has long been regarded an effective way of load balancing in parallel computing systems. When combined with containerized execution nodes pulling in work upon becoming idle, microtasking has the desirable property of automatically adapting its load distribution to the processing capacities of participating nodes - more powerful nodes finish their work sooner and, therefore, pull in additional work faster. As a result, microtasking is deemed especially desirable in settings with heterogeneous processing capacities and poorly characterized workloads. However, microtasking does have additional scheduling and I/O overheads that may make it costly in some scenarios. Moreover, the optimal task size generally needs to be learned. We herein study an alternative load balancing scheme - Heterogeneous MacroTasking (HeMT) - wherein workload is intentionally *skewed* according to the nodes' processing capacity. We implemented and open-sourced a prototype of HeMT within the Apache Spark application framework and conducted experiments using the Apache Mesos cluster manager. It's shown experimentally that when workload-specific estimates of nodes' processing capacities are learned, Spark with HeMT offers up to 10% shorter average completion times for realistic, multistage data-processing workloads over the baseline Homogeneous microTasking (HomT) system.

WOC '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8209-0/20/12…$15.00
https://doi.org/10.1145/3429885.3429962

1 Introduction

Parallel data processing represents a large and important class of workloads running on public cloud computing platforms. Load balancing - dividing work among the execution nodes of a cluster - has an important role in determining the performance of these workloads. As needed by an application framework, execution nodes are allocated as provisioned containers by a cluster manager controlling the Virtual Machines (VMs) that make up the cluster[1]. The nodes are often *heterogeneous* in terms of their processing capacities. Execution node hetergeneity may have to do with how the cluster manager allocates currently unreserved resources from the VMs when an application framework has a new job to run. For example, based on currently available resources in the VMs, Apache Mesos may offer containers of different sizes (provisioned resources for task execution) to a given application frameworks it is supporting, e.g., [6, 11, 13]. Also, lower-cost VMs themselves may exhibit variation over time in their capacity to execute jobs, e.g., [19, 20].

Such node heterogeneity may have negative implications for workload performance (such as completion

[1] Alternatively or in addition, executors may be cloud/Lambda functions procured directly from the cloud.

times) and cost incurred by the workload owner or "tenant" (due to resource wastage in under-utilized nodes). In particular, it may exacerbate the well-known "straggler problem" [1], wherein a small subset of slow tasks stall an entire (parallel computed) job stage by causing a synchronization delay at a program barrier (i.e., *all* parallel tasks need to complete before the program can proceed, so the slowest task(s) determine the execution time of the job stage).

[9] advocates that parallel jobs should be divided into relatively small-sized tasks ("microtasks") via fine homogeneous partitioning of the input dataset on which processing is being performed[2]. Microtasking can lead to good load balancing when combined with a "pull-based" operation: when underbooked or idle, nodes pull work (tasks) from a pending queue so faster workers simply pull in more work. Furthermore, the synchronization delays are reduced without needing knowledge of either the speed of the nodes or the resources required to achieve particular execution-times for the tasks. So, we refer to such approaches as *oblivious* load balancing. However, there also exist studies, e.g., [17], that challenge the microtasking idea, pointing out that the relatively large overhead of microtasking can, in some cases, significantly slow down computation.

Although Homogeneous microTasking (HomT) can provide certain qualities-of-service and load-balancing efficiencies without detailed information about the cluster or workload [3, 5, 9], its usefulness may be hindered by additional processing and disk-IO overhead [8, 9, 22]. Also, optimal microtask sizes need to be learned, cf. the HomT curves of Figs. 5 and 6.

The contribution of this paper are:

- We consider variants of "skewed" or Heterogeneous MacroTasking (HeMT) corresponding to different degrees of accuracy/certainty in supply or demand characterization ranging from an oblivious, incrementally adjusted HeMT to a more sophisticated version where offline/online knowledge of node capacities is also leveraged.
- Using a variety of Amazon EC2 experiments on our open-source Spark over Mesos prototype [7, 14] with different workloads and nodes (regular or burstable EC2 instances), we show the efficacy of HeMT over HomT. Representative experiments described herein employ two important multistage workloads: PageRank and K-Means.
- We identify and suggest interesting directions for future work, especially related to adaptive scheduling across the application frameworks and cluster manager layers, including improved information exchange between them via enhanced APIs.

