Real-time MapReduce Scheduling

Linh T.X. Phan Zhuoyao Zhang Boon Thau Loo Insup Lee
University of Pennsylvania

ABSTRACT

In this paper, we explore the feasibility of enabling the scheduling of mixed hard and soft real-time MapReduce applications. We first present an experimental evaluation of the popular Hadoop MapReduce middleware on the Amazon EC2 cloud. Our evaluation reveals tradeoffs between overall system throughput and execution time predictability, as well as highlights a number of factors affecting real-time scheduling, such as data placement, concurrent users, and master scheduling overhead. Based on our evaluation study, we present a formal model for capturing real-time MapReduce applications and the Hadoop platform. Using this model, we formulate the offline scheduling of real-time MapReduce jobs on a heterogeneous distributed Hadoop architecture as a constraint satisfaction problem (CSP) and introduce various search strategies for the formulation. We propose an enhancement of MapReduce's execution model and a range of heuristic techniques for the online scheduling. We further outline some of our future directions that apply state-of-the-art techniques in the real-time scheduling literature.

1. INTRODUCTION

MapReduce [7] has emerged as one of the most popular frameworks for data-intensive distributed cloud computing. The benefits of this simple yet powerful programming model has been demonstrated on a wide spectrum of domains, ranging from search and ads analysis (e.g., [13,11,2]), bioinformatics (e.g. [25,17,16]), to artificial intelligence, machine learning and data mining (e.g. [30,3,9]).

In this paper, we explore the feasibility of enabling *real-time scheduling* in MapReduce job executions, i.e., scheduling MapReduce jobs in a multi-user environment where some of these jobs have to complete within a given deadline. We argue that current MapReduce scheduling techniques are not suited for real-time applications. One of the goals of our work is to identify factors that affect *execution predictability* at the level of individual worker tasks. Such predictability enables the MapReduce scheduler to make fine grained task scheduling decisions that ensures that deadlines can be met. While our eventual goal is to provide real-time guarantees for cloud applications in general, this paper focuses primarily on the MapReduce framework given its wide usage and well-defined execution model.

As MapReduce continues to evolve from a middleware for data-intensive distributing computation of batch jobs to one better suited for continuous stream processing [4], there are emerging classes of cloud-based applications that can benefit from increasing timing guarantees. For instance, real-time advertising or other forms of personalization over the web requires a real-time analysis process that predicts user intent based on their profile and search histories. These applications typically require *soft real-time* requirements, where meeting deadlines (or minimizing missed deadlines) typically translates into higher profits or utility for the content providers.

In terms of applications with *hard real-time* requirements, an intriguing prospect is the deployment of mission critical applications with tight deadlines over the cloud. For instance, control centers for traffic management are connected to different devices (such as detectors on roads, cameras, traffic lights, etc.). The operators can supervise the state of the road by consulting databases with recent information from detectors and modify the state of control devices. A real-time decision support tool is needed to help operators in detecting traffic problems and choosing appropriate control actions.

The ability to satisfy timing constraints of such real-time applications as illustrated above is not only required by the nature of the applications but also driven by the needs for a more flexible, transparent and trust-worthy service agreement between cloud providers and users. Most current service level agreements (SLAs) provide only static information on the machine specifications and no guarantees on the actual performance of the system. This gives users very little knowledge and control over the timing behavior of the applications that run on the cloud environment.

Problem Formulation. Given a set of MapReduce jobs, each with several map and reduce tasks and a deadline (which can be either hard or soft deadline), we aim to provide a scheduling algorithm for the tasks such that (i) all hard real-time jobs will meet their deadlines, and (ii) either the number of soft real-time jobs that meet their deadlines is maximized, or the maximum *tardiness* of the soft real-time jobs is minimized. We consider both *online* and *offline* variant of the problem. In the offline variant, all MapReduce programs and their respective parameters are known a-priori, and an optimal scheduling (assignment of tasks to machines at specific times) is given to the system. In the online variant of the problem, MapReduce programs are continuously being issued by end-users, and the system does not a priori know the number of tasks per program. The scheduler hence has to make an online decision on assigning a task to the next available machine.

This paper makes three distinct contributions towards the above problem formulation:

MapReduce performance evaluation: We conduct a performance evaluation of MapReduce programs written in Hadoop [12] on the Amazon EC2 cloud. Unlike prior measurements, our measurements focus on identifying factors that affect the predictability of MapReduce programs, i.e., factors that makes task-level real-time scheduling challenging in this environment. Our performance evaluation reveals tradeoffs between overall system throughput and execution time predictability. Our study further highlights a number of factors that affect real-time scheduling, such as slots per processing core, data placement, concurrent job execution, and master scheduling overhead.

