2010 39th International Conference on Parallel Processing Workshops

Optimizing Service Level Agreements for Autonomic Cloud Bursting Schedulers

Sriram Kailasam[1] Nathan Gnanasambandam[2] Janakiram Dharanipragada[1] Naveen Sharma[2]

[1]*Distributed and Object Systems Lab*
Dept. of Comp. Sci. and Engg., IIT Madras
Chennai, India 600036
{*ksriram@cse, djram@*}*.iitm.ac.in*

[2]*Xerox Research Center Webster*
Xerox Corporation
Webster, NY, USA 14580
{*nathang, naveen.sharma*}*@xerox.com*

***Abstract*—The practice of computing across two or more data centers separated by the Internet is growing in popularity due to an explosion in scalable computing demands and pay-as-you-go schemes offered on the cloud. While cloud-bursting is addressing this process of scaling up and down across data centers (i.e. between private and public clouds), offering service level guarantees, is a challenge for inter-cloud computation, particularly for best-effort traffic and large files. The parallel workload we address is real-time and involves inter-cloud processing and analysis of images and documents. In our production printing domain, dedicated processing/network resources are cost-prohibitive. Further, the problem is exacerbated by data intensive computing - we encounter huge file sizes atypical of intercloud parallel processing. To address these problems we propose three flavors of autonomic cloud-bursting schedulers that offer probabilistic guarantees on service levels required by customers (such as speed-up and queue sequence preservation) by adapting to changing workload characteristics, variation in bandwidth and available resources. In particular, these opportunistic schedulers use a quadratic response surface model for processing time in concert with a time-of-day dependent bandwidth predictor to increase the throughput and utilization while simultaneously reducing out-of-sequence completions for a document processing workload.**

I. Introduction

Cloud Bursting integrates a private cloud (existing corporate infrastructure) with a public cloud to create a highly scalable computing platform (see Figure 1). This practice of computing across two or more data centers (internal plus external) has been growing in popularity because of the scalability and pay-as-you-go benefits it offers with respect to data intensive computing (see [1]). In particular, using cloud bursting for overflow parallel processing has also been possible more recently through preconfigured instances on public clouds [2]. These practices have tremendous advantages for large and small organizations alike as they need to provision only for average demand with the cloud fulfilling the remainder of the workload. Usually, the data center resources inside the production facility is referred as internal cloud (IC) while those outside the facility (accessed through Internet) are referred as external cloud (EC) [3].

We shall now briefly discuss some of the interesting scenarios for cloud bursting and get to our problem. Typically, the capacity in the IC is fixed (static) while it may be varied in the EC (elastic). In future (with the increasing number of cloud providers), one could possibly choose from a pool of Cloud Providers at run-time depending on the input job's service level agreements (SLAs). Thus the possible interesting scenarios are a combination of Static/Elastic IC with Single/Multiple EC having Static/Elastic nature. In this paper, we consider scheduling across Static IC and Single Static EC, for applications with heavy bandwidth and processing requirements. For such applications, transferring the inputs (and outputs) across clouds becomes comparable to the processing time and hence splitting the workload across clouds becomes a non-trivial task. Unlike our domain, certain kinds of scientific computing (e.g. [1] [4]), business analytics and software-as-a-service applications may involve relatively light-weight inputs followed by a lengthy cloud-appropriate computation.

We are focussing on a domain that needs a certain resident computational capacity (internal cloud) in addition to overflow computational capacity on a public cloud (such as Amazon EC2, Rackspace etc.). This notion is also known as hybrid clouds (see [3] [5]) where performing suitable proportions of computation remotely and locally (i.e. within a private cloud, enterprise or small business) may be for reasons of redundancy, security and reliability (see [6]). But a more important reason for the purposes of our investigation is that our workloads are real-time production workloads whose transfer time to the external cloud, is comparable to their computational time. Therefore, if the local resources remain idle while transferring inputs to a remote cloud, it adversely affects the throughput and hence service level agreements. This means that some computation must ideally be carried out continuously at both ends while transfers (and subsequent processing) are optimally scheduled with respect to deadlines or chronological priority. To this end, we propose three variants of inter-cloud scheduling heuristics that optimize various guarantees on service while catering to a parallelizable workload.

Another advantage of considering hybrid clouds is that remote computation can completely be scaled down during periods of low demand without incurring processing or more importantly, bandwidth costs. This feature is of value to workloads that are real-time, time-varying/seasonal and

1530-2016/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPPW.2010.54

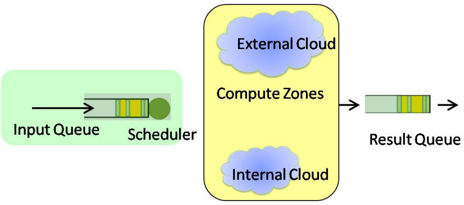

Figure 1. Cloud Bursting Conceptual Diagram

consisting of large datasets such as high-resolution images, reconnaissance imaging or medical documents. During such lean periods, it may be optimal to carry out all the processing on the private cloud with a small set of resources while respecting service level agreements. To this end, the second objective of this paper, is to demonstrate the viability of real-time processing of bandwidth-intensive datasets opportunistically on the hybrid cloud while respecting service level agreements.

To adhere to the SLAs, the schedulers must be able to learn the round trip latencies including processing time on the other cloud for different job sizes and bandwidth availability. We learn the processing and transmission times using a quadratic response surface model and a time-of-day dependent bandwidth predictor, and subsequently use the estimates to decide which job in the queue needs to be bursted out. Chief among SLAs we consider for these long-running (relative to, say, HTTP workload) jobs are the Out-of-Order (OO) metric and makespan (we elaborate further on SLAs in Section 2). Minimizing out-of-order jobs guarantees that the jobs preserve their slot in the result queue. Jobs are given a ticket that they will finish a certain number of seconds from their submission point. Thus the OO metric is directly correlated to whether or not the expectation of the ticket-holder (human or machine) will be met. However, in this distributed inter-cloud setting, minimizing out-of-order jobs imposes a constraint that adversely affects makespan.

