An Empirical Analysis of Scheduling Techniques for Real-time Cloud-based Data Processing

Linh T.X. Phan Zhuoyao Zhang Qi Zheng Boon Thau Loo Insup Lee
University of Pennsylvania
{linhphan, zhuoyao, qzheng, boonloo, lee}@cis.upenn.edu

***Abstract*—In this paper, we explore the challenges and needs of current cloud infrastructures, to better support cloud-based data-intensive applications that are not only latency-sensitive but also require strong *timing guarantees*. These applications have strict deadlines (e.g., to perform time-dependent mission critical tasks or to complete real-time control decisions using a human-in-the-loop), and deadline misses are undesirable. To highlight the challenges in this space, we provide a case study of the online scheduling of MapReduce jobs executed by Hadoop. Our evaluations on Amazon EC2 show that the existing Hadoop scheduler is ill-equipped to handle jobs with deadlines. However, by adapting existing multiprocessor scheduling techniques for the cloud environment, we observe significant performance improvements in minimizing missed deadlines and tardiness. Based on our case study, we discuss a range of challenges in this domain posed by virtualization and scale, and propose our research agenda centered around the application of advanced real-time scheduling techniques in the cloud environment.**

I. Introduction

In recent years, there has been an emergence of several data-parallel middleware platforms primarily targeted towards large-scale data processing in the cloud environment, for example, MapReduce [18], Pig [35], Dryad [26] and DryadLINQ [45]. These middleware platforms provide simple yet powerful and extremely versatile programming models, and can be applied to a wide spectrum of domains where large-scale data analytics is performed. Unlike traditional Database-as-a-Service [2], [16] models for latency-sensitive user queries, these analytic models are typically used for data processing in batch mode, where high throughput and high resource utilization are key concerns.

Recent trends and uses of cloud data analytics have moved beyond this "batch processing only" mode. MapReduce applications are now becoming increasingly latency-sensitive, operating under demanding workloads that require fast response times for data-intensive computations under high data rates. These include online log processing [5], traffic simulation [42], personalized recommendations [17], advertisement placement [10], social network analysis [6], real-time web indexing [20], and continuous web data analysis [12], among others.

Beyond latency-sensitivity, we pose the following intriguing research challenge: *Is it possible to develop a cloud infrastructure that not only supports latency-sensitive and large-scale data processing but also provides strong timing guarantees? If so, what types of applications can we envision on such a cloud?* We argue that current applications are slowly but surely evolving in that direction. These emerging applications include interactive OLAP queries [9], network traffic analysis [30], flight control [13], military warfighter coordination [14], and real-time actuation of physical devices [24] based on sensor feeds. These applications typically require deployment at a large-scale to process data from multiple locations, and increasingly can benefit from having timing guarantees (e.g., to perform time-dependent mission critical tasks or to complete real-time control decisions using a human-in-the-loop). In the long term, we predict that the continued evolution of cloud applications and the scale of cloud buildout will ultimately challenge the cloud's ability to provide timing guarantees.

To explore some initial answers to our broad challenge above, we provide a case study of Hadoop's implementation of MapReduce, to understand its ability to meet timing guarantees. Specifically, given a stream of Hadoop jobs continuously submitted to a cloud cluster, where each job has a number of tasks and a deadline, we aim to schedule them on the cluster such that a particular real-time objective is achieved. These objectives can be expressed in terms of minimizing the (i) miss rate, the fraction of the applications that miss their deadlines, and (ii) the maximum (total) tardiness, the maximum (total) elapse time from deadline to completion time of the applications that miss their deadlines, or a combination of various time-dependent objectives.

We focus on MapReduce as our first case study, because its popularity and well-understood execution model provide a useful starting point for further exploration. We perform a series of performance benchmarks using Hadoop and Pig queries (compiled into Hadoop programs) on Amazon EC2. Our results show that the current Hadoop schedulers are ill-equipped to handle MapReduce jobs with deadlines as they incur high miss rates and large tardiness even under moderate loads. To explore possible next steps, we adapt existing multiprocessor scheduling techniques (e.g., Earliest Deadline First (EDF) [31]) from the real-time systems domain to the cloud setting. Our enhancements address cloud-specific issues not addressed by traditional multiprocessor scheduling techniques, such as data placement, data distribution (skews), online (potentially bursty) arrivals of jobs and data, and the inherent precedence relationships among MapReduce tasks.

We have developed a prototype *HadoopRT* system, which enhances Hadoop [22] with various real-time scheduling policies. Using Amazon EC2, we perform an extensive evaluation of a variety of Hadoop and Pig applications. Our evaluation results show that our proposed enhancements of EDF are sig-

nificantly more effective than the default Hadoop schedulers. Their relative performances depend on a variety of system load conditions and scheduling metrics.

Based on the insights obtained from our evaluation, we conclude with a broad discussion of challenges towards the vision of a real-time cloud, centered around the themes of advanced real-time scheduling techniques (probabilistic, hierarchical, and multi-modal).