CSP formulation: Based on the above factors and MapReduce's execution model, we present a formal model for capturing the real-time aspect of MapReduce applications and Hadoop execution platform. Using this model, we formulate the scheduling problem as a constraint satisfaction problem (CSP). Our formulation provides a model for understanding the performance characteristics of MapReduce programs under real-time requirements, and provides a basis for computing a theoretical optimal bound on performance. We then present several CSP search strategies to improve its efficiency and a refinement of the formulation.

Practical online heuristics: We outline the challenges in the design of online schedulers for real-time MapReduce computations, and present a collection of ideas from the real-time literature that can be applied towards the design of such a scheduler. Specifically, we propose some of the changes to the current Hadoop implementation to allow real-time guarantees. As there exists no optimal online schedulers, we present a range of heuristic techniques that are adapted from the real-time domain. We further propose the use of hierarchical scheduling, real-time virtual machines, and probabilistic models to tackle the scheduling challenges associated with the real-time constraints cum the unpredictability nature of the cloud.

2. RELATED WORK

There has been a large amount of work focusing on the performance optimization of MapReduce applications on Hadoop architecture. For instance, the performance study presented in [15] identified several factors effecting the system performance, including I/O mode, record parsing and scheduling strategy. The authors also proposed various strategies considering these factors to improve the completion time of MapReduce jobs. Similarly, [27] developed a MapReduce simulator called MRPerf, using which one can simulate the behavior of MapReduce applications. Based on the simulated results, one can measure the effect of data locality, network topology and node failures on the performance of MapReduce applications. A number of scheduling strategies have also been proposed as alternatives to the default Hadoop schedulers. Along the same lines, [29] introduced the LATE scheduler that schedules tasks with smaller approximate time-to-end first, and showed that it is able to improve the completion time of MapReduce jobs in heterogeneous environments. [22] proposed a technique to improve the total execution time of the jobs by sharing similar work among multiple jobs in Hadoop. None of these above framework considers real-time aspect of jobs, however. Instead, their primary focus is to improve the performance (e.g., completion time reduction) of regular batch-mode MapReduce jobs.

Another body of related work considers the completion time estimation, whose results are often used by online schedulers. For instance, Parallax [21] and its extension ParaTimer [20], which can estimate the remaining execution time of MapReduce jobs. The ParaTimer considers multi-stage MapReduce jobs with concurrent execution, machine failures, and data skew using a probabilistic model. These analysis results, however, cannot be used as the input to our real-time setting since they are subjected to the default Hadoop schedulers. Both Parallax and ParaTimer do not consider the scheduling problem as in our case: their primarily goal is to analyze the execution time of a job when being scheduled by the default Hadoop schedulers.

3. AMAZON EC2 MEASUREMENTS

We conduct experiments to evaluate the performance characteristics of MapReduce jobs running on the Amazon EC2 platform. The goals of our evaluation are to understand the extent in which the completion time of a MapReduce job is predictable, and the factors that are important to its predictability. We then use these factors to develop a performance model for MapReduce in Section 4. Unlike prior studies, we also examine completion times at the level of individual tasks executed across different nodes. The focus on *task level* completion time is essential, since without which, a schedule cannot make an informed decision on which task to execute next in order to meet specific timing constraints.

3.1 Review: MapReduce Scheduling

We provide a brief background review of scheduling in MapReduce. Our discussion is primarily based on Hadoop's implementation, although our observations and results can generally be applied to other MapReduce implementation. The terminology we adopt is as follows. In a MapReduce *job*, there are multiple map and reduce *tasks*, each of which is a single unit of work that can be performed in parallel at the map and reduce phases. Job scheduling in Hadoop is performed by a *master* node, which distributes work to a number of *slave* nodes. Each slave corresponds to one physical machine, and has a number of predefined map/reduce *slots* for executing Map and Reduce tasks. Typically, these slots can be more than the number of cores and the specific number of slots per machine are predefined as a configuration parameter.

Scheduling of MapReduce jobs in Hadoop proceeds as follows. Periodically, slave nodes inform the master about the availability of free map/reduce slots. The master in this case will assign a pending task accordingly to a free slot based on a *scheduling policy*.

The current Hadoop implementation includes two typical scheduling policies. The first is a *FIFO* policy, in which the scheduling maintains a FIFO waiting queue of jobs sorted by arrival time. Whenever a slot becomes available, the master simply selects a task from the next job on the waiting queue for execution. While FIFO is simple to implement, it does not prioritize particular MapReduce jobs, or ensure that all jobs get equal opportunities for making execution progress.