This rest of this paper is organized as follows. Heterogeneous macrotasking is described in Sec. 2. In Sec. 3, we study a simple "oblivious" approach to online adapting the size of heterogeneous macrotasks based on synchronization delays (variations in task execution times) at program barriers. HeMT for multistage workloads (K-Means and Pagerank over MapReduce) is descibed in Sec. 4. (HeMT for simpler workloads on statically provisioned containers or on burstable instances is described in [12].) We conclude with a brief summary and discussion of future work in Sec. 5.

2 Heterogeneous Macrotasking (HeMT) - Background

To avoid HomT overhead, the number of tasks can be set equal to the number of available "computation slots" (available containerized executors). However, in case of heterogeneous executors, synchronization delay may ensue if such "macrotasks" are equally sized. This motivates skewed or Heterogeneous MacroTasking (HeMT).

HeMT will require a reasonably accurate estimation of workload (reflected by task execution time) which can be easily obtained for many modern jobs due to their repetitive nature; e.g., many production workloads [10] and machine-learning related jobs such EM and K-Means [2] that consist of multiple iterations of the same computational complexity. Much recent work on task scheduling, e.g., [21], is based on such an assumption.

We implemented this HeMT partitioning algorithm on Spark and compared it with Spark's default partitioning scheme in the following, as well as the aforementioned HomT. Spark's default partitioning does not consider any resource heterogeneity of the cluster - it divides the input data regardless of the speed of computing nodes - and Spark tends to evenly divide on-memory data into as many partitions as the number of computing slots (usually processing cores), i.e., homogeneous macrotasking. For files located on disk, e.g. HDFS files, baseline Spark, like Hadoop, assigns one file block to a task. Spark naturally supports HomT: users can specify a desired number of partitions and Spark would evenly divide data according to this number.

The aim of this experimental study is to illustrate the benefits and challenges of HeMT. We implemented HeMT in Spark [14] (for an arbitrary number of executors) using information from middleware (here, Apache Mesos cluster manager [7]) or directly from monitoring services (e.g., AWS CloudWatch). For scalability, the application frameworks perform most elements of

[2]Specifically, [9] suggests microtasks take on the order of 100 ms to execute on contemporary systems.

(workload specific) HeMT learning, while the middleware scheduler may only perform more sophisticated workload scheduling (consolidation)[3]. The information exchange in our Spark-Mesos prototype is summarized in Figure 1.

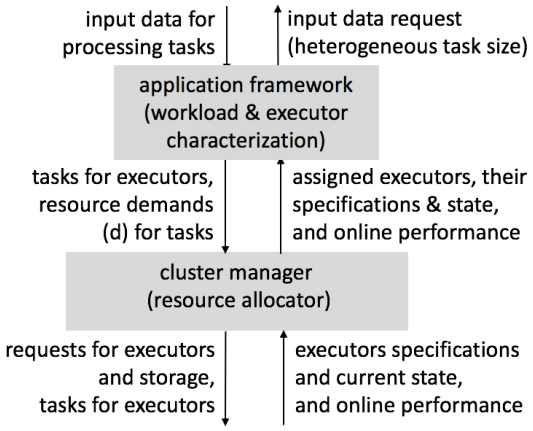

Figure 1. Overview of proposed modifications to application frameworks and cluster manager.

3 Oblivious Adapted HeMT (OA-HeMT)

In some environments, e.g., those without resource isolation leading to significant interprocess interference, determining the true workload processing power of available computational nodes may be challenging. So, a simple "oblivious" approach is needed to allow application frameworks or cluster managers to dynamically estimate the processing speed of available computational nodes according to the previous workloads of the same job, so that the future tasking can be well balanced.

A Spark-Mesos prototype was implemented to enable such an oblivious ad-hoc adaptive HeMT. The Mesos cluster manager obtains and passes on to the Spark application framework estimated executor processing speed through additional fields in their RPC messaging. Based on this information and the associated task sizes, Spark estimates the execution speed of different available executors and thereby determines how to partition future work into well-balanced tasks.

3.1 Different executors of the same task type

In this paper, we model task execution times simply as linear in the size of the input dataset to be processed (certainly, more complex models can be learned instead considering other important factors, e.g., data locality).

[3]Analogies can be made with "end-to-end" approaches such as exokernels or TCP congestion control in the Internet.