The remainder of the paper is organized as follows. We first examine the domain characteristics and the properties of the datasets we deal with and outline the salient contributions of this work. In the next section, we elaborate on the concept of slackness constraints (or expected time cushions) of jobs in a queue that have to be bursted out to an external compute cloud. In light of these conditions, we discuss how cloud-bursting is carried out opportunistically in Section 3 and reiterate the need for resident local computational capacity in domains where large file transfers are involved. In Section 4, we discuss three variants of cloud bursting schedulers. We describe our empirical evaluation using real workload from a document processing domain in Section 5 and contrast our work with the literature in Section 6. We conclude with ideas for future work in Section 7.

Domain characteristics

We consider a production facility (or factory) that handles large volumes of document workload namely newspapers, books, marketing material, mail campaigns, credit card statements, image personalization, variable data promotional material etc. The computational workload of such document organizations may be image processing, image enhancements, raster processing, text/image optimizations for enhancing the quality of printed outputs and several business processes that naturally lend themselves to parallel processing. Given the volume of the workload these computational processes are time consuming. Typically, these are carried out on private clouds (small/medium businesses) but these local data centers are not provisioned for peak workload owing to cost. The workloads also wildly fluctuate and are periodical (weekly, monthly, yearly etc.) closely following the seasonal consumption patterns of a consumer economy. The overall length of image processing per job could depend on the input document sizes, quality requirements, workflow stages, output formats and so on. Subsequent to computation, physical production activities follow where penalties for under-utilization are heavy. This mandates that the computational workload performed upstream should strictly adhere to the SLAs dictated at every stage.

Contributions

One of the contributions of this paper is that we offer evidence to support the viability of real-time cloud-bursting while transferring extremely large sizes of workload back and forth over the best-effort transport structure of the regular Internet. In addition, we focus on techniques for delivering service level guarantees during cloud-bursting for a computational workload on documents and images. While there has been some work on efficient brokering strategies across clusters (see [7]), scheduling techniques and learning models for inter-cloud processing needs further research. The proposed slackness constraints with respect to the workload queues offer a new way to balance chronological priority with other traditional service level agreements (such as speed-up, make-span, utilization etc.) in support of small/medium organizations that are interested in utilizing cloud bursting effectively. The cloud-bursting techniques and results presented in this paper are applicable to a number of domains including academic computing environments with multiple types of jobs. To this end, one of the questions that prominently features in this paper is *given a workload, how do we determine when (a scheduler decision under resource variation), where (to which cloud) and how much (the quantum of work) to burst out for optimizing certain downstream service level attributes.* We discuss these, and the opportunistic and autonomic nature of our schedulers in the sections below.

II. Constraints and SLA for Job Queue

A. Slackness Constraint

A key consideration in the design of our inter-cloud scheduling mechanisms is the notion of slackness associated

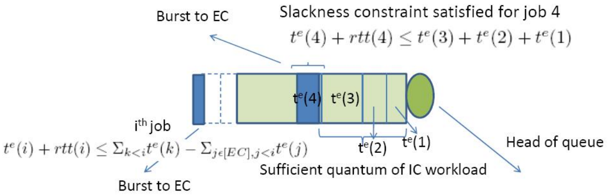
Burst to EC
Slackness constraint satisfied for job 4
$t^e(4) + rtt(4) \leq t^e(3) + t^e(2) + t^e(1)$
$t^e(4)$
$t^e(3)$
i^{th} job
$t^e(i) + rtt(i) \leq \Sigma_{k<i} t^e(k) - \Sigma_{j \epsilon [EC], j<i} t^e(j)$
$t^e(2)$
$t^e(1)$
Head of queue
Sufficient quantum of IC workload
Burst to EC

Figure 2. Slackness constraint for Single Resource in IC

with jobs waiting in a central queue for processing. Informally, slackness refers to time cushions available to certain jobs to make a round trip to an external compute cloud (EC) before their turn for local processing arrives (FCFS queue). These slackness values are a function of current job's characteristics and position in the queue, as well as the attributes of jobs preceding it. As shown in figure 2, the fourth job (j_4) can complete processing in the external cloud before it is required (by the downstream production stages that assume task arrival chronology) and hence j_4 can be bursted out. The jobs are all assumed to originate from the internal cloud. The input and output sizes of jobs are substantial — often hundreds of megabytes even upon data compression. So on the regular Internet these would consume a significant amount of transfer time back and forth from the external cloud.

The slackness considerations we outline herein provide a general framework to model the tolerances associated with a scheduling decision and their subsequent impact on service level attributes. In particular, these considerations are critical when transfer overheads are significant (due to file sizes or bandwidth variations) - how we burst out to an external cloud from an internal cloud is influenced by the huge sizes of jobs we deal with. We now formally define the slackness constraints. Let there be a list of jobs J in the queue where the i^{th} job is denoted j_i. We define the following:

$t_0(i), t_c(i)$	start time, end time of j_i
$t^e(i)$	estimated computation time of j_i on a standard machine
s_i (o_i)	input (output) size of j_i
$l(t)$	data transfer rate to/fro the external cloud at time t (for asymmetric links define upload and download rates)
$\text{ft}^{ec}(i, S)$ ($\text{ft}^{ic}(i, S)$)	finish time estimate of j_i in EC (IC) when the system state is S
d_i	decision variable indicating placement of j_i (0 means IC; 1 means EC)
$t^e_c(i)$	estimated completion time of j_i (depending on where it was scheduled) $= d_i * ft^{ec}(i) + (1 - d_i) * ft^{ic}(i)$
T_i	estimated completion times of the first i jobs $\{x \vert x = t^e_c(i')$ and $i' < i\}$

The slack time for j_i is given as:

$$slack(j_i) = max(T_i) \tag{1}$$

In other words, slack is the time cushion of the first job from the head of queue or a previously identified position whose estimated completion time in the external cloud could be greater or equal to the completion times of the jobs preceding it in the internal cloud. Further, these slackness conditions allow for a quantum of work to be available locally in the IC, so that the job that gets bursted out has enough lead-time to make the round trip in addition to getting remotely processed. The constraint, therefore, is that the

$$slack(j_i) \geq t^e(i) + s_i/l(t_i) + o_i/l(t_i + t') \tag{2}$$

where t_i indicates the time at which upload to EC begins and t' is the time at which the job result gets downloaded. In practice, slackness constraints are checked for every pair of resources across clouds.