As an alternative strategy, *fair* scheduling policy ensures that all submitted MapReduce jobs will get an equal opportunity for executing their tasks. Due to space constraints, we refer the user to Hadoop's online documentation for more details on this policy.

3.2 Experimental Setup

Our experiments are carried out on 20 EC2 instances, each of which runs a dual-core processor with 1.7GB of memory. Each instance runs 5 *compute units* (where each unit equivalents CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor). Our experiments are based on two MapReduce programs: *WordCount* and *TeraSort*. Our choice of using these two programs in our evaluation is due to their popularity as well as well-understood performance characteristics. We are also in the process of performing a similar evaluation of other MapReduce programs, particularly those that are more computationally expensive, e.g., statistical learning algorithms.

In all our experiments, we adopt the use of FIFO scheduling. We further disable speculative execution and the use of pipelining, i.e., the reduce phase is started only after the map phase is completed. We observe no machine failures during our experiments. We note that disabling these features allows us to study in isolation a number of factors impacting real-time predictability, and simplifies our performance model in the next section.

3.3 Slot-to-Core Ratio

Our first set of evaluation studies the impact of increasing the number of slots per core at each machine on task completion time. As the number of slots per core increases, more tasks can be executed per physical machine, hence providing increase in throughput due to the interleaving of I/O and computation at each node, at the expense of increased variance in task completion times due to OS-level scheduling and context switching. In the rest of this section, we examine the average task and overall job performance of *WordCount* and *TeraSort* program executions, as the *slot-to-core* ratio increases. The ratio is defined by the ratio of the number of slots to cores at each machine. All experiments here involve running one MapReduce job, i.e., either *WordCount* or *TeraSort*, but not both jobs (or multiple instances of the same job) simultaneously.

Figure 1 shows the impact of slot-to-core ratio on the average completion time for map tasks executing in the *WordCount* and *TeraSort* programs as the slot-to-core ratio increases. In both programs, the input size of each map task is 60 MB, and each job contains 100 map tasks each (hence the aggregate data size is 6 GB). The results are based on averaging the map task completion times across all nodes.

We observe that as the ratio increases, the average task completion time increases. This increase is as expected, due to the fact that the Hadoop scheduler assigns tasks to slots without consideration of each node's capacity. Hence, in the scenario where there are more pending tasks than slots, each machine now has to process more tasks simultaneously (resulting in contention for CPU, local disk I/O, and possibly network bandwidth), hence resulting in degradation in completion time. Moreover, we observe from Figure 2 that as the slot-to-core ratio increases, there is also an in-

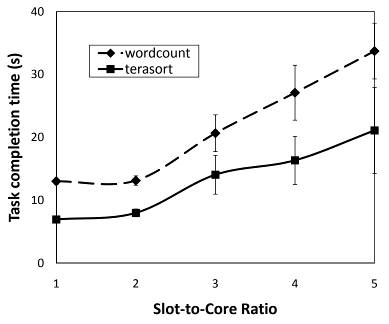

Figure 1: Slot-to-core ratio vs average task completion time (s) (error bars indicate standard deviation).

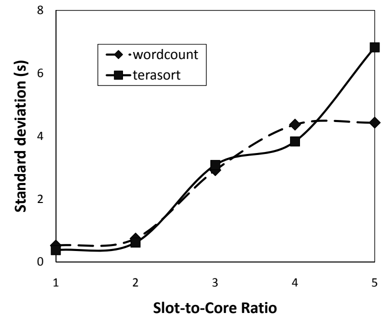

Figure 2: Slot-to-core ratio vs task completion time variability (standard deviation)

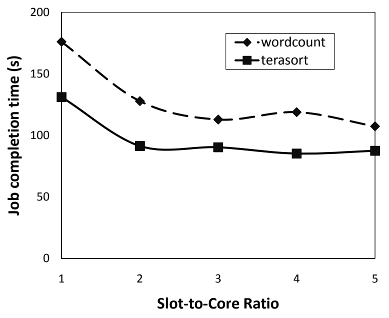

Figure 3: Slot-to-core ratio vs job completion time (s).