II. Related Work

Scheduling theory for cloud computing has received growing attention over the past years. For instance, Zaharia et al. [48] developed the LATE scheduling algorithm that executes duplicates of speculative slow jobs on fast nodes to improve their response times. Agrawal et al. [1] improved Hadoop's performance by modifying the Hadoop's scheduler to favor shared scans between jobs that read from the same inputs. Fairness scheduling has been explored in a number of frameworks such as the default Hadoop's fair-sharing scheduling [22], the Quincy scheduling [25], and scheduling techniques based on delay scheduling [47] and copy-compute splitting [46]. Grid-based scheduling techniques have also been adopted into the cloud setting [34]. Price models and cost-based resource provision techniques have also been studied (e.g., [19], [23], [32]). All these techniques, however, targeted non real-time applications where the primary objectives are to increase throughput and minimize average response time instead of meeting deadlines. Scheduling of data-intensive applications has been widely explored in the field of transaction scheduling in real-time databases (e.g., [28]). However, these techniques require fine-grain information such as transactions' data access patterns and periodization, which are often not available for general jobs running in the cloud environment.

In our technical report [36], we first introduce deadline-based scheduling metrics for hard and soft real-time MapReduce jobs. We perform an experimental study on Amazon EC2 to understand the extent to which the completion time of a MapReduce job is predictable, and the factors that are important to its predictability. We also outline a number of online scheduling heuristics, which are used as a basis for our MapReduce case study in this paper. [43] proposed the FLEX slot allocation, which considers timeliness as a metric for Hadoop. While improving upon the default Hadoop's fair scheduler, the scheduling techniques used in FLEX assume restricted settings such as homogeneous CPU resources and negligible data communication delay, which are unlikely to hold true in the deployment scenarios for our target applications in Section I. Verma et al. [41] proposes a scheduling algorithm that combines EDF with fair scheduling. Our motivating case study also has an EDF variant that enhances EDF with locality-awareness and overload handing; however, the focus of our case study, and of the paper as a whole, are on broader issues related to the challenge statement in Section I, rather than the specifics of one particular policy. Herein, we also significantly improve upon the experimental methodology compared to [41], with extensive evaluations on an actual cloud platform (Amazon EC2) with detailed comparisons against state-of-the-art Hadoop schedulers (in the presence of data skews and communication delays).

An important area of related work is scheduling theory for real-time systems. Many well-known theories have been established in this area, with a growing focus on multiprocessor (e.g., [3], [4]) and virtualized platforms (e.g., [8], [11], [15], [29], [39], [40]). The results established in this area are promising foundation for our proposed research. However, as they are based on relatively simple task models and focus primarily on computing resources, several adaptations and extensions are necessary to adapt them to the cloud setting and data-intensive applications such as MapReduce.

III. Case Study: MapReduce

We present a case study of MapReduce to give some insights into the challenges faced in providing stronger timing guarantees for cloud data processing. We first formulate the online scheduling of soft real-time MapReduce jobs on a cloud cluster that runs Hadoop middleware. We then propose our cloud adaptations of the well-known Earliest Deadline First (EDF) scheduling policy as solutions and present evaluation results highlighting their effectiveness.

A. Problem Formulation

The cluster consists of M connected *slave* nodes on which jobs are scheduled and a *master* node on which the scheduler resides. Each slave node is a multicore processor, which is configured to have a number of slots for executing tasks. Each job submitted to the cluster consists of a set of independent map tasks, followed by a set of independent reduce tasks. Each map task processes an input data block, which consists of a number of (key, value) tuples that are stored at one of the slave nodes (called data node of the task). Each reduce task (containing also shuffle and sort phases) computes final results from the output data of all the map tasks.

Parameter	Description
M	Number of slave nodes
c_i	Number of cores of node i
s_i	Number of slots of node i
rate_{ij}	Data rate (bytes/s) between nodes i and j
m_J	Number of map tasks of job J
r_J	Number of reduce tasks of job J
map_J	The set of map tasks of job J
reduce_J	The set of reduce tasks
D_J	Relative deadline of job J
d_J	Absolute deadline of job J
tsize	Data size (bytes) of a tuple
$\mathsf{ntuples}(T)$	Number of input tuples of task T
$\mathsf{dnode}(T)$	The data node of task T
$u_i(T)$	Unit WCET (per tuple) of task T on node i
$E_i(T)$	Estimated WCET of task T on slave i

TABLE I
JOB AND PLATFORM PARAMETERS.

Table I summarizes our terminology. Here, $\mathsf{ntuples}(T)$ (for reduce tasks) and $u_i(T)$ can be obtained by runtime measurements. $E_i(T)$ can be computed based on its unit worst-case execution time (WCET), number of input tuples, tuple size and data transfer rate. The WCET of a task T on a node i is defined as the maximum amount of time required to execute T on node i, assuming there is no contention of CPU resource, plus the input data transfer time if the input data of T is not on node i. We describe in details how these values are obtained in Section III-E.