Consider a sequence of datasets of sizes $\{D_k\}$ that need to be processed in the same way, i.e., the same job applied to each dataset. The k^{th} dataset D_k is divided (by the application framework) into a number of tasks, one for each executor $i \in L_k$ assigned to process the k^{th} dataset D_k (by the cluster manager). These tasks are created by dividing the dataset D_k.

For each executor $i \in L_k$, let v_i be the most recent estimate of its "speed" for the job under consideration. Let $L_k^o \subset L_k$ be the set of executors that have not before been assigned to this job. For all $i \in L_k^o$, let $v_i = \overline{v}$ where $\overline{v}$ is the average v_j for $j \in L_k \backslash L_k^o$ (example other choices could be the minimum or maximum rather than the average or the average speed over all executors that have been applied to this job in the past). Let

$$V_k \;=\; \sum_{i \in L_k} v_i \;=\; |L_k|\overline{v},$$

where $|L_k|$ is the number of executors assigned to the k^{th} job. Executor $i \in L_k$ is assigned a dataset of size $d_i = D_k v_i / V_k$. That is, the faster executor (larger v_i) is assigned to work on a larger dataset (larger d).

Let t_i be the execution time of executor $i \in L_k$ on the assigned task of size d_i of the k^{th} job. For all executors $i \in L_k$, their speed can be updated, for example, according to a simple first-order autoregressive estimator

$$v_i \;\leftarrow\; \alpha \frac{d_i}{t_i} + (1-\alpha) v_i$$

where forgetting factor $1-\alpha$ is such that $0 < \alpha < 1$. The straightforward tradeoff in the choice of α is that smaller α means that the speed estimate is less responsive to the latest datapoint d_i/t_i, but will result in less oscillation/overshoot in a dynamic setting where resources and workload often change.

For the initial ($k = 1$) job, D_1 is evenly divided among the executors $i \in L_1$ and subsequently $v_i = d_i/t_i$.

It's entirely possible that different datasets of the *same size*, i.e., $d = d'$, will require different execution times $t \neq t'$ for the same job type under consideration. Over time, we expect that such variations will be "averaged out" in the executor speed estimates; i.e., each executor will experience the same task-difficulty distribution "per unit" input data (unless there is some bias so that some executors tend to receive more difficult tasks per unit input data for a given job). This motivates a updating factor α that this not close to zero.

Note that different application frameworks (different job types) will need to maintain their own estimates of (workload specific) executor speeds.

3.2 An experimental result

To see the effect of such adaptive workload partitioning, we performed an experiment with a two worker-node

cluster, where each node is an AWS m5.large VM with two vCPUs. No resource isolation technology was used, so Spark executors could share CPU cycles with other processes. A sequence of fifty Spark WordCount jobs were presented through a submission queue. We introduced two CPU-interfering processes [15] on one node at one point in time (because there are two cores per node), and then on the other node at a later point in time, thus reducing the processing speed of Spark executors on those nodes. How Spark jobs adaptively partitioned to re-balance their workloads with $\alpha = 0.5$ is shown in Fig. 2.

One can see how overall job execution times (determined by the slowest task) increased dramatically but then fell as our modified Spark learns executors' speed from previous trials (here, for a given executor, execution-time variation per unit document was low).

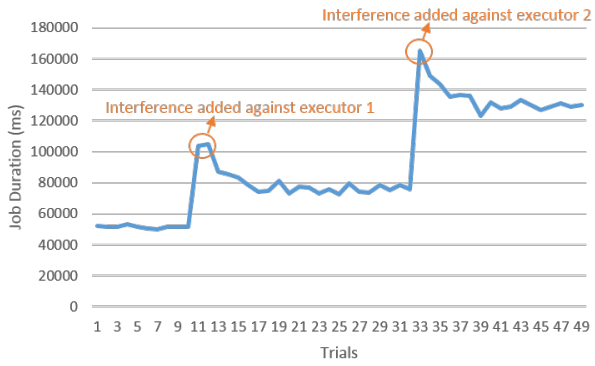

Figure 2. Adaptive workload balancing with introduced interfering processes at two points in time ($\alpha = 0.5$).

