MRM: Delivering Predictability and Service Differentiation in Shared Compute Clusters

Masoud Moshref[†], Abhishek B. Sharma[*], Harsha V. Madhyastha[‡]
Leana Golubchik[†], Ramesh Govindan[†]

ABSTRACT

Computing-as-a-service is here to stay, and its benefits, such as resource elasticity and efficient utilization, have been well-documented. However, little attention has been paid to *service models* for computing-as-a-service. Existing service models are relatively simplistic in that they provide little or no predictability in job finish times and allow for limited service differentiation. In this paper, we propose a new service model for shared clusters, MRM, that addresses these shortcomings. MRM estimates job processing times conservatively to provide predictability in finish times, and uses pricing to incentivize users to contribute slack so that delay-sensitive jobs can be accommodated. We have instantiated MRM in the context of shared MapReduce clusters. Our results demonstrate that MRM can provide *predictability of job finish times* and *differentiated* service under a variety of user demands (workloads).

1. INTRODUCTION

Computing-as-a-service has been evolving steadily. Today, several enterprises host shared computing clusters for use by their employees (e.g., Google's cloud [27]), and several providers now offer computing services publicly (Amazon's web services (AWS), Microsoft's Azure). Moreover, service providers today provide computing abstractions at various levels: bare virtual machines, specialized languages and runtimes (e.g., for massively-parallel data processing—MapReduce [10], Dryad [15]), web services, and so on. For example, Amazon offers both bare virtual machine clusters as well as MapReduce clusters [2].

However, despite the emergence of several computing services and the wide range of abstractions they offer, little attention has been paid to the *service model*: the interface between the user and the operator that determines the type of service provided. Currently, relatively simplistic models seem to be the norm, where the operator undertakes to provide resources to complete a job, but does not provide any assurance of when the job will be completed (*predictability*) or provides limited ways in which users can ask for different levels of service (*service differentiation*). For instance, AWS and Azure use a "rental" based service model, in which users can choose from a range of virtual machine instances (of different sizes) and pay a fixed rate for each instance; the system makes no statement about when jobs finish. On the other hand, most grid-computing infrastructures charge users based on resource usage (e.g., node hours), and provide differentiation using a few discrete priority queues that are differentiated by job size and duration; low priority jobs have no guarantees on when they finish.

Our contributions. In this paper, we explore the design of a service model that attempts to provide both *predictability in finish times* and the capability for *differentiation* by allowing users to *select* desired finish times (e.g., choosing an earlier finish time for a delay-sensitive job). Our approach, called MRM (for Map-reduce Market), achieves these goals by enabling a service model in which a user is presented with a price-deadline curve at the time when she submits a job. The user can choose an appropriate point in this price-deadline curve based on the delay-tolerance of her job and her current wealth. This, in combination with deadline-based scheduling, ensures both predictable finish times and provides users with a choice of finish times.

To enable such a service model, we develop two key components. First, to ensure predictability, we estimate *a priori* the processing time of a job, at the time it is submitted for execution. To do so, we characterize the accuracy with which various analytical methods can estimate job processing times, and design a method for computing feasible finish times for a job. MRM computes the feasible finish times for a job based on the current system load and its estimate of the computing resources required by the job, which is determined based on resources consumed by prior executions of similar jobs. MRM then restricts the choice of deadlines for a job to be within the range of feasible finish times.

Second, to achieve service differentiation, MRM *prices* job completion deadlines. By charging more for earlier deadlines, MRM encourages users of delay-tolerant jobs to select deadlines later than the earliest possible finish time. Thereby, this deadline pricing mechanism incentivizes users to offer slack in the execution schedule. This slack can be used to accommodate earlier finish times (than possible with FCFS) for later arrivals of delay-sensitive jobs.

Any service model must be designed in the context of a computing abstraction, as details of prices and load depend on the abstraction. We have instantiated MRM for shared MapReduce [10] clusters. Given the widespread use of such clusters, and the lack of predictability and service differentiation in these clusters, we believe that MRM addresses an important and urgent need. We believe MRM can be extended to other massively-parallel programming frameworks inspired by MapReduce, such as Dryad [15], as

[*]NEC Labs, America
[†]Computer Science, USC
[‡]Computer Science, UC Riverside

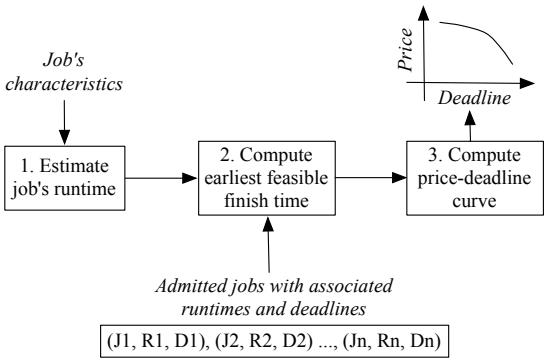


Figure 1: Overview of MRM.

well as higher-level frameworks such as DryadLINQ [33], Hive [3], and Pig [1].

Finally, experiments using our MRM prototype on a 40 server cluster reveal that MRM can achieve near-perfect predictability in realistic scenarios. Furthermore, our trace-driven simulations show that, by incentivizing users executing delay-tolerant jobs to choose looser deadlines, MRM reduces the waiting time of delay-sensitive jobs, allowing 35% of them to jump ahead of delay-tolerant jobs. Despite inaccuracies in the estimation of job processing times, MRM achieves this service differentiation while keeping the deadline violation rate comparable to that of the FCFS scheduler. In contrast, the priority scheduler—another strategy that enables expedited processing of delay-sensitive jobs—results in over a 30% deadline violation rate for delay-tolerant jobs.

The rest of this paper is organized as follows. We provide a brief overview of MRM in Section 2. We discuss how MRM determines feasible finish times by estimating job processing times in Section 3, and we present MRM's pricing strategy in Section 4. Then, in Section 5, we discuss how MRM offers predictability and service differentiation in combination. In Section 6, we present the results from our experiments to evaluate MRM's performance. We discuss related work in Section 7, and conclude with a discussion of future work in Section 8.

2. MRM GOALS AND OVERVIEW

In this section, we first describe MRM's design goals and contrast it with current solutions for job scheduling. We then provide an overview of MRM, and finally discuss the challenges involved in its design and implementation.

Design goals. We design MRM for clusters that provide computing-as-a-service; users submit *jobs* that are then scheduled on the cluster. We define a job more precisely later; for now, a job is a distributed computation on the cluster. Two goals guide our design of MRM: (1) to provide users with *predictability in finish times* of jobs, and (2) to enable the system to *differentiate* between delay-sensitive jobs and delay-tolerant ones.