We say that a task is *local* on its data node (which stores its input data) and *remote* to all other nodes. Reduce tasks are considered as remote on all nodes since their data nodes are unknown in advance (i.e., where the map tasks are scheduled). In our case study, every task is non-preemptable and shares the same deadline and release time as its job's. As a first step, we consider no failures and no pipelining/speculative execution.

Given the above system model, jobs are submitted continuously to the cluster, and the master scheduler needs to schedule their tasks on the slaves without a priori knowledge of future job arrivals. Our goal is to design a scheduling algorithm for the scheduler to minimize one of the following metrics: (1) the miss rate, (2) the maximum tardiness of the jobs that miss their deadlines, and (3) the total tardiness of all jobs that miss their deadlines. These are standard metrics that are used to evaluate the timing performance of soft real-time applications [33].

B. Existing Scheduling Policies in Hadoop

In the current Hadoop implementation, scheduling of MapReduce jobs proceeds as follows. Periodically, slave nodes inform the master about the availability of free map/reduce slots. The master will then select a pending job to execute on a free slot based on a *scheduling policy*, which can be one of the following:

FIFO. The scheduler maintains a FIFO waiting queue of jobs, sorted by their arrival times. Whenever a slot becomes available, the master selects the task at the head of the queue for execution.

Fair. The scheduler selects a task of a job whose utilized resource is farthest from its *fair share* for execution on the available slot.

Capacity [38]. The scheduler maintains multiple job queues, each containing jobs submitted by an organization (or group within), sorted by job arrival times. Each queue is given a fixed capacity in terms of the maximum number of slots it can use, based on organizational resource needs. Given a free slot, the scheduler schedules the first task in the queue that has the most free space (i.e., the lowest ratio of the number of running slots to queue capacity).

Once a job is selected, the scheduler will execute a task of the selected job that has input data closest to the free slot.

C. Basic Real-Time Scheduling Ideas

As observed in the previous section, none of the existing scheduling policies in Hadoop considers timeliness of jobs. In this section, we present a real-time variant for the Hadoop master scheduler. Our scheduler is based on EDF but adapted to the cloud execution platform and the MapReduce data-intensive characteristics. Intuitively, it implements two high-level ideas:

Data locality-aware. Like conventional EDF, the scheduler schedules tasks with earlier deadlines first (i.e., more urgent ones). However, to minimize data transfer overheads, it aims to schedule tasks on or close to the node where the input data are located. Here, the distance between a task and a node is measured in terms of the ratio between the task's WCET on the node (which may include input data transfer time) to the task's WCET on its data node. Hence, the scheduler may schedule for execution a less urgent but local task (on the node with a free slot), and delay the more urgent but remote tasks until future, so that it can schedule them on their data nodes when those nodes are free. Effectively, our enhancement balances the tradeoff between jobs' urgencies and data transfer overheads.

Overload handling. It is known from the real-time scheduling theory that when using EDF-based schedulers, anomaly may occur under heavy load, e.g., one task missing deadline causes a chain of subsequent deadline misses. When the system is under a heavy load condition, some pending jobs may still miss their deadlines even if we allocate all the available resources to them. If the scheduling objective is to minimize the miss rate, scheduling those jobs immediately is not useful (because they are going to miss their deadlines anyway) and may even cause other potentially schedulable jobs to miss their deadlines (as in the anomaly example). As such, our scheduler schedules jobs that are potentially schedulable first, and only schedules the ones that are predicted to miss deadlines when there are no more potentially schedulable jobs.

D. Concrete Scheduling Implementations

We now detail two concrete policies for selecting a task to execute on a free slot that implement the basic ideas discussed in the previous section. The first, EDF/TD, is used when the scheduling objective is to minimize total/maximum tardiness. The second, EDF/MR, is used when the scheduling objective is to minimize miss rate. Both policies are data locality-aware, and EDF/MR further implements overload handling.

EDF/TD scheduling policy. Algorithm 1 gives the pseudo-code for the EDF/TD scheduling policy. The function $\mathsf{edftd}(\mathsf{jobs}, i, w)$ computes as output a task to be executed on the free slot on node i. jobs is the queue of unfinished jobs, sorted by their absolute deadlines. w is a configurable constant weight factor of the scheduler, which indicates how important data locality is in making scheduling decisions. The smaller w is, the shorter the distance that the scheduler enforces between the data node of a task and the node where it is executed. In this paper, we determine w experimentally for our evaluation; however, we plan to investigate the quantitative relationships between w and the network condition as well as the application characteristics in future extensions. The set $\mathsf{localTasks}$ contains all ready tasks that are local to node i. As was mentioned earlier, Reduce tasks are considered as remote

to all nodes; hence, this set contains only Map tasks. The set nearTasks contains tasks that are close to node i, with respect to the weight factor w. Specifically, a task T is considered to be close to node i if its estimated WCET on node i (denoted by $E_i(T)$) is less than the minimum estimated execution time (denoted by $E_{\min}(T)$) times the weight factor w. Here, $E_{\min}(T)$ is the minimum of all $E_k(T)$ for all nodes k in the system.