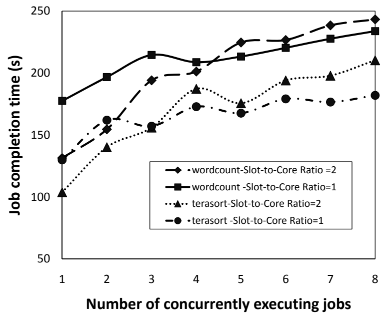

Figure 4: Slot-to-core ratio vs job completion time (s) in a multi-job environment

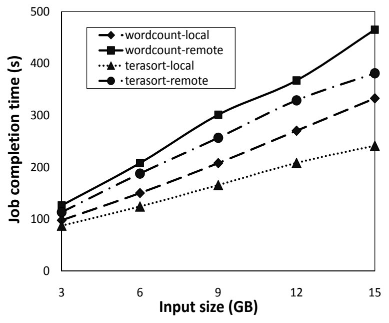

Figure 5: Job completion time (s) for varying data locality

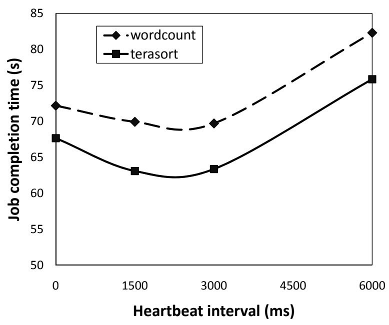

Figure 6: Job completion time (s) for various heartbeat interval

crease in the lack of predictability in the average completion times of tasks, as shown by the increase in standard deviation for completion times for a given ratio.

Interestingly, we note that the increase in slots per core actually results in an overall *performance improvement* for the entire job. This behavior seems counter-intuitive, given that individual tasks have increased completion times when the slot-to-core ratio increases. Figure 3 summarizes this observation, where the completion time decreases and levels off after the slot-to-core ratio increases from 1 to 5. The improvement in completion time is due to two factors: first, increasing the number of slots means that less compute resources are idle, since half of the slots are assigned to map and reduce tasks, and in the absence of pipelining and concurrent jobs, either the map or reduce phase is in progress at any time; Second (and more importantly), there is a reduction in queueing delay, since the availability of more slots means that more tasks can execute simultaneously. However, the improvement levels off after a ratio of 3, since beyond that, there are enough slots to handle all 100 map tasks with minimal queueing delays.

We make similar observations on the average completion time of reduce tasks, and we omit the results due to space constraints. All in all, our results demonstrate a tradeoff in performance throughput and predictability. As the slot-to-core ratio increases, performance of a job increases, but comes at the expense of increased variability at the completion time of individual tasks. This predictability (as we demonstrate in the next section) is essential for scheduling tasks at a finer-granularity, in order to meet tight timing constraints.

3.4 Multiple Concurrent Jobs

We next examine the impact of executing multiple concurrently executing MapReduce jobs. The setup of this experiment is the same as before, except that we fix the slot-to-core ratio to be 1 and 2 respectively, and then run multiple instances of either the *WordCount* or *TeraSort* programs. Figure 4 shows that as the number of concurrent jobs increases, the job completion time increases. Moreover we observe that having twice the number of slots actually results in performance degradation as the number of jobs increases. This is due to increased contention for resources (both compute and I/O), particularly when map and reduce phases are now interleaving due to the presence of several concurrent jobs.

Overall, our results demonstrate that where there are concurrently executing jobs, having a larger number of slots may initially result in improved job completion times at moderate load. However, as the number of concurrent jobs increases, having a higher slot-to-core ratio may actually result in degradation in performance, due to increased contention for resources and time-sharing among different tasks.

3.5 Data Placement

In the next evaluation, we examine the impact of data placement on the overall job completion time of a MapReduce program. Data placement has an impact on the completion time of map tasks, given that if a map task is placed on a machine where its input data resides (*node-local*), only local disk I/O is required as opposed to a more expensive network transmission.

The current Hadoop scheduler follows a simple strategy to exploit such data locality. Whenever a slot becomes available, the master scheduler will first try to launch a node-local map task to the slave that owns the slot, otherwise, it will try to launch a rack-local map task, and finally, a non-local map task is launched. This "best effort" approach is not ideal for real-time application, since timing guarantees become highly dependent on whether a map task is node-local or not.

To experimentally evaluate the impact of data placement on job completion time, we execute a single instance of the *WordCount* and *TeraSort* program on two different configurations: *local* in which a majority of map tasks are node-local, and *remote*, where a majority of map tasks are non node-local[1].

Figure 5 shows the job completion time (averaged over 5 runs) as the input size increases. We observe that locality does play a significant role in determining job completion time. For instance, when the aggregate input size is 15 GB, *remote* requires 40% and 58% longer completion times compared to *local*, for the *WordCount* and *TeraSort* programs respectively. Note that job completion times are proportional to the sizes of the input data. This shows that our experiments did not result in the saturation of bandwidth on EC2 (which would have resulted in an exponential increase in completion time due to queueing delays and congestion). We conjecture that as the data size increases, the difference between *local* and *remote* will be even more larger in the presence of limited network resources. Understanding the impact of data locality in the presence of network as the bottleneck is an interesting avenue for our future exploration.