Current mechanisms for scheduling jobs in shared clusters (e.g., FCFS, priority queues, and fair scheduling) can meet one, but not both design goals. Under the FCFS policy, assuming that the runtimes of jobs are known or can be estimated, we can easily estimate the finish time of a job from its position in the queue. However, FCFS does not provide differentiated service: delay-sensitive jobs can get stuck behind large jobs and experience high waiting times, even if those large jobs are delay-tolerant. Priority queueing can be used to implement differentiated service levels for jobs but may result in unbounded finish times for lower-priority jobs. Finally, fair scheduling supports job differentiation through weights but cannot ensure tight deadlines; a job's fair share may vary with the load, resulting in unpredictable finish times.

MRM overview. In contrast, MRM achieves predictability and differentiation by following the procedure illustrated in Figure 1. When a job is submitted by a user, MRM takes the characteristics of the job (available, for example, based on prior executions of similar jobs) into account in order to first compute an estimate of the job's runtime, say R seconds. MRM may not be able to complete the new job R seconds from the current time because there may be other jobs that are already queued up for execution on the cluster. Therefore, MRM computes the *earliest feasible finish time* for the new job by using its estimated runtime of the new job in combination with the runtimes and negotiated deadlines of jobs that it had previously admitted.

However, if MRM agrees to set this earliest feasible finish time as the deadline of the new job, then jobs get executed as per a FCFS schedule. Thus, to ensure service differentiation, MRM *negotiates* a deadline with the user who submitted the job. MRM associates a *price* with each of its feasible finish times for the job, thus defining a *price-deadline* curve for it. Finish times that are farther into the future have lower associated prices, thus incentivizing a user who submits a delay-tolerant job to choose a deadline that is later than the earliest feasible finish time.

The key innovation in MRM is the design of the methods by which it computes and prices feasible finish times. On the one hand, accurate estimation of feasible finish times enables predictability. On the other, judicious pricing of deadlines incentivizes users who submit delay-tolerant jobs to choose loose deadlines, thus enabling delay-sensitive jobs to be executed ahead of them without violating their deadline. We describe MRM's estimation of feasible finish times in Section 3, and its pricing mechanism in Section 4, and how we combine the two in Section 5.

3. PREDICTABILITY

In this section, we first use simple queuing models to obtain insights for achieving predictable finish times. Then, we describe how to estimate job processing times. In the next section, we describe how to achieve service differentiation, and then use these insights and estimation methods in Section 5 to design a method for predicting job finish times in the presence of service differentiation.

3.1 Modeling Predictable Finish Times

The simplest queueing model for clusters executing MapReduce jobs is two queues connected in tandem with the first queue representing the Map phase of a job and the second the Reduce phase [28]. However, since a large fraction of MapReduce jobs submitted to real-world production clusters do not have a Reduce phase [34], we can model them with a single queue. We first present results for the single queue case because they are easier to explain and capture key take-aways; we then present the results for tandem queue for concreteness.

Single queue: Map-only jobs

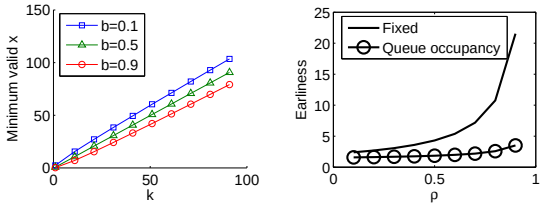
Notation and Problem Statement. Consider a scenario with Poisson job arrival rate λ, and i.i.d. exponentially distributed map phase duration with mean μ_m. We can then model the cluster as an M/M/1 queue with FCFS scheduling; we use M/M/1 instead of M/M/k for simplicity of analysis, but our insights from a M/M/1 queue apply in case of M/M/k as well. Let a_i, f_i and d_i be the arrival, finish and deadline time, respectively, of job i. We define $T_i = f_i - a_i$ as the *sojourn time* for job i. Suppose that we assign its deadline to $d_i = a_i + y_i$, i.e., the scheduler agrees to finish the job within y_i time units after its arrival. This means that the scheduler estimates the sojourn time as y_i. For jobs that finish before their deadline, i.e., $f_i < d_i$, we define their *earliness* as $e_i = d_i - f_i = y_i - T_i$.

We can characterize the *predictability* of this simple queueing model using its *deadline violation rate*, defined as the fraction of jobs which violate their deadline. It is trivially possible to achieve no violations by assigning each job an infinite deadline; we consider such a system to be not very useful. Rather, our goal is to *minimize the earliness of jobs while bounding the deadline violation rate to* β.

Predictability and Queue Occupancy. One way to estimate the deadline of a job is to estimate its sojourn time. One can conservatively use a fixed sojourn time estimate for all jobs, and set the sojourn time estimate such that the deadline violation rate bound is met. In this section, we show that this approach results in higher earliness than one which takes into account the estimates of the duration of jobs already in the queue.

With a fixed sojourn time estimate of $y_i = x$ for all jobs, and assuming map-only jobs with $\lambda < 1$ and $\mu_m = 1$, the lower bound for x assuming a deadline violation rate of β is given by:

$$P(T > x) = e^{-(1-\lambda)x} \leq \beta \Rightarrow x \geq \frac{-log(\beta)}{1-\lambda} \qquad (1)$$



(a) Minimum valid delay (b) Earliness for $\beta = \%5$

Figure 2: **M/M/1 deadline prediction**

For a given β and its corresponding x, the average earliness is:

$$E_{s1} = x - E[T|T \leq x] = x - \int_0^x t \frac{f_T(t)dt}{F_T(x)} \qquad (2)$$

$$= x - \frac{1}{1-\lambda} + \frac{xe^{-(1-\lambda)x}}{1-e^{-(1-\lambda)x}} \qquad (3)$$

Since $\lambda < 1$ and E_{s1} is an increasing function of x, we pick the lower bound for x in Eq. (1) to minimize the average earliness subject to a deadline violation rate of at most β:

$$E_{s1} \geq \frac{\frac{log(\beta)}{\beta-1} - 1}{1-\lambda} \qquad (4)$$

Clearly, as job arrival rate λ increases, both x and average earliness increase, so the system will be forced to choose large x in order to satisfy the deadline violation rate bound.