Algorithm 1 edftd(jobs, i, w)

```
for job J ∈ jobs do
  localTasks = {T ∈ J ∧ ready(T) | dnode(T) = i};
  if localTasks ≠ ∅ then
    return breakTies(localTasks);
  end if
  nearTasks = {T ∈ J ∧ ready(T) | E_i(T) < w × E_min(T)};
  if nearTasks ≠ ∅ then
    return breakTies(nearTasks);
  end if
end for
return the first ready task of jobs[1]
```

The algorithm works as follows. The scheduler scans through the job queue in increasing deadline order. For each job J, if J has some ready (to be executed) tasks with data on node i, it selects one of those to schedule based on tie-breaking rules. Here, ties are broken based on the smallest value of *laxity* (elapse time to deadline less estimated execution time) and the index of a task, in that order. In case all ready tasks of J are remote on node i, the scheduler executes the tasks whose data are near to node i. If no ready tasks in J satisfies the above criteria, the scheduler goes to the next job in the queue and repeats the same process. Eventually, if no tasks are found, the scheduler simply picks the first ready task of the first job in the queue.

EDF/MR scheduling policy. The scheduler partitions the ready tasks into two disjoint sets: Schedulable, consisting of all and only tasks of the jobs that are predicted to meet their deadlines; and Unschedulable, consisting of the remaining ones. Only the Schedulable set is considered for scheduling unless it is empty, in which case, the Unschedulable set is considered. The highest priority task is chosen for execution from the considered set based on the EDF/TD policy above.

A job is predicted to miss deadline if $t + \mathsf{remainExec}(J) > d_J$ where t is the current time, $\mathsf{remainExec}(J)$ is the estimated remaining execution time, and d_J is the deadline of J. $\mathsf{remainExec}(J)$ is the total estimated WCET of all unfinished tasks in J, assuming they are given all slots. Specifically, let s be the total number of slots within the system ($s = \sum_{k=1}^{M} s_k$) and nMaps (nReduces) be the number of unfinished map (reduce) tasks in J. Then,

$$\mathsf{remainExec}(J) = E^m_{\max} \times \left\lceil \frac{\mathsf{nMaps}}{s} \right\rceil + E^r_{\max} \times \left\lceil \frac{\mathsf{nReduces}}{s} \right\rceil$$

where $E^m_{\max}$ ($E^r_{\max}$) is the maximum among the estimated WCET $E_k(T)$ on any node k of any unfinished map (reduce) task T.

E. WCET Computation

We first outline our measurement method for the unit WCET per tuple u^m_i (u^r_i) of a map (reduce) task on a node i (which will be used for every other node with the same system configuration). Based on the measured values, we present our estimation of the execution time $E_i(T)$ of a task T on node i (c.f. Table I).

Unit WCET measurement. The unit WCET of a task on a node is estimated based on debug runnings on small sample inputs on the node. Specifically, we perform multiple runs of the corresponding job for a set of sample inputs on the node, and record the number of input tuples and the execution time for each map or reduce task. We then derive the unit execution time of each task (execution time per tuple). The unit WCET u^m_i (u^r_i) of a map (reduce) task on the node is assigned to be the average unit execution time, added with the standard deviation of unit execution times across all map (reduce) tasks[1]. Our earlier experimental results [36] show that these values are stable when the slot-to-core ratio (s_i/c_i) is set to 1.

Task WCET estimation. The WCET of a task on a node i is estimated as the total WCETs of all its input tuples (on the node), plus the input data transfer time if its input data is not on the node. The first part is the product of its unit WCET measured above and the number of input tuples. The second part is the ratio of its input size (i.e., number of tuples times the tuple size) to the transfer rate between the task's data node and the node i. Since the input data of a reduce task can be distributed over multiple nodes, we simply use the minimum transfer rate between node i and any other node. For our evaluation, we consider a fixed network topology, and we assign the minimum transfer rate to be the minimum of the rates measured during the same time period of the day as when the evaluation is taken.

For a reduce task, the number of input tuples depends on the output results of the map tasks of the same job, which is only known during run-time. To estimate this value, one simple way is to use the *selectivity*[2] of the map function. However, this approach works only on uniform distribution of (key, value) pairs. For skewed data distribution, the number of tuples processed by each reduce task varies significantly. In our case study, we measure by profiling. Specifically, we first log the size of partitioned data for each reduce task when a map task is done, then aggregate the logged information to get the total input size for each reduce task at the end of the map phase. The number of tuples will then be the ratio of the input data size to the tuple size.