3.6 Slave-to-Master Heartbeat Interval

In our final evaluation, we examine the impact of the *heartbeat interval* in Hadoop on job completion time. This interval is the period in which slave nodes connect to the master to inform about free slots. Having a larger interval means that there is a time delay in which available slots are reported to the master. However, having too frequent updates from multiple slaves may result in overloading of the master.

To experimentally quantify the impact of the heartbeat interval, Figure 6 shows that when the heartbeat interval is small (1ms), the master incurs high overhead, and hence this negatively impacts job completion times (averaged across 5 runs) of both *WordCount* and *TeraSort*. However, as the interval increase, the job completion time decreases, due to reduced load on the master. Beyond 3000ms (Hadoop's default setting), we note that the delay in reporting slot availability quickly results in an increase in job completion times.

All in all, while the current Hadoop implementation is carefully tuned for its current scheduling algorithms, the heartbeat interval itself has a significant impact on global job completion times. In the context of real-time applications with tight timing constraints, the heartbeat interval needs to be carefully optimized and tuned in order to meet timing constraints.

4. REAL-TIME SCHEDULING AS A CSP

We formulate the problem of scheduling a set of real-time MapReduce applications on a distributed heterogeneous Hadoop architecture as a CSP, which can be solved using well-known constraint solvers. We focus first on the *offline* setting, where the set of MapReduce jobs are known a priori, and the role of the scheduler is then to determine an optimal execution schedule for all tasks. We revisit the online scenario in Section 5.

The novelty of our formulation lies in the modeling of various factors unique to the MapReduce jobs and the underlying Hadoop architecture that affect the system performance discussed in Section 3. Specifically, our formulation considers three factors validated in the previous section: (i) slot-to-core ratio, (ii) the effect of input data placement on the data transfer time (from a remote or a local host), and (iii) the interval based on heart beats between the master and the slaves. We added a fourth factor: (iv) the heterogeneity of the processors, where a task's execution time varies based on the slot's processing capability.

[1]We emulate this situation by storing all data on one node, hence requiring most map tasks to require non-local access.

While aiming to be as close to the current Hadoop execution platform as possible, to simplify the formulation, we make the following assumptions: (i) each job contains no more than one map stage and one reduce stage; (ii) the worst-case execution time (WCET) of a task on each processor type is known a priori; (iii) all processors work perfectly without failure; and (iv) there is no speculative execution and no task migration. Our formulation can easily be extended to allow jobs with multiple map/reduce stages. The WCET assumption is necessary for real-time guarantees, which can be obtained using well-known WCET estimation technique [28]. We plan to look into probabilistic models to capture machine failure and speculative execution (see Section 5).

4.1 CSP Formulation

Real-time MapReduce applications. The system consists of N MapReduce jobs $J_1, J_2, \ldots, J_N$, with $N \in \mathbb{N}, N \geq 1$. Each job J_i consists of m_i map tasks (J_i^1, $J_i^2, \ldots, J_i^{m_i}$) followed by r_i reduce tasks ($J_i^{m_i+1}$, $J_i^{m_i+2}, \ldots, J_i^{m_i+r_i}$), with $m_i, r_i \in \mathbb{N}$ and $m_i + r_i > 0$, for all $1 \leq i \leq N$. J_i is released (i.e., when user submitted the job for execution) at an offset o_i (relative to the beginning of the system execution) and has a relative deadline of d_i (with respect to the release time), where $o_i, d_i \in \mathbb{N}$, $o_i \geq 0$ and $d_i > 0$. Each task J_i^k has the same release time and deadline as J_i does. We consider a mix of jobs with hard and soft deadlines. The first N_1 jobs have hard deadlines whereas the last $N - N_1$ jobs have soft deadlines, where $0 \leq N_1 \leq N$. As usual, all map (reduce) tasks of a job can execute in parallel. A reduce task can only execute after all the map tasks of the same job have completed.

Underlying Hadoop architecture. The MapReduce applications are executed on a distributed heterogeneous architecture, which consists of M processors $P_1, P_2, \ldots, P_M$ running as slaves and a dedicated processor P_0 running as the master, where $M \in \mathbb{N}$ and $M \geq 1$. Each P_i contains nc_i identical cores, each of which can only execute at most one task at a time. Further, P_i is configured to have ns_i slots for holding the tasks assigned by the master.