B. Out-of-Order Constraints

Even though there is a single queue, jobs and portions thereof may execute in parallel. As a result, the jobs that are processed internally or externally may complete ahead of their natural position in the queue. This could violate a first-come first served (FCFS) policy that customers would value, particularly when the total processing time may be of the order of tens of minutes. Therefore we define the Out-of-Order (OO) metric by considering the position at which jobs complete along with their output sizes (the operational rate of the subsequent production stages like printer or workflow processing depends on the size of the job output). Let

S	ordered sequence of sampling times
s_t	t^{th} sampling time
C_t	the subset of jobs that completed before s_t
J_{it}	the subset of jobs in C_t whose id is less than or equal to i
t_l	tolerance limit for the out-of-order completion of jobs
m_t	max id of the job in C_t satisfying out-of-order constraints
o_t	size of ordered data (within tolerable ordering limit t_l) available at time s_t

Further define

$$C_t = \{x \vert x \epsilon J \wedge t_c(x) \leq s_t\} \tag{3}$$

$$J_{it} = \{x \vert x \epsilon C_t \wedge x.id \leq i\} \tag{4}$$

Now we are interested in finding the highest value of job id (say i) in the result queue (at time s_t) upto which the results (ordered according to job id) can be consumed by the next stage without violating the tolerable limits of processing in order. For example, a tolerable limit of 0 (strict order) means that every job whose id is less than i must have been processed. Formally, the max job id satisfying out-of-order constraints is given as

$$m_t = \text{find } \max_i \ s.t. \ j_i \epsilon C_t \wedge i - t_l \leq \vert J_{it} \vert \tag{5}$$

The cumulative size of ordered data available for the next stage following processing at time s_t is given as

$$o_t = \sum_{x \epsilon J_{it} \wedge i = m_t} x.size \tag{6}$$

If we were to consider that the next stage after document processing is a printer, then the above value would indicate the amount of data (ordered) ready to be consumed by the printer at time s_t while maintaining the ordering constraints. Thus the OO metric (see equation 6) captures the out-of-order completion of jobs by computing a function (sum of the output size of the result) over the jobs that complete in order (within tolerable limits).

C. Other Service Level Agreements

Some other SLA definitions from our domain are as under:

1) **Makespan**: Makespan is defined as the total time taken to run the entire set of jobs. Let $arr(J)$ denote the arrival time. Therefore makespan C

$$C = max([t_c(i)]) - arr(J) \quad i|j_i \epsilon J \tag{7}$$

Note that jobs may complete in any order, requiring the max operator. The objective is to minimize the makespan.

2) **Utilization**: Let M denote the set of machines used during the job run and let $ru_m(J)$ denote the running time of a particular machine (m) during the total run time of the job set. Then the utilization ($u_m(J)$) of a particular machine (m) during the execution of the job set (J) is defined as

$$u_m(J) = \frac{ru_m(J)}{C_j} \tag{8}$$

and the average utilization over a set of machines M (belonging to IC or EC) is defined as

$$u_M(J) = \frac{ru_M(J)}{|M| * C} \tag{9}$$

3) **Speedup**: Given a set of jobs, speedup (s) is defined as the ratio of the total time taken to run the set of jobs sequentially on a standard (set of) machine(s) to the time taken to run it using the cloud bursting approach. Speedup for the entire run:

$$s = \frac{C}{t_{seq}(J)} \tag{10}$$

The speedup measures how fast the jobs completed execution. So the objective is to maximize the speedup. Improving the utilization of the system (homogeneous) has a direct bearing on the speedup.

4) **Burst ratio**: The burst ratio ($bu(J)$) is the ratio of the number of jobs in the job set J that were bursted out to the total number of jobs in J.

Burst ratio j^{th} batch (B_j):

$$bu(B_j) = \frac{\sum_{i|j_i \epsilon B_j} d_i}{b_j} \tag{11}$$

Burst ratio for entire run:

$$bu(B) = \frac{\sum_{j|B_j \epsilon B} (bu(B_j) * b_j)}{n} \tag{12}$$

This ratio gives the proportion of jobs that are bursted out (with time) for different schedulers and is an indicator of the resource utilization in EC.

These metrics are used to compare the performance of different schedulers. With respect to real-time processing, we assume that we need to choose jobs to burst out from as near the head of the queue as possible (note that just bursting out from the head of the queue violates several SLAs). This constraint on bursting not-only respects the FCFS discipline, it also minimizes the OO metric defined above. In other words, we cannot resort to the option of jobs near the head of the queue being computed locally while the tail region is processed by the external cloud (because it will compromise speed-up of the initial batches as well as other SLAs).

III. Opportunistic Bursting

Due to the huge size characteristics of our workload, EC resources should be opportunistically used along with the locally resident computation capability. Some amount of processing must be carried out on the IC at all times because — (a) with transfer times being of the order of processing time, it is possible to perform a quantum of work at the head of the queue while the transfers are happening. (Precisely which job is chosen is decided by the slackness conditions.) (b) continuously using bandwidth and resources on the external cloud is not cost-efficient at times of low workload. Therefore opportunistic bursting requires the use of learned models, dynamic calibration and benchmarking for effectively utilizing the processing capacity of the inter-cloud system and the offered bandwidth. While these estimation models may frequently have accuracy shortfalls, even imprecise estimates of remaining (long-tailed) workload have been shown to have merit in grid scheduling relative to a random scheduler (see [8]).