[1]Since safe WCET estimation is not required for soft real-time tasks, we use this method instead of taking the actual maximum value to avoid overly pessimistic WCETs.

[2]We define selectivity to be the ratio of output to input size for a given map task.

F. Evaluation

The goals of our evaluation are as follows. First, we aim to study the effectiveness of existing Hadoop schedulers at meeting job-specified deadlines. Second, we compare our real-time schedulers with Hadoop's schedulers. While Hadoop's default schedulers are certainly not designed with real-time applications in mind, our evaluation aims to experimentally quantify how well (or badly) they perform given a particular system load, and whether our strawman solutions (based on adaptations to well-known multiprocessor scheduling techniques) can offer improvements. Third, we aim to understand the impact of data skews and communication delays on scheduling performance.

To study the above three goals, we have implemented a prototype *HadoopRT* system, based on modifications to Hadoop (version 0.20.2). HadoopRT includes new scheduler plug-ins for the enhanced EDF scheduler (MR and TD variants with locality-aware and overload handling). Our implementation also includes an extended API that allows users to submit jobs with deadlines.

Our experiments are carried out on 21 Amazon EC2 medium size *compute instance*, where we deploy 1 master node and 20 slave nodes on each instance. Each instance has 1.7GB of memory, and runs 5 *compute units*. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Experimental workload. Our workload consists of MapReduce jobs selected from the following three MapReduce programs: the *WordCount* example available in Hadoop's distribution, and two Pig queries as described in [27]. The two Pig queries (*Aggregation* and *Join*) are representative queries that perform a SUM aggregation with a single group-by attribute, and a typical select-project-join query involving a relational parallel join of two tables respectively. Table II summarizes our workload, by showing the input data size, number of map and reduce tasks, and job completion time (averaged across all experiments) of the three programs used in our experiments. We measure the completion time for each job by taking the difference between the completion and submission times of a MapReduce program execution, averaged across several experimental runs.

Program	Input	# Map	# Reduce	Avg Time (s)
WordCount	2.4GB	40	40	210
Aggregation	5GB	80	160	87
Join	10GB	64	64	320

TABLE II
JOB DESCRIPTIONS

For each MapReduce program, we additionally obtain the unit WCET measurements (separately for tasks local and remote to slaves) by profiling each MapReduce program over multiple execution runs. Since we have a homogeneous cluster with identical compute instances and bandwidth, each program has only three unit WCET measurements, each for local map tasks, remote map tasks, and reduce tasks, respectively. To ensure a stable WCET measurement, we set the slot-to-core ratio at each slave to be 1 [36]. This results in stable tasks' execution times and thus, a stable unit WCET values (where the standard deviation is only 10% of the average). The task WCETs are computed based on the measured unit WCET and the experimental platform settings as detailed in Section III-E.

Based on the above three programs, we generate a workload consisting of 30 jobs (executed over a 1 hour period), selected randomly from the three MapReduce programs above. The arrival times of jobs follow the Poisson distribution with the expected number of occurrences $\lambda = 0.1$ (seconds). We assign the job deadlines in order to emulate normal and high load scenarios. For a given job J, we define its utilization U_J to be the ratio of its estimated WCET to its relative deadline. α is set to the sum of all U_J for all 30 jobs in each experiment, where a higher value of α means a heavier load on the system (jobs having tighter deadlines). In our experiments, we set α to be 1.0 and 3.0 for the normal and high load scenarios.

We use a mixture of 3 MapReduce programs in our experiments. One is the *WordCount* example which is available as part of Hadoop's distribution. The other two are Pig programs based on the data set and data generator described in [27]. Particularly, We choose the datasets *Rankings* and *UserVisits*, the Rankings table contains two attributes which are *pargeURL* and *pageRank*. the UserVisits table contains nine attributes which are *sourceIP*, *destURL*, *visitDate*, *adRevenue*, *userAgent*, *countryCode*,*languageCode*, *searchWord* and *duration*. The queries we used in our experiments are

- aggregation query: SELECT sourceIP, SUM(adRevenue) FROM UserVisits GROUP BY sourceIP
- join query: SELECT INTO Temp sourceIP, pageRank, adRevenue FROM Rankings as R, UserVisits as UV WHERE R.pageURL = UV.destURL

Experimental results. Given the above experimental setup, we evaluate the effectiveness of our real-time schedulers (EDF/MR and EDF/TD) and the Hadoop's default FIFO, Fair, and Capacity scheduler. In the capacity scheduler, we configured two job queues at the master, one with 80% capacity which is for urgent jobs and the other with 20% capacity. We classify a job J as urgent if its utilization U_J is above the average utilization of all jobs in the experiment. All experiments are averaged across two runs.

For each experimental run, we evaluate all scheduling policies against three metrics introduced in Section III-A:

- Miss rate: Percentage of jobs that missed their deadlines.
- Total and max tardiness: For each job that missed their deadlines, we compute its tardiness the total time difference between the job deadline and job completion for all jobs that miss their deadlines.
- Max tardiness: the maximum time difference between the job deadline and job completion among all jobs that miss their deadlines.