Each task J_i^k when being executed on a core of P_j has a WCET of $E_{i,j}$ time units and a worst-case input data transfer time of $C_{i,j}^k$ time units. The different WCET and data transfer time of a task corresponding to different processors are to capture the different speeds of the processors and the input data locality of the task. Communication between each slave and the master is done at every heart-beat of the slave, where each heart-beat interval is of length h time units ($h \geq 1$). The master is assumed to start at time 0, whereas the first heart-beat of the slave P_i occurs at time H_i for all $1 \leq i \leq M$. We assume that scheduling-related messages between the master and the slaves take negligible time. Further, a task is only executed after having acquired all its input data, and tasks allocated to the slot queue are executed in First-In-First-Out (FIFO) order. Note that, unlike in [20], we separate the data transfer time from the computation time to capture the data replacement (i.e., remote vs local) and the interleaving semantics between I/O and computation. The CSP formulation presented in this section follows a relatively simple semantics; its refinement will be discussed in Section 4.3.

Scheduling objective. Given the above MapReduce applications and their execution platform, we would like to synthesize a schedule for the applications such that all jobs with hard deadlines will meet their deadlines. In addition, the schedule either (a) minimizes the number of soft real-time jobs that meet their deadlines, or (b) minimizes the maximum *tardiness* of the soft real-time jobs. Here, the tardiness of a job is the elapse time from its absolute deadline to its finishing time.

CSP encoding. Given the constants described above, the scheduling problem is to determine for each task J_i^k ($1 \le i \le N, 1 \le k \le m_i + r_i$) the values of the following integer variables:

- π_i^k, the slave on which the task is executed;
- γ_i^k, the heart beat at which the task is scheduled to the slave;
- R_i^k, the time at which the input data of the task is fetched to the buffer (for the task execution);
- S_i^k, the time at which the task starts its execution; and
- F_i^k, the time at which the task finishes its execution

such that

(a) $\sum_{N_1 < i \le N} \{1 \mid F_i^{m_i+r_i} > d_i + o_i\}$ is minimized, or

(b) $\max_{N_1 < i \le N} \{F_i^{m_i+r_i} - d_i - o_i\}$ is minimized,

subject to the constraints: for all $1 \le i \le N, 1 \le k \le m_i + r_i$:

(C0). $1 \le \pi_i^k \le M$; (C1). $F_i^k = S_i^k + E_{i,\pi_i^k}$;

(C2). $S_i^k \ge R_i^k + C_{i,\pi_i^k}^k$; (C3). $R_i^k \ge H_{\pi_i^k} + h\gamma_i^k \ge o_i$;

(C4). $\sum\{1 \mid 1 \le i' \le N \wedge 1 \le k' \le m_{i'} + r_{i'} \wedge \pi_{i'}^{k'} = \pi_i^k \wedge S_{i'}^{k'} \le S_i^k \wedge F_{i'}^{k'} > S_i^k\} \le \mathsf{nc}_{\pi_i^k}$;

(C5). $\sum\{1 \mid 1 \le i' \le N \wedge 1 \le k' \le m_{i'} + r_{i'} \wedge \pi_{i'}^{k'} = \pi_i^k \wedge \gamma_{i'}^{k'} \le \gamma_i^k \wedge F_{i'}^{k'} > H_{\pi_i^k} + h\gamma_i^k\} \le \mathsf{ns}_{\pi_i^k}$;

(C6). $\sum\{1 \mid 1 \le i' \le N \wedge 1 \le k' \le m_{i'} + r_{i'} \wedge \pi_{i'}^{k'} = \pi_i^k \wedge R_{i'}^{k'} \ge R_i^k \wedge R_{i'}^{k'} - R_i^k < C_{i,\pi_i^k}^k\} = 1$;

(C7). $F_i^l \le H_{\pi_i^j} + h\gamma_i^j$, $\forall\, 1 \le l < m_i + 1 \le j \le m_i + r_i$;

(C8). $F_j^{m_j+r_j} \le o_j + d_j$ for all $1 \le j \le N_1$.

In the above, constraint (C0) specifies the eligible slaves for each task. (C1) relates the start time and finish time of a task. (C2) specifies that a task can only start executing after all its input data has been fetched to the input buffer. (C3) specifies that a task is only scheduled after it has been released, and its input data is only fetched after it has been assigned to the processor. Here, $H_{\pi_i^k} + h\gamma_i^k$ is the time at which the γ_i^kth heart beat of the slave $P_{\pi_i^k}$ occurs. (C4) limits the number of tasks that can be executing in parallel on a slave to be no more than the number of cores of the slave. Similarly, (C5) limits the number of tasks that are waiting/executing on a slave at any given time to be no more than the number of slots of the slave. (C6) states that the data transfer durations for the tasks are disjoint. (C7) states that a reduce task is only scheduled after all the map tasks of the same job have completed. Finally, (C8) specifies the deadline constraints of jobs with hard deadlines.