Now consider an approach which sets the sojourn time estimate of a new job, y_i, using the duration estimates of jobs already in the queue (recall that we assume an FCFS scheduler). Let x_k denote the conditional sojourn time of a job, given that k jobs are ahead of it in the queue. This conditional sojourn time for map-only jobs (a single M/M/1 queue) has an Erlang$(1, k+1)$ distribution. It is then possible to find x_k to achieve a given deadline violation rate β:

$$\sum_{k=0}^{\infty} P(T > x_k | k \text{ jobs in system}) P(k \text{ jobs in system})$$

$$= \sum_{k=0}^{\infty} \left(\sum_{n=0}^{k} \frac{1}{n!} e^{-x_k} x_k^n \right) \lambda^k (1-\lambda) \leq \beta \qquad (5)$$

One way to select x_k to satisfy Eq. (5) is to keep the violation probability smaller than β for every k. [1] Then Eq(5) holds irrespective of the arrival rate because $\sum_{k=0}^{\infty} \beta \rho^k (1-\rho) \leq \beta$. We solved Eq. (5) for x_k numerically, and we plot its values for different k and β in Figure 2(a). The linear curves suggest that we can approximate x_k using a linear function of k and β. We estimate $x_k(\beta) = f(\beta) \times (k+1)$, with $f(\beta) = b_0\beta + b_1$ obtained using non-linear least square regression. We achieve R^2 accuracy of 0.998 for $f(\beta) = -0.36\beta + 1.16$. The negative slope of $f(\beta)$ (i.e., $b_0 = -0.36$) is because, if we can tolerate a higher deadline violation rate β, we can set x_k to a smaller value ($x_k = 0$ for $\beta = 1$).

[1] This approach is also usable for any arrival process as it ignores $P(k \text{ jobs in system})$.

We can also compute the average earliness when deadlines are set based on conditional sojourn time as per Eq. (6) and (7). We compute the minimum x_k satisfying Eq(5) and compare the average earliness of the fixed sojourn time estimates to those offered by the approach that takes queue occupancy into account (Eq. (6) and (7)) in Figure 2(b).

$$E_{d1} = \sum_{k=0}^{\infty} E\left[x_k - T | N = k+1, T \le x_k\right] \rho^k (1-\rho) \qquad (6)$$

$$E[x_k - T | N = k+1, T \le x_k] = x_k - \int_0^{x_k} t \frac{f_T(t|N=k+1)}{F_T(x_k|N=k+1)} dt$$
$$= x_k - (k+1) + \frac{e^{-x_k} x_k^{k+1}}{k!(1 - \sum_{n=0}^{k} \frac{e^{-x_k} x_k^n}{n!})} \qquad (7)$$

Going one step further, we can set the sojourn time estimate, y_i, based on the *specific* set of jobs in the queue, Q_i, when job i arrives. This approach works with any arrival process and job duration distribution, assuming we can calculate the *conditional sojourn time distribution* – the distribution of the sojourn time of a job given the current jobs in the queue:

$$T_{Q_i} = T | \text{jobs } Q_i \text{ in the queue at arrival of } i \qquad (8)$$

In this case, we can always find y_i such that $P(T_{Q_i} > y_i) \le \beta$.

Computing the Sojourn Time. How can one estimate distribution of the conditional sojourn time for a new job arrival? Assume that, for each job we can estimate its *processing time* with a Gaussian error. (We can do this, for example, by using historical processing times for jobs with similar characteristics, as discussed in the next section.) This means that the processing time of each job will be a Gaussian random variable with mean equal to the predicted value. Then, we can compute the *joint distribution* (JD) of the processing time distributions of all the jobs in the queue, together with the processing time distribution of the new arrival. This joint distribution is the conditional sojourn time distribution. Assuming that the errors of the job processing time estimation are independent, the conditional sojourn time is a Gaussian random variable with mean (variance) equal to sum of the mean (variance) of the estimated duration of the jobs in the queue and the new job. For example, to ensure a maximum violation rate of 5%, we can estimate the sojourn time as the mean plus 1.64 standard deviations of the joint distribution.

Tandem queues: Jobs with Map and Reduce tasks

We can derive analogous bounds on earliness for jobs with map and reduce phases using a model with two M/M/1 queues in tandem. With tandem queues, the sojourn time $T = T_m + T_r$ where T_m is the sojourn time in the first (map) queue and T_r is the sojourn time in the second (reduce) queue. From Jackson's theorem [9], we know that the number of jobs in the two queues in tandem are distributed as if each queue is M/M/1. Hence, $T_m \sim Exp(\mu_m - \lambda)$ and $T_r \sim Exp(\mu_r - \alpha\lambda)$, $0 \le \alpha \le 1$.

Fixed Sojourn Time estimates . For $y_i = x$, i.e. fixed sojourn time estimates, we only need to know the distribution for T which is Hypo-exponential$(\overline{T_m}, 1, \overline{T_r}, 1)$. A Hypo-exponential random variable is a sum of multiple exponential random variables each with their own rate. Here, it has one random variable with rate $\overline{T_m}$ and another with rate $\overline{T_r}$. Equation 9 shows the violation bound for the case that $\overline{T_m} \neq \overline{T_r}$. [2]

$$P(T > x) = \frac{\overline{T_m} e^{-\overline{T_r} x} - \overline{T_r} e^{-\overline{T_m} x}}{\overline{T_m} - \overline{T_r}} < \beta \qquad (9)$$

Putting the Hypo-exponential PDF in eq. (2) the earliness formulation will be as eq. (10). We compute minimum x numerically using eq. (9) and put it in eq. (10) to compute minimum earliness while preserving the deadline violation bound. Figure 3 shows the earliness with fixed sojourn time estimates when the deadline violation rate $\beta = 5\%$ and 23% of jobs have a reduce phase.

$$E_{s2} = x - \frac{\overline{T_m T_r}\left(\frac{x e^{-\overline{T_m} x}}{\overline{T_m}} - \frac{x e^{-\overline{T_r} x}}{\overline{T_r}} + \frac{x e^{-\overline{T_m}} - 1}{\overline{T_m}^2} - \frac{x e^{-\overline{T_r}} - 1}{\overline{T_r}^2}\right)}{\overline{T_m} - \overline{T_r} - \overline{T_m} e^{-\overline{T_r} x} + \overline{T_r} e^{-\overline{T_m} x}} \qquad (10)$$

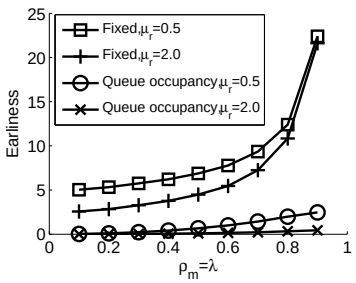


Figure 3: Tandem queues

Sojourn time based on queue occupancy . The sojourn time of a job with reduce phase observing k_m and k_r jobs ahead of it in the map and reduce queues, respectively, follows *Hypo-exponential*$(\mu_m, k_m+1, \mu_r, k_r+1)$. Using the same approach as for map-only jobs, we use conditional sojourn time distribution based on the number of jobs in the map and reduce phases, k_m and k_r. We numerically compute the minimum $y_i = x_{k_m,k_r}$ and put it in the earliness formula computed based on the phase-type definition of Hypo-exponential distribution [23]. Figure 3 compares the earliness for the fixed sojourn time estimates against the approach that queue occupancy into account.