We performed another experiment involving two hosts being statically provisioned with one and 0.4 cores[4], i.e., heterogeneous executors by *initial* provisioning. The results with different forgetting factors are shown in Fig. 3. Spark learns the optimal way of partitioning the workload after initial trials. As expected, when executors' capacity does not change, larger α results in faster adaptation to the heterogeneity. Eventually the map-stage execution time is reduced to around 60 seconds, which is in agreement with the results shown in [12] for provisioned containers where a near-optimal data partitioning can be simply derived a priori using resource-allocation information provided by Mesos.

This online adaptive task sizing can also be applied to the provisioned containers, lambda functions, and burstable instances, where the computation capacities of the nodes can be estimated and quantified *a priori*,

[4]See [12] for HeMT experimental results with provisioned containers and burstable VMs.

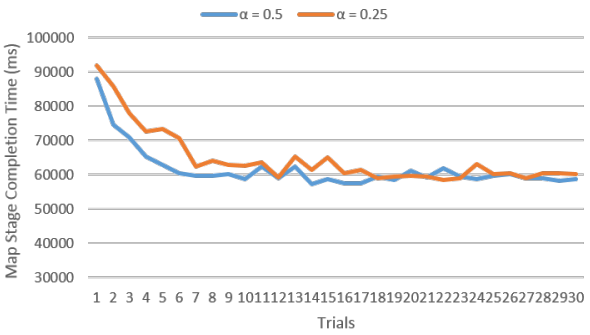

Figure 3. Workload re-balance, with α equal to 0.5 and 0.25 respectively, when executors are different by initial provisioning.

and can be further fine-tuned online to achieve better performance, again see Section 4 below.

4 HeMT - repartitioning on multiple computation stages

A typical MapReduce workload consists of one or more jobs, each job has multiple basic computation stages (including "hard" parallel-program barriers/synchronization-points) presented in the previous sections concatenated together through data shuffling. So for the first computation stage, we can simply divide the initial input data according to the computation capacities of the executors.

A partitioner defines how a task assigns its intermediate results to different "buckets" which will be fetched by different tasks in the following stage. For the following stages, task data are fetched from the intermediate outputs of the tasks in the previous stages. The tasks in the previous stages first shuffle the processed records into different buckets (each corresponding to one fetching task in a future stage) according to a partitioner function, then those buckets are written onto storage media for associated future tasks to fetch. The default hash partitioner shuffles those records into those buckets in a statistically even fashion. So, we need to define a new partitioner that can skew the shuffle buckets for HeMT. For concreteness, we give one simple implementation of skewing using hash code in Algorithm 1[5].

The comparison of effective data flows when using the default hash partitioner and our skewed hash partitioner respectively is shown in Fig. 4. The relevant idea of balancing workload through partitioner can be found in [3, 18].

[5]Certainly, a more sophisticated partitioning algorithm can be made given more information regarding key distribution and processing complexity of each record.

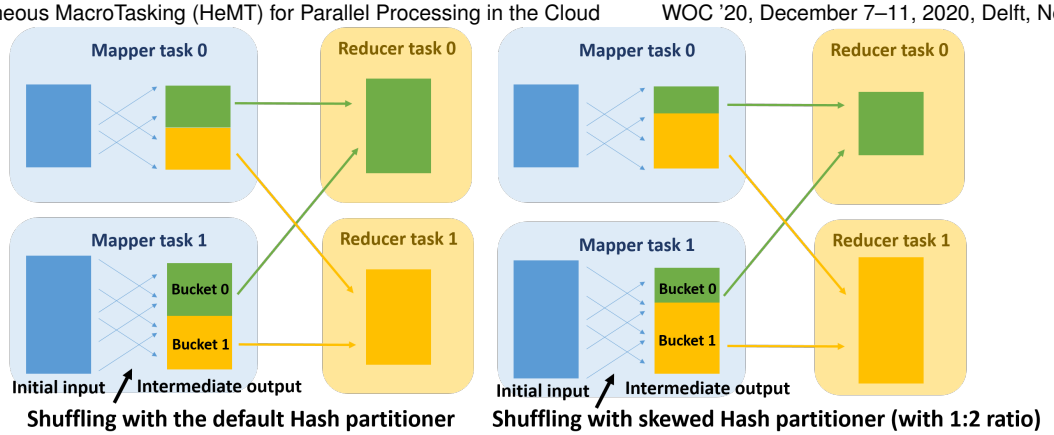

Figure 4. Data flows in even hash shuffling and skewed hash shuffling respectively.