One can easily verify that a feasible solution of the encoded CSP indeed corresponds to a feasible schedule of the system. If no solutions exists, the system is not schedulable.

4.2 Implementation and Evaluation Status

We have implemented the presented formulation in Gecode [10]. The initial evaluation shows the solver is able to find solution fairly efficiently for small settings. However, as the number of jobs increases, the system becomes time consuming, given that the formulation is NP-hard. We are currently evaluating different search strategies to speed up the solver. Since the encoded CSP has a finite number of variables, with each having a finite domain, its search space is also finite. Hence, its solution can be found using standard search techniques such as backtracking, constraint propagation, and local search. The following are heuristic approaches that we are currently exploring to improve the search efficiency:

Variable ordering: The timing variables R_i^k, S_i^k and F_i^k are ordered in increasing order of the job's release time (o_i) and subsequently in the order of the task IDs (k).

Value ordering: The timing values of the tasks can be ordered based on their priorities, where tasks with the highest priority are scheduled first. The task priority can be assigned based on the following strategies (ties broken based on the variable ordering): (i) *Earliest Deadline First*, where tasks with smaller absolute deadline are scheduled first; and (ii) *Least Laxity First*, where tasks with smaller slack time are scheduled first. In addition, each task is assigned to an available processor that takes the least total data transfer time and computation time to complete the task. Additional applicable heuristic approaches will be outlined in Section 5.

4.3 CSP Refinement

The above formulation can be refined to capture more precisely the Hadoop implementation by eliminating the assumptions with respect to the following factors:

OS scheduler: In the CSP encoding, we assume that tasks assigned to a processor are executed in the order of their arrivals if there is insufficient available CPU resource (which happens when there are more tasks than cores). Further, all tasks are non-preemptable. In most common OS sharing queuing models, however, these tasks will be co-running and switching among each other in a more complicated fashion depending on the particular OS scheduling (e.g., shortest job first, round robin, priority-based, fair-share, multilevel feedback queue). Our formulation can easily be modified to incorporate preemption and such scheduling mechanism if it can be made deterministic. Note that the resulting formulation will also be more computationally expensive.

Interleaving semantics between computation and I/O: Our formulation requires that at any given time on each slave, at most one task is fetching its input data (from a remote host or from local file system). We can remove this constraint to allow multiple I/O activities to happen at the same time. In this case, the given data transfer time $C_{i,j}^k$ for each task should capture the worst scenario such as in the presence of network and I/O contention, as well as pipelining.

5. DISCUSSION

We briefly outline some of our ongoing work, as well as more speculative (yet intriguing) avenues of future work that apply state-of-the-art techniques in the real-time scheduling literature.

5.1 Ongoing Work

Enhancements to MapReduce's execution model. The actual execution of MapReduce jobs on Hadoop depends not only on the master's scheduling strategy but also on that of the OS scheduling and the slot configuration. As illustrated in Section 3, the higher task parallelism degree the OS scheduler is allowed (by setting the higher slot-to-core ratio) the more unpredictability the task execution experiences. Further, to allow for a simple implementation of the MapReduce programing model, Hadoop separates the map slots and the reduce slots. This separation does not only restrict the scheduling space of the master but also leads to poor utilization of the resource. The latter happens when some reduce-slots are unused due to insufficient reduce tasks that are ready for execution, even though there may be map tasks waiting for execution. Similarly, all map slots are unused after the map tasks in the system have completed.

We are currently modifying the existing Hadoop to remove the separation between the map and reduce slots to maximize the resource utilization. We also configure the number of slots to be the same as the number of cores to add predictability via full control of the master's scheduler.

Heuristics for online scheduling. Although the CSP encoding in Section 4 allows for optimal schedules, its use is restricted due to its static nature and well-known combinatorial complexity. Most real-time cloud computations are often continuous streams of aperiodic jobs that can arrive at any time, and whose characteristics are unknown a priori. In this scenario, we require an online scheduler that dynamically schedules jobs as they arrive. This online scheduling problem in fact a generalization of a simpler problem – the scheduling of a set of independent real-time aperiodic jobs with more than one distinct deadline on more than one machine – which has no optimal solution [14]. The hardness of the problem is made more complex, due to the possibility of machine failures (hence use of speculative execution) and effects of virtualization, both of which complicates WCET analysis.

In the absence of optimal online scheduling algorithms, we are investigating heuristic functions which synthesize characteristics of tasks and resources affecting real-time scheduling decisions. Towards this direction, we explore the prominent existing techniques in the real-time multiprocessor scheduling domain (see [6] for a survey) and adapt them for the cloud setting. Typically, these techniques combine deadlines and resource requirements of the tasks.