From Figures 3 and 2(b), we can see that the latter approach offers a qualitatively different behavior of earliness times as a function of arrival rate, so in this paper we only consider approaches that assign deadlines based on queue occupancy. However, to use this approach, we need to estimate job processing time and find the standard deviation of error. Next, we discuss how to do this in practice.

[2] Otherwise, we can just use Erlang distribution

3.2 Estimating Job Processing Times

A typical MapReduce job consists of three pipelined phases: Map, Shuffle, and Reduce. The Map and the Reduce phases can consist of multiple independent tasks that execute in parallel. The output of the Map phase is partitioned into (disjoint) bins which are stored locally. The Shuffle phase handles this partitioning and transport of Map outputs to relevant Reduce tasks. To estimate the processing time of such jobs, we first need to estimate the duration of the Map, Shuffle, and Reduce phases. Here, we focus on performing this estimation at the granularity of a job with a single Map and Reduce phase; we defer the estimation of processing times for more complex data-flow graphs to future work.

Overview. For jobs that are repeatedly executed, data from multiple runs can be used to estimate an accurate bound on their processing time [11]. Another scenario that arises in practice is users submitting *similar* jobs. We can define similarity along two dimensions: (1) number of map and reduce tasks[3], and (2) *type* of a job. A job's type is determined by the function performed by its map and reduce tasks. For instance, the map and reduce tasks of a job for sorting data process data differently compared to their counterparts in a job that computes the frequency of occurrence for each word in a set of documents. Hence, all data sorting jobs are of the same type, say *sort*, which is different from the type of *word-count* jobs used for computing word frequencies.

In this section, we demonstrate that we can *estimate the processing time of a job on its arrival based on statistics of prior executions of similar jobs on the cluster*. In our experiments, we have observed that the processing time of a MapReduce job depends upon three features: (1) the number of Map tasks, (2) the number of Reduce tasks, (3) the size of the input and output of the Map and Reduce tasks. We model the size of a Map task's output, and a Reduce task's input and output using two features—*map-reduction* factor, and *reduce-reduction* factor. The *map-reduction* and *reduce-reduction* factors measure the average reduction in input size after the Map and Reduce phase, respectively. They act as proxies for a job's type, e.g., in case of a *grep* job searching for a "rare" pattern in a large volume of text, the Map phase will filter out most of the original input, and thus, have a large data reduction factor, whereas there is no reduction in the data size after Map and Reduce phases for a *sort* job. From a resource requirement perspective, the two reduction factors determine the contention for network and disk I/O, and the number of Map and Reduce tasks determine the computation resources needed across the cluster.

In MRM, these features collectively form a feature space [4], and each job represents a point in this feature space. When users submit a job, they must specify the feature vector associated with this job; users can obtain this information from trial runs of the job.

Given a job's feature vector, MRM predicts the job's processing time using a statistical model. This model is constructed using training data obtained from a live cluster. Specifically, after a job's completion, MRM obtains: a) task completion times for each of the three MapReduce phases of the job, and b) values for the map and reduce reduction factors. MRM adds these processing time samples to its database of statistics for previously executed jobs with a corresponding set of features, and uses these samples to train a statistical model for processing time prediction.

Processing time estimation methods. To achieve predictability, MRM must estimate a job's processing time as well as the standard deviation of estimation error (refer to Section 3.1). There are several classes of methods that we can choose from: *linear regression* methods (e.g., ordinary least squares (OLS) and ridge regression), *canonical correlation analysis (CCA)* based methods, and *Gaussian Process* (GP) based methods. However, only two of these methods (OLS and GP [8, 25]) enable us to compute the standard deviation of the prediction error in a principled way, and hence, are more relevant for MRM. We present a detailed comparison of all methods in Section 6.1.

We choose to use GP-based processing time estimation in MRM. This is because, unlike OLS, which captures the linear relation of output on training features and minimizes the mean square error, Gaussian process regression does not assume any parametric representation for the function mapping job features to processing times. Instead, it tries to infer a distribution over functions mapping features to processing times from measurement data [8]. Informally, given the value of an unknown function f at an finite, but arbitrary, set of points, say $x_1, \ldots, x_n$, GP assumes that the probability distribution $p(f(x_1), \ldots, f(x_n))$ is jointly Gaussian with mean $\mu(x)$ and covariance $\Sigma(x)$. We can compute the mean and the covariance matrix from measurement data, and then use it to estimate the value of f at a new point x^*. In MRM's case, x_i represents the features for job i and $f(x_i)$ is the processing time for a phase. As we show later in Section 6.1, GP-based estimation enables us to accurately predict the processing times of MapReduce jobs.

4. SERVICE DIFFERENTIATION

Our discussion in the previous section showed how MRM can achieve predictability under FCFS scheduling by estimating the deadline for a job based on its conditional sojourn time. However, FCFS scheduling does not provide any service differentiation. To provide service differentiation, MRM requires at least some jobs to offer *slack* by picking a deadline that is later than their earliest finish time. MRM incentivizes jobs to offer slack *by associating every feasi-*

[3] In Hadoop, the number of map tasks is proportional to the input size, e.g., if a Hadoop cluster is configured with default settings (including HDFS block size equal to 64MB), then a job processing a 640MB input file will have 10 map tasks.

[4] Other features may also be important in estimating processing times, e.g., jobs that read compressed input might differ in overall processing time from jobs that read uncompressed input. We leave an exploration of the complete feature space for future work.

ble deadline for a job with a corresponding price. Hence, a *delay tolerant* job is likely to accept a deadline that is later than its earliest feasible finish time. MRM can then provide service differentiation, without compromising predictability, by allowing *delay-sensitive* jobs—which are willing to pay more to finish earlier—to skip ahead of other jobs waiting in the queue. However, when enqueued jobs have offered slack, computing the earliest feasible finish time for a new arrival is more complex than in case of FCFS with no slack.