A. System Models

The capability of a scheduler to honor slackness constraints while making assessments based on learned models differentiates this work from the literature (e.g. [1] [7]). Thus, the system estimates the finish times in IC and EC considering the current load, the expected run times of the jobs (processing time estimates) and the expected bandwidth usages for upload/download of the job/result.

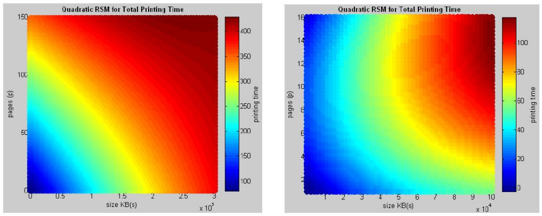

Figure 3. Quadratic Response Surface Models for Processing Time

1) Processing time model: A quadratic response surface model (QRSM) (see [9]) was used and subsequently tuned by observing data from the actual system. We start with an initial best estimate model based on a standard set of production data observed across a variety of locations and subsequently learn and tune the model depending on the specific conditions and resources available. A quadratic response surface model would assume that a quadratic polynomial f would relate y to the N independent variables considered i.e.

$$y = f(x_1, x_2, ..., x_N)$$

More specifically,

$$y = a + \sum_{i=1}^{N} b_i x_i + \sum_{i,j=1; i \neq j}^{N} c_{ij} x_i x_j + \sum_{i=1}^{N} d_i x_i^2$$

The coefficients (a, b_i, c_{ij}, d_i) for $i, j = 1$ to N and $i \neq j$ are learnt as the solution to a linear programming model. The variable y corresponds to processing time on a given resource. The dimensions x_i are the important features of documents and images — namely document size, number of images, the size of the images, number of images per page, resolution, color and monochrome elements, image features, number of pages, ratio of text to pages, coverage, specific job type etc. From the above, a relevant set of features are extracted and utilized for every job type. We show in Figure 3 the QRSM model for processing time from our experiments. Without loss of generality we can add or delete dimensions to our model as may be required by the specific job class. Learning and tuning of the model depending on the job class is part of future work.

2) Transit time model: The autonomic system also captures network conditions, calibrates its settings and modulates the cloud-burst engine's network activities. The upload and the download bandwidth from an arbitrary internal cloud to the external cloud vary sporadically because of factors such as last-hop latency, time-of-day variations, bandwidth throttling, unavailability of higher capacity/bandwidth lines etc. Since the application we consider is extremely data intensive, we particularly adapt to Internet conditions by estimating the effective bandwidth.

Figure 4(a) shows the variation of bandwidth across different times of a day. This is calibrated automatically and learned for every location and the time of day they operate. This can further depend on the seasonality of the particular IC's demand. We experimentally determine a certain number of threads for downloading/uploading a file in parallel at a given point of time that can maximize the bandwidth utilization. Figure 4(b) shows the number of threads that were used to maximize the bandwidth utilization.

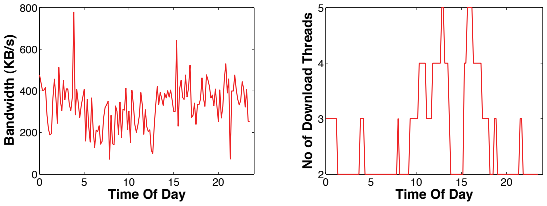

Figure 4. (a) Time of day model for bandwidth variation (b) number of threads used for keeping the upload/download pipes fully utilized

The effective bandwidth is measured at different times of the day by periodic test uploads/downloads of size 1MB from the internal to the external cloud. This is used in conjunction with the actual values of the upload/download times observed during the experiment. The network estimation model is updated according to the following equation:

$$S_n = \alpha . Y_n + (1 - \alpha) . S_{n-1}$$

where,

$S_n =$ weighted moving average for the network speed after the n^{th} file upload / download

$Y_n =$ network speed during the n^{th} measurement

$\alpha =$ weight

B. Cloud Bursting Architecture

The cloud-bursting architecture is pipelined and event-based. Pipelining helps to squeeze out more throughput from the system because every stage of the pipeline is executed in parallel, as opposed to only processing in parallel. The event-based nature of the model keeps the different stages loosely coupled. The overall architecture is explained in Figure 5 — (1) The user submits the job through a web interface to the system; (2) The web-server then places these jobs into a job queue; (3, 4) The job queue is continuously monitored and the job gets picked by the Controller (scheduler). The controller parses the job and invokes the scheduler. The scheduler may decide to split the job and hence portions of the job may be migrated to the EC. (5) Individual cloud controllers take charge (internal or external) (6) Dispatches to either cloud (7) Job gets processed in either internal cloud and/or external cloud (8) Job is retrieved and the user collects the output. The pipelined architecture can be thought of as a network of asynchronous queues - upload, execution, download queues and job moves from one queue to other.

As part of opportunistic decision making, the scheduler estimates the different parameters of the job using the estimation models. The aforementioned models for processing

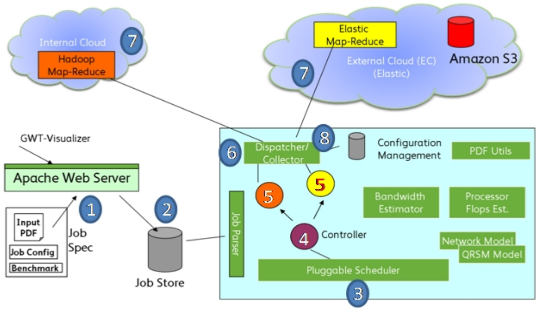

Figure 5. Cloud Bursting Architecture

time and offered bandwidth are utilized to estimate the parameters required for bursting (such as the number of threads, round-trip-time or processing time on a resource).