In the following, we present the experimental results under different data distributions and load conditions.

Uniform data distribution. Table III and Table IV show the results for normal and high load scenarios under uniform data distribution.

We make the following observations:

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	16.7	4990	1420
Fair	30.0	7627	1565
Capacity	3.3	150	150
EDF/MR	0.0	0	0
EDF/TD	0.0	0	0

TABLE III
PERFORMANCE OF SCHEDULERS UNDER NORMAL LOAD.

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	46.7	16958	2535
Fair	76.7	33672	2483
Capacity	43.3	10825	1811
EDF/MR	13.3	7756	2106
EDF/TD	20.0	1261	321

TABLE IV
PERFORMANCE OF SCHEDULERS UNDER HIGH LOAD.

- Default schedulers are clearly ill-equipped for meeting job deadlines. Even under a normal load scenario, FAIR results in a miss rate of 30%. Under high load, all existing schedulers (FIFO, Fair, and Capacity) perform poorly, resulting in high miss rates and tardiness.
- On the other hand, the real-time schedulers (EDF/MR, EDF/TD) outperform Hadoop's default schedulers. Under normal load, all the real-time schedulers result in zero miss rates. Under high load, we observe that the real-time schedulers perform favorably compared to the default schedulers. EDF/TD achieves an order of magnitude improvement in tardiness over the default schedulers.
- Overload handling is effectively at mitigating the negative effects of cascaded missed deadlines under high load. EDF/MR (which includes this optimization) has a low miss rate of only 13.3% (6X and 3.5X improvements over Fair and FIFO/Capacity) since it avoids running unschedulable tasks under high load. EDF/TD, on the other hand, incurs a higher miss rate compared to EDF/MR without this optimization, but nevertheless, results in 2-4X improvements over default schedulers. All in all, the two EDF variants show a tradeoff between miss rates and tardiness.

Skewed data distribution. To evaluate the impact of data skews on the performance of the algorithms, we repeat our experiments in Section III-F while varying data distribution. We modify the input data to the three MapReduce programs, such that the data distribution is now heavily skewed (Zipf distribution parameterized to 1.0), as opposed to a uniform distribution. We observe from Tables V and VI that both EDF/MR and EDF/TD outperform default schedulers across all performance metrics in both the normal and high load scenarios, an observation consistent with our earlier evaluation results. For instance, in the normal load case, both EDF schemes result in 0% miss rate, compared to FIFO (23%), Fair (30%), and Capacity (6.7%). EDF/MR and EDF/TD also show the tradeoff between miss rate and tardiness that we observed in our earlier experiments.

To examine the benefits of the overload handling optimization, we compare EDF/MR against a pathological EDF/MR

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	23.0	5461	1418
Fair	30.0	8186	1567
Capacity	6.7	378	325
EDF/MR	0	0	0
EDF/TD	0	0	0

TABLE V
PERFORMANCE OF SCHEDULERS WITH SKEWED DATA DISTRIBUTION UNDER NORMAL LOAD.

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	53.3	19147	2424
Fair	76.7	31753	2179
Capacity	63.3	13720	1877
EDF/MR	33.3	18824	2715
EDF/TD	43.3	4258	749

TABLE VI
PERFORMANCE OF SCHEDULERS WITH SKEWED DATA DISTRIBUTION UNDER HIGH LOAD.

case in the high load case, where the WCET is miscalculated to assume a uniform (instead of a skewed) distribution. We observe that this results in a 10% higher miss rate, indicating that the overload handling (for forgoing unschedulable tasks) optimization is highly dependent on the current estimation of task WCET.

Locality-aware scheduling. We study the effectiveness of locality-aware optimizations in ensuring good scheduler performance. Given that our EC2 setup has high speed connectivity across all nodes, we emulate transmission delays by adding a significant computation delay to all remote map tasks. This results in each remote task requiring approximately 8 times the execution time of a local task. By setting the w parameter (weight factor of scheduler) in the locality-aware EDF policy to be 2, we ensure that the locality-aware enhancement is used at times when running EDF-based schedulers. One important future work to pursue is a cost-based approach towards setting the w parameter, based on the measured differences between local and remote task WCET.

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	50.0	41706	5992
Fair	90.0	125338	6940
Capacity	66.7	55111	5240
EDF/MR	0	0	0
EDF/TD	0	0	0

TABLE VII
PERFORMANCE OF SCHEDULERS UNDER NORMAL LOAD WITH INCREASED EXECUTION TIME DIFFERENCES BETWEEN LOCAL AND REMOTE TASKS.