In this section, we study the factors that impact the design of a pricing function (Section 4.1), and then propose a pricing function for MRM (Section 4.2). Later, in Section 5, we discuss how to compute the earliest finish time for jobs in the presence of slack using a *prediction interval* for processing time and *backfilling*.

4.1 Theoretical intuition for pricing

Consider the single queue model for map-only jobs with the following simplifications: time is slotted, and any delay sensitive/delay tolerant job can be processed in one time slot. We can represent the *state* of this queue using a bit string with each bit representing a slot: the least-significant bit represents the current slot, and bit i is 1 if there is a job waiting in the queue with deadline i, and 0 otherwise. For example, a bit vector 10100 at $t = 0$ denotes two jobs in the queue with deadlines $t = 4$ and $t = 2$. At the beginning of each slot, multiple delay tolerant jobs can arrive, each with a probability of p; the precise arrival process for delay tolerant jobs is not relevant for our analysis. To be conservative, we assume that, in a given slot, each delay-sensitive job arrives after delay-tolerant jobs. A delay-sensitive job always wants to be processed in its arrival slot, and is willing to pay \$1 for it. A delay sensitive job that arrives during a busy slot [5] cannot be serviced; we refer to such a job as being rejected by MRM. Users who submit delay-tolerant jobs are offered a price vs. deadline curve, and pick a deadline based on how much they are willing to pay for it. However, a user who submits a delay-tolerant job always wants to pay at most c, an amount strictly less than \$1, even when MRM can process the job in the slot in which it arrives.

In this model, the operator risks losing revenue whenever the current slot is busy (i.e., least significant bit is 1), because if a delay sensitive job arrives, it will be rejected. Note that the current slot is busy only because we need to execute a delay tolerant job (that arrived in the past) to meet its deadline. Rejecting a delay sensitive job causes a loss of \$1, and since delay tolerant jobs only pay an amount strictly less than \$1, the operator loses some revenue. However, since a delay sensitive job arrives with probability p, the *expected* revenue loss when the server is busy processing a delay tolerant job during the current slot is p. Hence, to avoid losing revenue, the operator should charge the delay tolerant job \$$p$.

[5] A slot is busy, i.e., its bit value is 1, if MRM must run a job in it to meet its deadline.

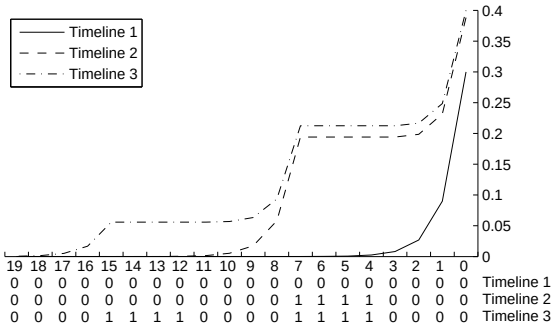


Figure 4: M/M/1 deadline prediction

While it is straightforward to compute that p is the revenue loss from the current slot being busy, the general problem of computing the revenue loss from assigning slot i as the deadline for a delay tolerant job is complex. For example, consider the bit vector 10000. It denotes one job in the system at current time $t = 0$ with deadline at $t = 4$. There will be a rejection in this case only if deadline sensitive jobs arrive at all 5 slots. This is because, if there is no arrival for any slot 0 to 3, then a work-conserving server will finish the delay tolerant job earlier than its slot 4 deadline. The probability of rejecting a deadline sensitive job conditioned on slot 4 being assigned as deadline to a delay tolerant job is therefore p^5. Hence, a delay tolerant job must pay p^5, where $p \leq 1$, as the price for having slot 4 as its deadline.

Figure 4 shows the price vs. deadline curve for three different bit strings for $p = 0.3$. Timeline 1 has the lowest price for each slot because the queue is empty while Timeline 2 has lower price than Timeline 3 as it has more empty slots and has therefore lower load. Hence, the price function is lower for larger *slack* but is non-linear. Notice the flat prices for slots 4–7 for Timelines 2 and 3, and 12–15 for Timeline 12. The price for these slots is the same as the price for the earliest empty slot before them—slot 3 and slot 11, respectively. This is because a delay tolerant job is allowed to select a slot in 4–7 or 12–15 as its deadline, but since these slots have previously been selected as deadlines, MRM must use slot 3 or 11 to meet this deadline.

Algorithm for computing the price curve. A brute force approach to find the probability of rejection is to find the number of rejections before and after scheduling a slot for any pattern of deadline-driven job arrival. The computation complexity of the algorithm is $O(n2^n)$ for each slot in a timeline with n slots. To describe our algorithm with computation complexity of $O(n)$, we define two concepts *Capacity (C)* and *Overflow (O)*. Suppose that we divide a bit string B into left, B_L, and right, B_R, bit strings from an arbitrary point[6]. The capacity of bit string B_R, C_{B_R}, is the number of slots that the server can run a job from B_L in slots of B_R because it could not remain idle. The overflow of bit string B_L, O_{B_L} is the number of background jobs in

[6] We omitted the discussion about the boundary cases for brevity.

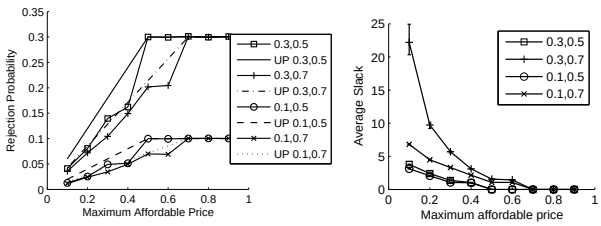
B_L that should run in B_R to have no rejection. For example, the capacity of 00000 is $5-k$ if k deadline-driven jobs come, but its overflow is always 0. In contrast, the capacity of 11111 is always 0, and its overflow is k if k deadline-driven jobs come. We can define capacity and overflow of a bit string recursively: $C_B = C_{B_L} + max[C_{B_R} - O_{B_L}, 0]$ and $O_B = O_{B_R} + max[O_{B_L} - C_{B_R}, 0]$. This means that the capacity of B is the capacity of B_L plus whatever capacity remains on B_R after it accommodates overflow of B_L. The corresponding argument for overflow is similar. Note that as $C_B \leq |B|$ and $O_B \leq |B|$ where $|B|$ is size of bit string, the complexity of finding C_B and O_B in a divide-and-conquer algorithm is $O(n^3)$. However, notice that if either $|B_L| = 1$ the complexity of calculating the capacity will be $O(n)$. So if we sweep from right to left adding one bit at a time to calculate capacity of B the complexity will be $O(n^2)$. We do the same from the left to right to calculate the overflow.