In the prototype, the internal cloud uses a Hadoop Map-Reduce ([10]) cluster formed by the printer controllers, whereas the external cloud uses Amazon S3 as the job store and the Elastic Map-Reduce for computation. The compute intensive tasks are expressed using the map-reduce paradigm to exploit the Hadoop Map-Reduce cluster available in the internal and the external clouds. The jobs are embarrassingly parallel and hence splitting them and scheduling them in different clouds does not introduce any inter-cloud communication (while the computation is going on) apart from the final merge of the results. After computation, the job output is compressed and downloaded. And finally it is added to the result queue.

IV. Burst Scheduler Design

As mentioned earlier, one of the questions that prominently features in this paper is *given a workload, how do we determine, when, where and how much to burst out for optimizing certain downstream service level attributes*. Here, ”when” refers to the precise time the scheduler makes the decision, ”where” refers to the choice of one or more public clouds, and ”how much” refers to the quantum of workload bursted out. The goal of the scheduler is to recursively pick the right job from the head of the queue or the previously chosen job, that if *cloud-bursted* (sent externally) would not delay the rest of the jobs in the queue i.e. the output of a job that is bursted would be required only a small time τ before the jobs preceding it complete in the internal cloud. In practice this is hard to achieve given the uncertainties in resource availability, Internet congestion and estimation errors. In this section we discuss designs of three schedulers. These schedulers only look at the current state of the system to make decisions on splitting and placement of jobs. Hence they are traffic oblivious (the estimation models are used to predict the job execution time and transfer time given the current load in the system).

A. Greedy Scheduler

This scheduler makes a job-level greedy decision – schedules the job (in IC or EC) where it is expected to complete earliest. The aforementioned decision factors the expected execution time/bandwidth requirements for the job using the estimation models, the current transit bandwidth and the resource availability in both the clouds. The pseudo-code is shown in Algorithm 1.

Algorithm 1: Greedy Scheduler

```
Data: $J$: list of jobs in a given batch
Result: $j_i \longrightarrow (EC$ or $IC) \; \forall i$: assignment decision
for $i = 1$ to $|J|$ do
    $t_{ic} \longleftarrow ft^{ic}(j_i)$
    $t_{ec} \longleftarrow ft^{ec}(j_i)$
    if $t_{ic} \leq t_{ec}$ then
        Schedule $j_i$ in IC
    else
        Schedule $j_i$ in EC
    end
    $i \longleftarrow i + 1$
end
```

B. Order Preserving Scheduler

The motivation for this scheduler is that the jobs must complete more or less in the order of arrival with the added constraint that no internal job waits for the results from the bursted out job. Recall from Section II, if the scheduling decision respects slackness constraints, then the jobs scheduled in EC are never on the critical path. Thus the slackness criteria automatically reduces the probability of waiting for results of bursted jobs for later positions in the queue and makes it more robust under network variation. The pseudo-code is shown in Algorithm 2. This algorithm tries to minimize out-of-order completion of jobs as follows: First, it reduces the variation in the job sizes by chunking the large job into smaller jobs and adding them as new jobs in the job-list(lines 3-10). Next, those jobs that satisfy the slack condition are bursted out and the rest are scheduled in IC.

C. Scheduler Optimizations

We now propose some enhancements to the above mentioned schedulers that are needed with large file transfers and long-tailed workload. These optimizations aid queue sequence preservation, resource utilization and/or scalability for the inter-cloud parallel computation.

Size-interval based Bandwidth Splitting: The workload consists of jobs whose sizes are highly variable. So the upload of a large job can block several smaller jobs thereby decreasing the utilization and throughput of jobs in the EC. Therefore we hypothesize that - size-interval based splitting (see [8]) for available bandwidth would improve the utilization of EC by moving jobs faster to EC. Instead of simply increasing the number of queues we partition the upload tasks into size intervals — namely small, medium and large

Algorithm 2: Order Preserving Scheduler

```
Data: $J$: list of jobs in a given batch
Result: $j_i \longrightarrow (EC$ or $IC)\ \forall i$: assignment decision
$size \longleftarrow |J|$
$i \longleftarrow 1$
while $i \leq size$ do
    $v \longleftarrow \sigma(i : i + x)$
    if $v > th$ then
        $C \longleftarrow$pdfchunk($j_i, v$)
        $J$.remove($i$)
        $J$.insert($i, C$)
        $size \longleftarrow size + |C| - 1$
    end
    $t_{ec} \longleftarrow ft^{ec}(j_i)$
    if $t_{ec} \leq$ slack($J, i$) then
        Schedule $j_i$ in EC
    else
        Schedule $j_i$ in IC
    end
    $i \longleftarrow i + 1$
end
```

buckets. This effectively isolates the small jobs from the large jobs and decreases the variance in each bucket, thereby improving the utilization of the EC. We determine the upper bounds of each of the upload queues as detailed in Algorithm 3. Here lines 3-12 identify the potential jobs that can be bursted out (jobs that may satisfy the slack condition) and stores them in list L. Line 13 computes the normalized left-over capacity for the upload queues. Next, the sorted list L is partitioned according to the ratio of the left-over capacity thereby equalizing the network load across the different queues (lines 15-17). In practice, some of the potential jobs that are identified as large/medium may not be bursted out as the upload queue fills up. Therefore, our policy is to allow jobs in the lower queue to get uploaded via higher queues as well, to maximize the bandwidth usage.