Our evaluation results in Tables VII and VIII show that both EDF/MR and EDF/TD outperform the default schedulers in the normal and high load scenarios, with the expected tradeoff in miss rate and tardiness in their relative comparisons. To quantify the benefits of locality-aware optimizations, we compare EDF/MR with and without this optimization, and observe a significant performance improvement (4X reduction in miss rate, 9.4X reduction in total tardiness), highlighting

Scheduling Policy	Miss rate (%)	Tardiness (s)	
		Total	Maximum
FIFO	66.7	92066	8272
Fair	100.0	194911	8039
Capacity	86.7	115690	9849
EDF/MR	20.0	13363	2743
EDF/TD	43.3	4662	727

TABLE VIII
PERFORMANCE OF SCHEDULERS UNDER HIGH LOAD WITH INCREASED EXECUTION TIME DIFFERENCES BETWEEN LOCAL AND REMOTE TASKS.

the importance of locality-awareness in making scheduling decisions.

IV. CONCLUSION

We have presented an empirical analysis of various scheduling techniques for large-scale real-time data processing applications on cloud platforms. Our evaluation results of a Hadoop's MapReduce case study on Amazon EC2 show that existing schedulers in the current Hadoop implementation perform poorly in terms of meeting deadlines and response time. This is expected since these schedulers were not designed for real-time performance. On the other hand, direct applications of conventional real-time scheduling techniques (e.g., EDF) without considering cloud platform related factors, such as data locality and unpredictable system load, offer little improvement over Hadoop's default schedulers. We have also proposed and evaluated different locality-aware cum overload-handling enhancements of EDF. Our extensive empirical evaluations show that the proposed methods are highly effective in minimizing both deadline miss ratio and tardiness. Moreover, the effectiveness of each scheduling algorithm is dependent on (1) an accurate measure of data distribution (for the purpose of correctly estimating reduce task WCET for overload handling) and (2) locality-aware enhancements.

Our empirical analysis reveals several interesting insights to the challenges in guaranteeing timeliness in the cloud environment. We outline some of our ongoing research.

Cloud Unpredictability. Unpredictability is the biggest bane of any real-time scheduler. To provide a specific example based on WCET measurements, the effectiveness of the real-time scheduling policies in Section III-B largely hinges on correct prediction of task and job level WCET. Indeed, our evaluation results clearly demonstrate that having perfect knowledge of WCET (from prior performance profiling) can result in a significantly better overall system performance in meeting timing guarantees. Moreover, mispredicting WCET in the presence of data skews and increased delays may result in poor scheduling decisions.

To improve the accuracy of both techniques, one potential direction is to investigate *parameterized* WCET [7] techniques for large-scale data analytics in the cloud environment. The idea is to incorporate with the analysis useful information that may affect a task's execution time (or its dependent tasks) as parameters to achieve more accurate estimates. These parameters include (i) data-related information, e.g., the size of the input/output data, the distribution of the data values; (ii) middle-ware and architectural information, e.g., the block size, data buffer size, I/O and network speed, data-transfer rate, processor configuration; and (iii) design-level knowledge, e.g., different phases of the application that may lead to mutually exclusive paths in the program.

For run-time measurements, these parameters help to provide a better coverage on the range of input data and system architecture, which in turn leads to more coverage of relevant execution scenarios. Likewise, static methods compute the result symbolically and give results based on these parameters. The information encoded by these parameters does not only help to address the challenges described above but also makes the estimation more flexible.

Another promising area of research is in *probabilistic* scheduling techniques for soft real-time applications. These techniques are particularly useful for performing statistical analysis of WCET [21], and has been applied to great success in real-time applications with probabilistic task arrival and execution times (e.g., [33]).

Runtime Scheduler Adaptation. One observation from our evaluation study is that even with all our assumptions, determining a *one-size-fits-all* scheduling policy is non-trivial. For instance, depending on the overall system goal of minimizing miss rate or tardiness, EDF/MR or EDF/TD is preferred over the other. Policies are highly dependent on the presence data skews, and the absolute differences between local vs remote task execution in determining whether to apply locality-based enhancements. Even when a single scheduling policy is fixed, the specific configuration parameters require careful performance tuning based on deployment scenarios.

One interesting direction we plan to pursue is the use of *multi-model resource allocation* [37] techniques, commonly used in dynamic real-time systems. At a high level, multi-model techniques significantly reduces the search space, by restricting the scheduler to search for optimal policies within "modes" (preset cloud configurations), and switching across modes whenever cloud configurations change. Applying multi-mode scheduling to the cloud requires additional research to deal with its unique nature, in particular, precedence and data dependency of MapReduce workflows, developing a set of scheduling strategies for different modes of a cloud system and their switching mechanisms, and incorporating the use of feedback-based control methods.

Hierarchical Scheduling. The motivating applications in our introduction suggest that cloud analytics is increasingly deployed at an Internet-scale, since the input data themselves are inherently distributed. This is in contrast to today's MapReduce usage, which is typically assumed to run within a single data center with high bandwidth connectivity. Scaling up will further lead to unpredictability at many levels (e.g., cluster, data center, administrative domains over the Internet).