Now that we know a way to calculate the capacity and overflow of a bit string, we discuss the algorithm to find the probability of additional rejection because of allocating slot k to a background job. We define left bit string L as bits left to slot k and R for the right slots. Two cases may happen: First, if no deadline driven job come at slot k, it was empty if we did not allocate it. So, it would be required only if the overflow of left bit string is larger than the capacity of the right one, $O_L > C_R$. Second, if a deadline driven job come at slot k, it will be rejected unless the scheduled background job ran before in R. So the background job will use one capacity of R, and we have an additional rejection if $O_L \geq C_R$. Eq. (11) summarizes the probability formula for both cases and can be calculated in $O(n)$.

$$(1-p)\sum_{m=0}^{0_R} P[O_L > m]P[C_R = i] + p\sum_{m=0}^{0_R} P[O_L \geq m]P[C_R = i] \tag{11}$$

We cannot reduce the order of algorithm for an *arbitrary* slot from $O(n^2)$. However, we are calculating the probability for *consecutive* slots for a complete price function, which allows us to reduce the amortized complexity over n slots to $O(n)$ using $O(n^2)$ memory space. First note that if either $|B_R| = 1$ or $|B_L| = 1$, the complexity of calculating the capacity and overflow of B will be $O(n)$. So knowing the capacity of right hand side of slot $k-1$, we calculate the capacity of right hand side of slot k by just adding slot $k-1$ to it, which takes $O(n)$ steps. For the overflow of left bit string, we can reuse the computation of overflow of left of slot 0: For slot 0, we calculate overflow of the left bit string iteratively starting from slot $|B|-1$ to slot 0 adding one bit at a time and keep intermediate results in memory. Then for any slot k, the overflow of left side is available in memory. So the amortized complexity over n slots will be $O(n)$.

Takeaways for designing deadline vs. price functions. We define the *slack* offered by a job as the difference between its chosen deadline and its earliest feasible finish time. For example, under our pricing model, delay sensitive jobs do not offer slack, whereas the slack offered by a delay tolerant job arriving at slot 0 to an empty queue and selecting slot j as its deadline is j. Our simple model for pricing provides the following two takeaways for designing a price vs. deadline function for MRM: 1) The price should be a non-linear decreasing function of slack; and (2) the pricing function should be load dependent, i.e., it should take the processing times and deadlines of jobs already in the system (see the different curves for Timelines 1, 2, and 3 in Figure 4). Additionally, the price should take into account a job's processing time and the purchasing power of delay tolerant jobs. The need to account for processing time arises when all jobs are not of the same duration (unlike the case in our model), and we discuss the important role of the purchasing power of delay-tolerant jobs in setting prices next.



(a) Rejection probability and its upper bound (b) Average slack

Figure 5: Pricing simulation result (Legend numbers are λ and p)

Impact of delay-tolerant jobs' purchasing power. The maximum amount c that delay-tolerant jobs are willing to pay denotes their purchasing power comparing to the purchasing power of delay sensitive jobs that pay γ for slot 0. With $c \geq p \times \gamma$ the purchasing power of delay tolerant jobs is high enough for them that the system converges to the cases that allow them to select the slot of their arrival as deadline, i.e., offer no slack. Hence, $c \geq p \times \gamma$ will result in the rejection probability equal to λ where $\lambda < 1$ is the arrival rate for delay-tolerant jobs in each slot.

For the case that $c \leq p \times \gamma$, we find the upper bound for the rejection probability using the equality of expected revenue loss of rejection and the expected revenue from delay tolerant jobs: The delay sensitive jobs come with probability p and are rejected with probability P_r. We lose γ for each rejection, so the expected revenue loss per slot will be $\gamma \times p \times P_r$. The revenue for each delay tolerant job is at most c and they arrive with rate λ at each slot; therefore, the expected revenue from serving delay tolerant jobs is *at most* $c \times \lambda$. Note that $c \times \lambda$ is the upper bound for the expected revenue as a delay tolerant job may not find a slot with price exactly c and must select the one with lower price. Based on the derivation of the price function the expected penalty and the expected revenue are equal, so $\gamma \times p \times P_r \leq c \times \lambda$ which leads to eq (12). Using the equation, the operator can predict the rejection probability based on the wealth distribution among delay sensitive and delay tolerant jobs. Also a com-

pany may adjust the wealth distribution among the groups submitting different types of jobs to maintain the bound on the rejection probability of delay sensitive jobs.

$$P_r \leq \begin{cases} \frac{c \times \lambda}{\gamma \times p} & c < p \times \gamma \\ \lambda & c \geq p \times \gamma \end{cases} \quad (12)$$

Figures 5(a) and 5(b) show the probability of rejecting a delay sensitive job and the slack (in number of slots) offered by delay tolerant jobs for different values of c. We generate these plots using simulations for our simple model of pricing with $\gamma = 1$. Each data point is averaged over 10 runs.[7] We have three simulation parameters, c, λ, and p. In Figure 5(a), we also show the theoretical upper bound on the rejection probability under our pricing model using curves labelled "UP".

4.2 Pricing MapReduce jobs

Our pricing model as described above made several simplifications: it considers only one server in the system, it assumes all jobs span only one time slot, and a delay sensitive job can only run in the slot in which it arrives. However, in practice, a MapReduce cluster typically has multiple servers, jobs can span multiple time slots, and a small delay may be tolerable even for delay sensitive jobs (so that delay sensitive jobs do not leave, but instead only accept their earliest finish time as deadline, i.e., they do not offer any slack). Extending the model to account for these realities is very complex. Hence, we propose a candidate price function based on the takeaways from our pricing model, and evaluate MRM's performance with them in Section 6.

We introduce the following notation used to define our pricing function. Let Q represent the set of jobs currently enqueued for execution at any particular point in time. For every job $i \in Q$, let f_i^e denote its earliest feasible finish time. If the user who submits job i selects $d_i \geq f_i^e$ as its deadline, we define the slack offered by this job, δ_i, as the sum of free processing time between deadline and earliest feasible finish time and normalize it by the average estimated jobs processing time.

We then propose the following pricing function $c(\delta_j, p_j)$ for a new job j with processing time p_j offering slack δ_j.

$$c(\delta_j, p_j) = \kappa \left(\frac{p_j}{\delta_j + 1} + \sum_{i \in Q} \left(\frac{p_i}{\delta_i' + 1} - \frac{p_i}{\delta_i + 1} \right) \right) \quad (13)$$

Here, for every job $i \in Q$, δ_i and δ_i' are the slack available in the system until d_i before and after the new job j is admitted.