Algorithm 3: Size-interval based Bandwidth Splitting

```
   Data: $B$: ordered list of jobs in a given batch, $iload$: initial compute load in
         IC, $s_{up}, m_{up}, l_{up}$: size of data to be uploaded in the small, medium
         and large queues, $n$: number of processors in IC
   Result: $s_{bound}, m_{bound}$: size-interval bounds for small and medium queues
1  $L \longleftarrow \emptyset$
2  $rload \longleftarrow 0$
3  for $i \leftarrow 1$ to $|B|$ do
4      $job \longleftarrow B$.get($i$)
       // Completion time in EC under no load
5      $t_{ec} \longleftarrow job.t_{up} + job.e_{ec} + job.t_{down}$
6      if $t_{ec} < iload + \frac{rload}{n}$ then
7          $L \longleftarrow L \cup job.size$
8      else
9          $rload \longleftarrow rload + job.e_{ic}$
10     end
11     $i \longleftarrow i + 1$
12 end
13 $s \longleftarrow 1 - \frac{s_{up}}{s_{up}+m_{up}+l_{up}}$, $m \longleftarrow 1 - \frac{m_{up}}{s_{up}+m_{up}+l_{up}}$,
   $l \longleftarrow 1 - m - s$
14 $L$.sort()
15 $[L_s, L_m, L_l] \longleftarrow L$.partition($s, m, l$)
16 $s_{bound} \longleftarrow L_s$.last()
17 $m_{bound} \longleftarrow L_m$.last()
```

D. Discussion

We now contrast the design principles of the different schedulers.

Greedy Scheduler: This scheduler makes a simple job-level greedy choice that can cause bursted out jobs to end up in the critical path. This makes the schedule vulnerable to estimation errors and fluctuations in the network bandwidth. Some of the undesirable outcomes are listed below:

- Job order in result queue is shuffled (sometimes highly out-of-order)
- Making a greedy decision to push out and pull jobs in, based on the transient value of bandwidth, imposes the risk that at download time the bandwidth in reality is lower than the initial estimate. This may cause poor performance.

Order Preserving Scheduler: Though in principle this scheduler is more robust to network fluctuations due to the slackness criteria, the errors in the estimation of the job execution time can affect the schedule. For instance, an overestimation of the job's execution time that takes max time in IC (refer Equation 1 used to compute slack) or an underestimation of the bursted out job can cause out-of-order completions. The former case can cause extra jobs to be bursted to the External Cloud. If this were to happen towards the end of the run, it will increase the makespan (due to jobs that are scheduled in EC). The latter case would result in EC completing the jobs much before hand and remaining idle while IC is still executing. The current QRSM model occasionally overestimates the execution time of jobs. Improvements to these models is part of future work. Despite improvements, errors are common in this domain wherein the multitude of features within a document contribute to the total processing time.

Therefore, we need periodic rescheduling strategies to be triggered when the IC or EC becomes idle. For instance, when a resource in IC becomes free it picks up a job from the head of the EC queue such that the remaining time for it to complete is greater than the time it would take to re-execute the same in the internal cloud. Similarly, when the EC upload queue is idle and IC has jobs waiting to execute, then we scan the IC wait queue from the last and check if there is any job that satisfies the slack criteria. Then that job is pulled from IC and scheduled in EC. These strategies can mitigate the estimation errors and are part of future work.

Order Preserving Scheduler with Size-interval Bandwidth Splitting: The finer points of Size-interval Bandwidth Splitting are described below:

- it improves the utilization of the upload bandwidth by using parallel threads for upload (as detected by the network model to be optimum)
- it equalizes the load across the different upload queues

- it isolates the small jobs from the large jobs thereby increasing the job arrival rate in EC
- it allows a lower sized job to travel through an upper sized job queue to EC

When the variability in the job sizes is high, size-interval splitting is the most useful as discussed in [8]. Our approach is slightly different from them. We allow lower sized jobs to travel through higher sized job queue to EC. But we do not allow higher sized jobs to travel through lower sized job queue. While this approach maintains isolation of small jobs from large ones, it favors smaller jobs to move faster to EC thereby facilitating better utilization of EC. When the job size variability is low, the behavior of size-interval splitting defaults to that of having a single interval.

V. Experimental Evaluation

We compared the schedulers against each other and observed the relative performance in terms of completion times, ordered data output availability, speed-up, resource utilization and burst ratio by using three samplings (explained below) from real production workload. The experimental observations suggested a few potential optimizations to the aforementioned schedulers which we summarize towards the end of this section.

A. Experiment Set-up

The experiments were carried out using a test-bed that consisted of 8 virtual machines forming the internal cloud and a maximum of 2 virtual machines forming the external cloud (i.e. Amazon EMR [2]). The process varies the number of download/upload threads and converges upon the optimum number of threads to be used for that time-period. Thus the different portions of the job(s) and result(s) are downloaded in parallel, using multiple threads to maximize the bandwidth utilization to/from the external cloud. Next, we created three buckets from the production jobs that were considered. These jobs were production quality documents consisting of images and text varying in size from 1MB to 300MB. The first bucket was biased towards small jobs; the second one had a uniform distribution of job sizes, while the last one was biased towards large jobs. In the experiment, a batch of jobs from a particular bucket would arrive every 3 minutes according to a poisson process with mean arrival rate $\lambda = 15$ per batch. The schedulers then attempt to complete as many as possible in the least time (maintaining other SLAs).

B. Empirical Evaluation

1) Performance improvement with Cloud Bursting: Figure 6 shows that Cloudbursting improves the performance by 10 percent over IC-only scheduler (average network speed=250kbps, EC instances = 2). While the makespan for the greedy and the order-preserving scheduler is almost same, the difference between the two schedulers is illustrated in the order of completion of jobs (see Figure 7). *We have abbreviated Order Preserving Scheduler as Op in the legend for the graphs*. A general observation from Figure 7 is that the Greedy scheduler shows more number of high peaks (in magnitude as well) while there are more number of valleys in the Order Preserving scheduler. This effect is amplified in the case of distribution biased towards large jobs (see Figure 8). A high peak means that the job is not available for processing when it is required (or in other words it induces a wait period due to the requirement of in-order processing) and its magnitude indicates the amount of wait time. A valley means that the job output is available before it is consumed and is not a problem. From Figure 7 and 8, we conclude that scheduling according to the slackness criteria reduces the chance of an internal job waiting for the results from an external job and hence is more robust to network variations/errors.