Since a cloud infrastructure is inherently hierarchical, one can view the cloud scheduling as a multi-level hierarchical scheduling problem, consisting of the infrastructure scheduler (e.g., Amazon EC2's cloud scheduler), the middleware schedulers (e.g., Hadoop's master scheduler), and the virtual

machine schedulers (e.g., Xen scheduler). From the above observation, a promising approach that we plan to explore is the use of hierarchical scheduling techniques (e.g., [37], [39]) in the cloud setting as they are not only a natural choice of the cloud infrastructure but also allow compositionality and reduce system complexity, particularly when scaling cloud analytics to operate at Internet-scale. These techniques also provide mechanisms for guaranteeing timeliness in virtualized environments as demonstrated in [40] and [44].

V. Acknowledgments

This research is funded by NSF grants CNS-1117185, CNS-1040672, CNS-0834524, and ARO grant W911NF-11-1-0403.

References

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. In *VLDB*, 2008.

[2] Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/.

[3] J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *RTCSA*, 2000.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996. 10.1007/BF01940883.

[5] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.

[6] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[7] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. In *RTCSA*, 2009.

[8] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical multiprocessor CPU reservations for the linux kernel. In *OSPERT*, 2009.

[9] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *SIGMOD*, 2010.

[10] Songting Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.

[11] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35:42–51, September 2007.

[12] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328, 2010.

[13] Silviu S. Craciunas, Andreas Haas, Christoph M. Kirsch, Hannes Payer, Harald Rak, Andreas Rottmann, Ana Sokolova, Rainer Trummer, Joshua Love, and Raja Sengupta. Information-acquisition-as-a-service for cyber-physical cloud computing. In *Hotcloud*, 2011.

[14] Dan Craytor. Transforming warfare with cloud services. *ITEA Journal*, 2011.

[15] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting temporal constraints in virtualised services. In *COMPSAC*, 2009.

[16] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, 2011.

[17] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: integrating r and hadoop. In *SIGMOD*, 2010.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] Tim Dornemann, Ernst Juhnke, and Bernd Freisleben. On-demand resource provisioning for bpel workflows using amazon's elastic compute cloud. In *CCGrid*, 2009.

[20] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *PVLDB*, 2010.

[21] D. Griffin and A. Burns. Realism in Statistical Analysis of Worst Case Execution Times. In *WCET*, 2010.

[22] Hadoop. http://hadoop.apache.org/.

[23] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. FlexPRICE: Flexible provisioning of resources in a cloud environment. In *Proc. of IEEE International Conference on Cloud Computing (CLOUD)*, 2010.

[24] David Irwin, Prashant Shenoy, Emmanuel Cecchet, and Michael Zink. Resource management in data-intensive clouds:opportunities and challenges. In *LANMAN*, 2010.

[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[26] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[27] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[28] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In *Real Time Computing*, 1994.

[29] J. Lee, A. Easwaran, and I. Shin. Llf schedulability analysis on multiprocessor platforms. In *RTSS*, 2010.

[30] Youngseok Lee, Wonchul Kang, and Hyeongu Son. An internet traffic analysis method with mapreduce. In *NOMS*, 2010.

[31] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, Jan 1973.

[32] Ke Liu, Hai Jin, Jinjun Chen, Xiao Liu, Dong Yuan, and Yun Yang. A Compromised-Time-Cost Scheduling Algorithm in SwinDeW-C for Instance-Intensive Cost-Constrained Workflows on Cloud Computing Platform. *J. of High Performance Computing Applications*, 24:445–456, Nov 2010.

[33] A. F. Mills and J. H. Anderson. A Stochastic Framework for Multiprocessor Soft Real-Time Scheduling. In *RTAS*, 2010.

[34] Ioannis Moschakis and Helen Karatza. Evaluation of gang scheduling performance and cost in a cloud computing system. *The Journal of Supercomputing*, 2010.

[35] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[36] Linh T.X. Phan, Zhuoyao Zhang, Boon Thau Loo, and Insup Lee. Real-time mapreduce scheduling. Technical report, U. of Pennsylvania, 2010. MS-CIS-10-32.

[37] L.T.X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *ECRTS*, 2010.

[38] Yahoo Capacity Scheduler. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html.

[39] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.

[40] Chenyang Lu Sisu Xi, Justin Wilson and Christopher Gill. Rt-xen:real-time virtualization based on hierarchical scheduling. Technical report, Washington U., 2010. WUCSE-2010-38.

[41] A. Verma, L. Cherkasova, and R. Campbell. Aria: Automatic resource inference a nd allocation for mapreduce environments. In *ICAC*, 2011.

[42] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[43] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Middleware 2010*, 2010.

[44] Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong, and Insik Shin. Implementation of compositional scheduling framework on virtualization. In *CRTS*, 2010.

[45] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing. In *OSDI*, 2008.

[46] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, U. of California, Berkeley, 2009.

[47] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[48] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.