There are three things to note about this pricing function. First, the first term on the RHS—$p_j/(\delta_j + 1)$—incorporates our two takeaways from Section 4.1: that price should be directly proportional to a job's processing time, but should be a non-linear decreasing function of slack δ_j. Second, the second term puts a price on the *change* in the amount of slack available in the system as a result of accepting job j offering slack δ_j into the system. Consider a particular such job k with processing time p_k that was previously accepted with slack δ_k. As a result of the new job j jumping ahead of k, the available slack until d_k may decrease, e.g., if $d_k > d_j$. Hence, we charge job j an additional amount $\left(p_k/(\delta_k' + 1) - p_k/(\delta_k + 1)\right)$ for decreasing the slack relative to k's deadline by jumping ahead of it. Note that if j selects a deadline that is after the deadline all the jobs currently waiting in the queue then the second term on RHS of (13) is zero. Lastly, the κ parameter lets us control pricing relative to the purchasing power of users with access to the system. If κ is set to 1, then a job with one unit of processing time that offers zero slack is charged \$1.

We considered other forms of price vs. deadline curves – e.g. pricing decaying exponentially based on the ratio δ_j/p_j or a quadratic decay as a function of δ_j^2/p_j. These functions are biased in favor of small jobs: they force large jobs to offer more slack, compared to small jobs, if they want to pay a certain price. They are suited for a scenario where small jobs are less delay tolerant compared to large jobs.

Other choices for pricing function. We consider and evaluate three different strategies: a) fixed premium with exponential price decay (eq. (14)), b) fixed premium with quadratic price decay (eq. (15)), and c) load-aware pricing (eq. (16)). In Eq. (16), ρ is the system load and γ [8] is the average job processing time across all jobs. In all three cases, we lower bound the price of a job j by r_j.

$$c_j = \kappa r_j \exp\left(\frac{-\delta_j}{p_j}\right), \ \kappa \geq 1 \quad (14)$$

$$c_j = \kappa r_j - \frac{\delta_j^2}{p_j}, \ \kappa \geq 1 \quad (15)$$

$$c_j = r_j \exp\left(\frac{\rho p_j}{\gamma}\right) \quad (16)$$

Calculating system load, ρ. To compute system load ρ, we need to take job deadlines into account, in addition to their processing times. To illustrate this, consider the example in Figure 7(a), jobs A and B require 10 and 20 seconds of execution time over 16 map slots, and their deadlines are $t = 70$ and $t = 110$. We can interpret this as the cluster having a load of $1/7$ until $t = 70$, and load of $2/3$ for $t \in (70, 100]$. However, as shown in Figure 7(b), if job C's deadline is $t = 110$, then the system load increases to 1 for $t \in [0, 110]$. Hence, users' deadline choices determine periods of heavy/light system load.

Based on this discussion, we define ρ as follows. Suppose at time $t = 0$, there are n jobs with processing times $p_1, \ldots, p_n$ and deadlines $d_1 \leq \ldots \leq d_n$ in the system. Consider a new job A with processing time p_A, for which the

[7] We do not show error bounds as the ranges are very tight.

[8] We can compute γ using moving averages and/or from historical job statistics collected in system logs.

user wants to select deadline d_A. The total remaining slack in the system after A has been scheduled is given by $\beta = d_A - \Sigma_{k:d_k \leq d_A} p_k$. We define job A's contribution to the total slack as β/p_A, and set $\rho = max(0, 1-\beta/p_A)$ in Eq. (16). Note that if $d_A = f_A$ (earliest finish time for A), then $\rho = 1$. Though several other definitions of ρ are possible, as we show later in this section, we find that our definition performs well.

Through our early evaluation, we found that the load-aware (e.q. 16) worked better than the other two which validates our second take-away in Section 4.1. However, the load definition of this function only considers the jobs until the deadline and not after it although we saw they are also important (Section 4.1). It also defines the load based on the duration of the new job which is not the focus of this paper. As a result, we concluded with our price function in e.q. 13.

5. ACHIEVING PREDICTABILITY AND SERVICE DIFFERENTIATION

MRM uses the slack provided by delay-tolerant jobs to provide differentiated service to any delay-sensitive job, by allowing it to jump ahead of jobs waiting in the queue at the time of its arrival. However, MRM needs to ensure predictability while doing this, and three challenges arise in this context. First, MRM needs to compute the conditional sojourn time for the new arrival in the presence of jobs that offer slack. Second, MRM needs to compute a set of feasible deadlines for the new arrival, so that the user can select prices (and the associated deadlines) for the new arrival. Finally, MRM needs to schedule these jobs in order to ensure the selected deadlines are met. We address each of these challenges below.

Using prediction-intervals to compute conditional sojourn time. Whenever MRM allows a job j to jump ahead of jobs waiting in the queue, it must ensure that doing so does not violate the deadline of jobs that j jumps ahead of. Ensuring this requires re-computing the conditional sojourn time distribution for all the jobs that j jumps ahead of. To avoid the computation of multiple sojourn time distributions on a new arrival, MRM estimates a prediction-interval (PI) upper bound for a job's processing time. The PI upper bound for a job depends on MRM's estimate of its processing time and the standard deviation of estimation error. For example, the 95% PI upper bound for job j is equal to $\mu_j + 1.64 \times \sigma_j$, where μ_j is the estimate for j's processing time and σ_j is the standard deviation of the estimation error. MRM can then use *the sum of these upper bounds and the upper bound for the new job as its conditional sojourn time*.

Now, we can easily use the upper bounds instead of job durations in the scheduler to use the available slack to expedite a delay-sensitive job and still maintain the deadline violation bound. This approach is simple to implement but more conservative. When the processing time estimation error follows a Gaussian distribution, a PI-based approach will lead to fewer deadline violations than a JD-based approach at the expense of larger earliness. This is because a PI-based approach always selects later deadlines compared to the joint distribution:

$$\sum_j \mu_j + \alpha\sigma_j \geq \sum_j \mu_j + \alpha\sqrt{\sum_j \sigma_j^2} \qquad (17)$$

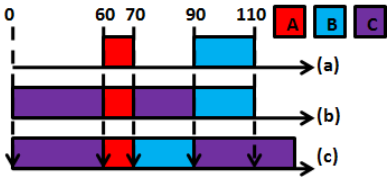


Figure 7: Example of MRM's estimation of earliest finish time for a new job.