Algorithm 1: Partitioning function of skewed hash partitioner

Data: Record r to be assigned to a bucket; array of executors' computation capacities, *executors*
Result: The index of the target bucket
$sum = 0$;
for *e from* 0 *to executors.length* **do**
  sum += *executors*[e];
  executors[e] = sum;
hash = r.hashCode mod *executors*.sum;
return the number of elements in *executors* greater than or equal to *hash*.

We present the performance of HeMT using two typical workloads - K-Means and PageRank. Those two have different and representative computation patterns. K-Means consists of repetitive simple two-stage Spark jobs. PageRank, on the other hand, is a single Spark job containing multiple computation stages concatenated together through shuffling.

We ran K-Means on the cluster with two executors hosted on two containers, one was allocated with one CPU core, the other was allocated with 0.4 cores. To make results more consistent, instead of setting a convergence criterion to stop the iterations, we fixed the number of iterations to 30. The input source was 256 MB data file on HDFS, with block size 128 MB (so there are two blocks). The complete-job execution times of HeMT and homogeneous task-sizing (including HomT) are shown in Fig. 5 (consistent with the results for single-stage workloads [12]). Again, for homogeneous task sizing (the Spark default), execution times are longer: due to "synchronization" delays (because the executors have different capacities) when there are fewer, larger tasks, and due to I/O overhead when there are more, smaller tasks.

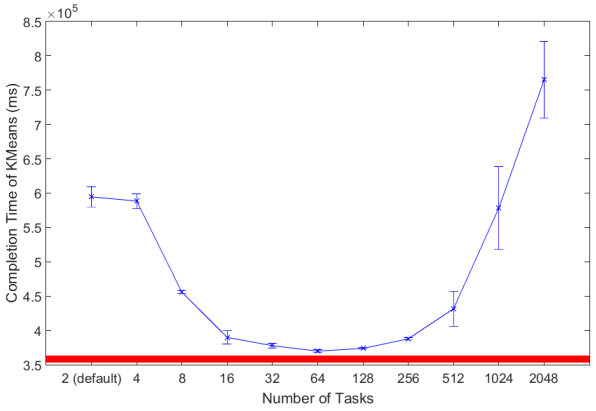

Figure 5. K-Means execution times with 95% confidence bars.

On the same cluster, we run PageRank with 256 MB input data for 100 iterations. The results are shown in Fig. 6. Note that the PageRank, compared with K-Means, is more sensitive to microtasking, because each iteration of PageRank is relatively short (around 10s in the default 2-way partitioning), therefore each task is shorter as well. For example, if we use 64-way partitioning, then each task generally lasts for only 0.1 - 0.2 seconds. Therefore, the relative task scheduling overhead would be larger for the PageRank workload.

Figs. 5 and 6 show HeMT outperforms homogeneous tasking. The optimal homogeneous task-size is *application specific* and its choice can have significant performance consequences, so it also needs to be learned.

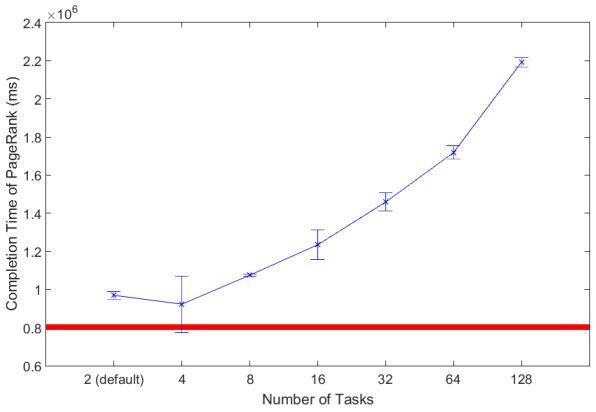

Figure 6. PageRank executing times with 95% confidence bars.

5 Summary and Future Work

In summary, we investigated the pros and cons of heterogeneous macrotasking (HeMT) in large-scale parallel processing workloads that routinely run on modern public cloud platforms. We implemented an open-source prototype of HeMT within the Apache Spark application framework with complementary changes to the Apache Mesos cluster manager [7, 14]. Our experimental results, typical representatives of which were reported above, showed that HeMT outperformed HomT when accurate workload-specific estimates of nodes' processing capacities could be learned. In our representative multistage-workload experiments, Spark with HeMT was able to improve average job completion times by about 10% compared to the default system.