In selecting the tasks to be executed next, our initial work looks into the following strategies, with ties broken arbitrarily: (1) earliest deadline first; (2) least laxity first; (3) shortest minimum WCET first (where the minimum WCET is the minimum among the WCET of the task when being executed on the available slaves); (4) shortest total WCET and data transfer time first; (5) randomly chosen first; (6) tasks with density (ratio between WCET and relative deadline) greater than a threshold δ first, and earliest deadline first if no such task exists; and (7) the k tasks with the highest density.

The processor selection is done based one of the following criteria: (a) the first processor on which the task is estimated to finish before its deadline; (b) the processor on which the task is estimated to finish earliest.

We are currently implementing and evaluating these proposed heuristics for MapReduce jobs on Hadoop. Such extensive experimental studies will serve as a basis for designing new heuristic approaches specifically for the online scheduling of real-time cloud computations. Our next objective will be to derive the utilization bound and schedulability test for these heuristic-based algorithms.

5.2 Future Directions

Hierarchical scheduling and real-time virtual machines. In real-time literature, a popular scheduling approach is to utilize a two-level scheduling [19,8] approach to separate critical tasks from non-critical ones. In this approach, dedicated servers are used to control the execution of the soft real-time tasks. These servers are then scheduled alongside the hard real-time tasks by the master. Often, each server is given a fraction of the CPU time that allows sufficient remaining CPU time to guarantee schedulability of the hard real-time tasks. Interesting avenues of research variants of the well-known CPU allocation schemes such as constant bandwidth servers [1], total bandwidth servers [26], and resource kernels [24]. At the same time, we have developed compositional analysis techniques [23, 5] that can be adapted to provide real-time guarantees on virtual machines underlying the cloud computing platform.

Probabilistic models for soft real-time applications. When the system contains only soft real-time applications, our scheduling objective will be to minimize the tardiness bound or the number of jobs missing deadlines. Such soft real-time applications are often implemented using an average-case provisioning; however, our deterministic task model assumes a worst-case system provisioning (e.g., WCET). This is overly pessimistic because WCET could be orders of magnitude greater than average-case execution time. Additionally, WCET estimation is difficult and especially complex on virtual machines and multi-core settings, whereas mean execution time can easily be obtained from observed data.

As such, we would like to explore a less conservative task model based on probability distribution in the absence of hard constraints. Such a probabilistic task model can also be better integrated to the probabilistic nature of machine and software faults. Along this direction, we plan to extend initial probabilistic real-time framework [18] for the online setting and cloud computations.

6. REFERENCES

[1] L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, 1998.

[2] N. Archak, V. S. Mirrokni, and S. Muthukrishnan. Mining advertiser-specific user behavior using adfactors. In *WWW*, 2010.

[3] J.-H. Böse, A. Andrzejak, and M. Högqvist. Beyond online aggregation: parallel and incremental data mining with online map-reduce. In *MDAC*, 2010.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

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

[6] R. I. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical report, Dept. of Computer Science, University of York, 2009.

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

[8] Z. Deng and J.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, 1997.

[9] Q. Gao and S. Vogel. Training phrase-based machine translation models on the cloud: Open source machine translation toolkit chaski. *The Prague Bulletin of Mathematical Linguistics*, (93):37–46, 2010.

[10] Gecode. `http://www.gecode.org/`.

[11] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, 2010.

[12] Hadoop. `http://hadoop.apache.org/`.

[13] D. Hillard, S. Schroedl, E. Manavoglu, H. Raghavan, and C. Leggetter. Improving ad relevance in sponsored search. In *WSDM*, 2010.

[14] K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Trans. Comput.*, 41(10):1326–1331, 1992.

[15] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[16] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. *Lecture Notes in Computer Science*, 5737:341–+, 2009.

[17] S. Matthews and T. Williams. Mrsrf: an efficient mapreduce algorithm for analyzing large collections of evolutionary trees. *BMC Bioinformatics*, 11(Suppl 1):S15, 2010.

[18] A. F. Mills and J. H. Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *RTAS*, 2010.

[19] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS*, 2001.

[20] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD Conference*, pages 507–518, 2010.

[21] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *ICDE*, pages 681–684, 2010.

[22] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

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

[24] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *MMCN*, 2001.

[25] G. S. Sadasivam and G. Baktavatchalam. A novel approach to multiple sequence alignment using hadoop data grids. In *MDAC*, 2010.

[26] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *RTSS*, 1994.

[27] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, pages 1–11, 2009.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[29] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[30] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *CloudCom*, 2009.