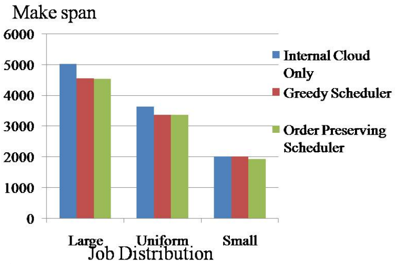

Figure 6. Comparison of Makespan for different schedulers

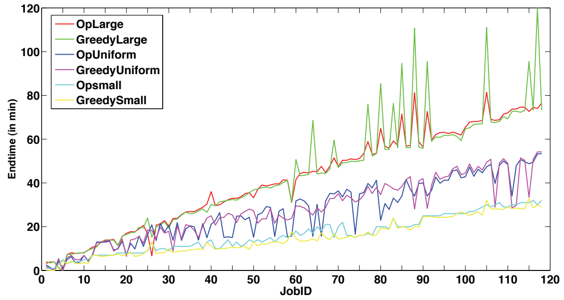

Figure 7. Completion Times for Large Uniform and Small Job-size distribution

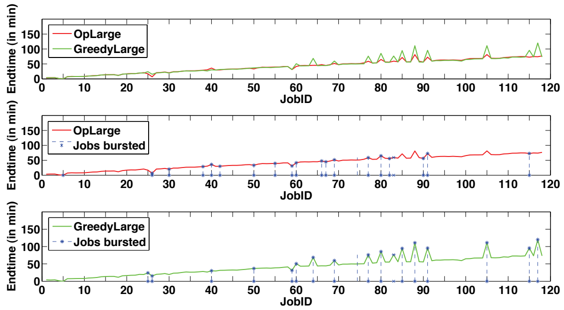

Figure 8. Completion Times for Large Job-size distribution

2) Comparing OO related metrics: Figure 9 shows that the OO metric (sampling interval is 2min) for large jobs (bucket) under high network variation in case of Order Preserving scheduler is greater than the Greedy scheduler.

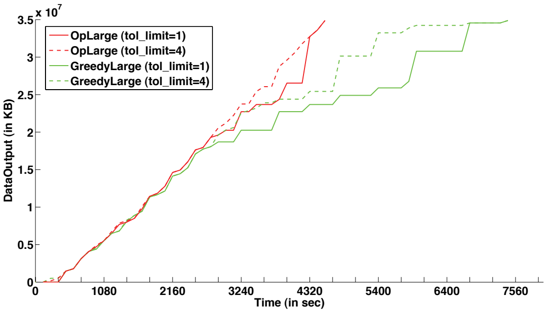

Figure 9. Data Output Size available at different time intervals for *large* distribution under high network variation

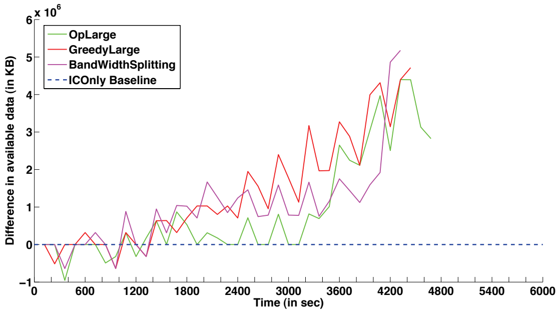

Figure 10. Comparison of relative difference of OO metric w.r.t. ICOnly Scheduler with $tol\_limit = 4$, job distribution= *large*

This implies that the future downstream activities that expect data in chronological order can operate at higher rates for the Order Preserving scheduler. Basically, more the number of high peaks, more is the wait period and therefore it follows from figure 8 as to why the Greedy scheduler suffers. Now, increasing the tolerance limit increases the data output availability, but at the cost of more out of order completions. Thus the tolerance limit can be considered as a tradeoff parameter between data output availability and ordering requirement, and may be specified according to the application requirements.

Figure 10 plots the relative difference of the OO metric for a given scheduler w.r.t. IC-only scheduler (treated as baseline). We observe that the Order Preserving scheduler and the Size Interval Bandwidth Splitting scheduler show higher OO metric w.r.t. the Greedy scheduler (almost at all points of time). The Bandwidth Splitting scheduler is baised towards smaller jobs while pushing to the EC. Hence, the OO metric suffers if many small jobs are scheduled to EC followed by a large job. Favoring smaller jobs however improves the EC utilization thereby reducing the makespan. Hence, we see a sharp increase in the data availability for the Bandwidth splitting scheduler towards the end of the execution time (This happens upon completion of the large job).

3) Comparing Makespan, Speed-up, Utilization and Burst ratio: From Table I, we observe that the speedup depends on the system utilization as well as the computation to communication overhead. In case of large jobs the average computation time is higher than the network delivery time. Hence the execution unit is more utilized. Whereas in the other two cases the average computation time drops down and the network delay assumes more significance (affecting the rate at which jobs can be delivered to the cloud for processing). Hence, we obtain a higher speedup in the case of large jobs. We also observe that a slight decrease in the IC utilization (look at Op-uniform) along with an improvement in the burst ratio (higher EC utilization), keeps the speedup intact. Thus better bursting decisions could potentially improve the utilization of EC and IC thereby increasing the speedup.

	IC-Util		EC-Util		Burst-ratio		Speedup	
	Greedy	Op	Greedy	Op	Greedy	Op	Greedy	Op
Large	78.6	81	45.8	44	0.19	0.17	6.73	6.76
Uniform	82.42	74.42	17.71	46.57	0.17	0.26	5.6	5.6

Table I
PERFORMANCE METRICS

4) Potential Optimizations: The coefficient of variation in the job sizes for the bursted jobs (per batch) is close to 1. This suggests that size-interval based splitting (see [8]) of bandwidth can improve the rate of delivery of jobs to EC, thereby improving the EC utilization. Indeed this was confirmed when the size-interval bandwidth splitting optimization was applied to the Order Preserving scheduler. The EC utilization increased to 58%, IC utilization was about 81% for the large job size distribution, and the speedup increased by 2% over the Order Preserving scheduler.