In future work, we will consider application frameworks and middleware embodying more advanced, integrated online learning frameworks that leverage information from offline workload profiling [16] and service-level agreements to more precisely (online) characterize workloads' resource needs (demand) and executors' capacity (supply). Actions by different application frameworks based on such learning include HeMT at a fast timescale and determination of preferred types of executors based on cost/performance tradeoffs. For a budget-conscious tenant, we also plan to integrate such actions by application frameworks with scheduling by the cluster manager, i.e., [4] and more efficient, server-specific alternatives [11], based on *sizing* executors according to workload characterizations and considering data-locality constraints too. That is, the cluster manager's scheduler would consider estimates of the resource needs of tasks of its application frameworks (perhaps as a function of input dataset size) in order to obtain adequate performance (scheduling in "fine grain" mode).

Acknowledgement:

This research was supported in part by NSF CNS 1717571 grant and a Cisco Systems URP gift.

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. 2010. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proc. USENIX OSDI*.

[2] R.O. Duda, P.E. Hart, and D.G. Stork. 2001. *Pattern Classification, 2nd Ed.* Wiley.

[3] Y. Fan, W. Wu, D. Qian, Y. Xu, and W. Wei. 2013. Load Balancing in Heterogeneous MapReduce Environments. In *Proc. IEEE HPCC & EUC*.

[4] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. USENIX NSDI*.

[5] R. Li, P. Guo, B. Hu, and W. Hu. Nov. 2019. Libra and the Art of Task Sizing in Big-Data Analytic Systems. In *Proc. ACM SoCC*.

[6] Mesos [n. d.]. Apache Mesos - Containerizers. http://mesos.apache.org/documentation/ latest/containerizer-internals/.

[7] Mesos [n. d.]. Mesos multi-scheduler. https://github.com/PSU-Cloud/mesos-ps/pull/1/files.

[8] E.B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. 2012. Flat Datacenter Storage. In *Proc. USENIX OSDI*.

[9] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. 2013. The Case for Tiny Tasks in Compute Clusters. In *Proc. USENIX HotOS*.

[10] A. Pucher. [n. d.]. Qubole Announces Apache Spark on AWS Lambda. https://alexpucher.com/blog/2015/06/29/cloud-traces-and-production-workloads-for-your-research/.

[11] Y. Shan, A. Jain, G. Kesidis, B. Urgaonkar, J. Khamse-Ashari, and I. Lambadaris. Sept. 2018. Scheduling distributed resources in heterogeneous private clouds. In *Proc. IEEE MASCOTS*.

[12] Y. Shan, G. Kesidis, B. Urgaonkar, J. Schad, J. Khamse-Ashari, and I. Lambadaris. Oct. 2018; https://github.com/PSU-Cloud/spark-hemt. Heterogeneous MacroTasking (HeMT) for Parallel Processing in the Public Cloud. https://arxiv.org/abs/1810.00988.

[13] Spark [n. d.]. Apache Spark - Running Spark on Mesos. https://spark.apache.org/docs/latest/running-on-mesos.html.

[14] Spark [n. d.]. Spark with HeMT. https://github.com/PSU-Cloud/spark-hemt/pull/2/files.

[15] sysbench 2016. https://github.com/akopytov/sysbench.

[16] M. Tirmazi, A. Barker, N. Deng, M.E. Haque, Z.G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. 2020. Borg: the next generation. In *Proc. ACM EuroSys*.

[17] E. Totoni, S.R. Dulloor, and A. Roy. 2017. A Case Against Tiny Tasks in Iterative Analytics. In *Proc. HotOS*.

[18] R. Vernica, A. Balmin, K.S. Beyer, and V. Ercegovac. 2012. Adaptive MapReduce Using Situation-aware Mappers. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)*.

[19] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang. 2017. Combining spot and on-demand instances for cost effective caching. In *Proc. ACM EuroSys*. Belgrade.

[20] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. June 2017. Using Burstable Instances in the Public Cloud: When and How?. In *Proc. ACM SIGMETRICS, Urbana-Champaign, IL*.

[21] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY,

USA, 484–496. https://doi.org/10.1145/2987550.2987576

[22] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M.J. Freedman. 2018. Riffle: Optimized Shuffle Service for Large-scale Data Analytics. In *Proc. EuroSys*.