The Cloud Bursting efficiency can be improved by keeping the pipeline full. Due to the data intensive nature of the jobs, the scaling (at EC) must be just enough to ensure saturation of the download bandwidth. Such scaling policies forms part of future work.

VI. RELATED WORK

While cloud-bursting is emerging as an active research area for computing on a hybrid cloud (as defined in [3]), the focus has been on computational needs being effectively outsourced or balanced by an elastic cloud that could be rented on demand. To this end, industry vendors have various product offerings ranging from out-sourced email or storage [11] to elastic parallel computing [2]. These offerings make the assumption that workload is light or incremental in nature - load (storage and/or processing) accrues slowly over time while most functionalities could take advantage of batch processing. While certain kinds of document production can leverage batch processing, we study the viability of real-time processing for file-sizes that are atypical to the aforementioned workload (100s of megabytes).

Multi-cluster systems (e.g. [12] et al.) and aggregated grids (e.g. [13]), do not support dynamic scalability prevailing in the current pay-as-you-go model of cloud computing. Further, the need to transmit large files over clouds separated by a low bandwidth Internet pipe, results in the transit time being of the order of processing time, thereby making the cloud bursting approach more challenging. Metabrokering strategies (e.g. [7]) are also used to aggregate resources while respecting constraints of various types — the strategies are complementary to our approach in the sense that domains which have slackness and out-of-order requirements could benefit from the proposed schedulers while our domain could use meta-brokering strategies while bursting to multiple clouds. Policy-based techniques could also be used for balancing workloads across clouds (see [1] [6]). The difference in our work is that there are specific queue dynamics that have to be respected (slackness and out-of-order constraints) in addition to quality of service optimizations (speed-up, utilization etc.). Another difference is that in our system processing and transit times are of the same order of magnitude - causing severe bandwidth related bottlenecks. Novel to our work is the use of learned models (quadratic response surface model for processing time and a time-of-day model for bandwidth) that drive autonomic behavior continuously. Furthermore, our workload comprises of documents and images whose characteristics if gleaned could prove very effective in association with the models. While predicting with a high degree of accuracy may still involve more work, prior work on long-tailed workload (which is normal in our domain) has shown that utilizing a guessed remaining workload is quite effective in scheduling (see [8]) decisions.

VII. Conclusions and Future Work

Recent work in Cloud Bursting has not considered workloads whose transit time to an external cloud is of the order of processing in either cloud. This issue caused in part by large file sizes and the disparities in the adoption of high-speed bandwidth pipelines raises questions about the viability of cloud-bursting, particularly for small/medium-sized organizations. Our work demonstrates that small organizations could use cloud-bursting effectively but schedulers have to be more sophisticated to understand the domain's dynamics. We have concentrated on inter-cloud distributed analytics on workloads that are predominantly computations on documents and images — an area that provides apriori visibility into the features and characteristics of the jobs in a queue. We have proposed three flavors of schedulers that are not only effective in cloud-bursting of large jobs but also honor various queue constraints (such as slackness) and service level agreements. To this end, we have shown the viability of optimizing and guaranteeing a multitude of quality of service metrics on a hybrid cloud connected by a thin pipe. The extension of the scheduler techniques discussed in this paper to multiple job classes would make the cloud bursting approach applicable to a multitude of environments like academic computing domain. The usage of more advanced models to mitigate the transient nature of the Internet remains the subject of future investigation. Among other optimizations that can be studied in future, could be modulating the chunking of jobs as a function of their position in the input queue. This non-uniform chunking technique could impact utilization of the inter-cloud resources. We also plan to conduct experiments with larger number of resources in future.

References

[1] H. Kim, S. Chaudhari, M. Parashar, and C. Marty, "Online risk analytics on the cloud," in *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 484–489.

[2] Amazon, *Elastic Map-Reduce*, 2009. [Online]. Available: http://aws.amazon.com/elasticmapreduce/

[3] P. Mell and T. Grance, "The nist definition of cloud computing," National Institute of Standards and Technology, Tech. Rep., October 2009.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, November 2002.

[5] C. Babcock, "Hybrid clouds," pp. 15–18, September 2009. [Online]. Available: www.informationWeekanalytics.com

[6] H. Kim, M. Parashar, D. J. Foran, and L. Yang, "Investigating the use of autonomic cloudbursts for high-throughput medical image registration," in *GRID '09: Proceedings of the 2009 10th IEEE/ACM International Conference on Grid Computing*. Banff, Alberta, Canada: IEEE Computer Society, 2009.

[7] I. Rodero, F. Guim, J. Corbalan, L. Fong, and S. M. Sadjadi, "Grid broker selection strategies using aggregated resource information," *Future Generation Computer Systems*, vol. 26, no. 1, pp. 72 – 86, 2010.

[8] M. Harchol-balter, "Task assignment with unknown duration," *Journal of the ACM*, vol. 49, pp. 260–288, 2000.

[9] R. H. Myers and D. C. Montgomery, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, 2002.

[10] Amazon, *Hadoop: Open source implementation of MapReduce*, 2009. [Online]. Available: http://hadoop.apache.org/

[11] IBM, *CloudBurst*, 2009. [Online]. Available: http://www.ibm.com/ibm/cloud/cloudburst/

[12] H. Bal, "The distributed asci supercomputer project," *The International Journal of Supercomputer Applications and High Performance Computing 11(3), 212223*, vol. 34, pp. 76–96, 2000.

[13] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, vol. 11, pp. 115–128, 1996.