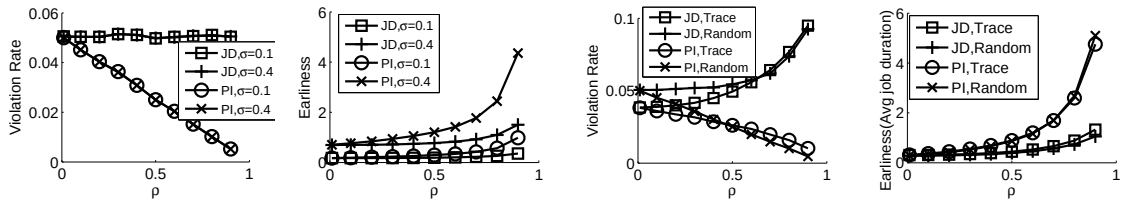
Figures 6(a) and 6(b) illustrate this tradeoff where the job processing times follow Gaussian distribution with mean 1 and mentioned standard deviation. Here, we simply use the mean value as the predicted duration.

In real-world, the distribution of the estimation error can deviate from Gaussian, and we examine its effect here. Figures 6(c) and 6(d) compare violation rate and the earliness (normalized by average job duration) on two settings "Random" and "Trace" for a server with one map and one reduce slot. For the Random case, we use the predicted durations to set deadlines but run based on generated Gaussian random durations with their mean and variance based on the estimated duration and its error standard deviation. For the Trace case, we run based on real durations. Note that the violation rate goes beyond the expected bound for Joint Distribution method even for the Random case. The reason is when the standard deviation is large for small jobs, the error distribution is not truly Gaussian but Rectified Gaussian [9] as the duration of jobs cannot be negative. Regardless of that, the metrics for both methods on the traces follow the same trend as the random generated numbers which indicates that the prediction error distribution has the same effect as Rectified Gaussian error.

Finding feasible deadlines. How does MRM determine the range of feasible deadlines for a new job, taking into account the deadlines committed to jobs already admitted into the system? First note that, if a deadline is feasible for a job, any deadline later than that is also feasible; hence, we only need to find the earliest feasible deadline. Given the deadlines of currently enqueued jobs, MRM greedily attempts to fill any available slack in the schedule to determine the new job's earliest feasible deadline. We illustrate this with a simple example.

Figure 7 shows an example job trace on a single server. Assume that the system is work-conserving and pre-emptive. When job C with PI upper bound of 80 seconds arrives at $t = 0$, there are two jobs A and B in the system. As per their PI upper bound duration, jobs A and B need 16 and 32 seconds to finish, respectively. Suppose the deadlines for A and B are $d_A = 70$ and $d_B = 110$. Then, for A and B to meet their

[9] Gaussian distribution when negative elements are reset to 0



(a) $\mu = 1$ (b) $\mu = 1$ (c) Estimated duration from traces (d) Estimated duration from traces

Figure 6: Comparing the violation rate and the earliness for an M/G/1 queue with estimated job duration for $\beta = 5\%$

deadlines, they must start by $t = 10$ and $t = 30$, respectively. A "lazy" schedule where MRM schedules a job as late as possible is shown in Figure 7(a). This schedule leaves a lot of *slack* in the system; the server is free during the intervals $t \in [0,60]$, $t \in [70,90]$, and $t \geq 110$. To finish C as soon as possible, we can fill some of this slack. Then, as shown in Figure 7(b), we can schedule 60 seconds of it during the interval [0,60] and the remaining 20 during [70,90]. Hence, the earliest finish time for C is $t = 90$s. However, if C took 100 seconds to finish, then the earliest finish time for C will be $t = 130$s (Figure 7(c)). Figure 7 illustrates the key step in computing the earliest possible finish time—identify the *location* in time of slack in the system and then fill this slack with the new job (to the extent possible).

In general, for a cluster with multiple machines, computing the earliest feasible finish time for a new $(n+1)^{th}$ job, with processing time (or PI upper bound) p_{n+1}, when MRM has already accepted n jobs with processing times $p_1, \ldots, p_n$ and deadlines $d_1, \ldots, d_n$ is NP-hard. For the case of two machines where all jobs have the same deadline $d_1 = d_2 = \ldots = d_n = d_0$ and jobs cannot be "partitioned" (e.g., all jobs have 1 Map and 1 Reduce task), we can obtain a reduction from the PARTITION problem that is known to be NP-hard [19]. In our current implementation of MRM, we use a greedy heuristic to estimate earliest feasible finish times. Our MRM instantiation takes both Map and Reduce slots into account. MRM's computation of the earliest feasible finish time assumes that jobs contend only for Map slots and Reduce slots. However, since we estimate task durations using prior job execution statistics (see Section 3.2), network contention does determine *when* Map and Reduce slots are available. Thus, MRM accounts for runtime resource contention indirectly.

Deadline-aware scheduler. Once MRM commits to a particular deadline for a job, it is the scheduler's responsibility to make sure that job finishes before its deadline. Clearly, an FCFS (the Hadoop default) or a Fair scheduler [34] cannot always achieve this. However, an *earliest deadline first* (EDF) scheduler will suffice. Since users can only select a deadline from feasible finish times, there always exists a schedule S that satisfies all deadlines; if S does not schedule jobs in the order of increasing deadlines, then a simple pairwise re-ordering can be used to obtain an EDF schedule from S without violating any of the deadlines [18]. Note that, as we are using the upper-bound prediction interval as the duration of jobs, this rearrangement still keeps the violation rate under the target bound even for estimated job processing times. We have implemented an EDF scheduler in Hadoop.

There is a close relationship between our design choice to not allow users to specify arbitrary deadlines for their jobs and the simplicity of MRM's scheduler. An EDF scheduler suffices for MRM only because it allows users to choose a deadline from a restricted set of feasible deadlines. An alternative design can be to accept arbitrary deadlines and, if all of them cannot be met, to schedule jobs in such a way that average or maximum tardiness is minimized. However, in this scenario, predictability is sacrificed.

Note that MRM's design does offer the possibility that a user may not like any of the deadline choices MRM offers (or the price it charges), and she may decide to submit her job elsewhere. However, the pricing function in Eq.(13) assumes that delay-sensitive jobs do not go away. Instead, when the earliest offered deadline is later than their desired one, they accept earliest offered deadline. We also assume that users have enough money or tokens to afford the price for some deadline. As shown in Figure 4 the price eventually drops to zero, and hence, if a user does not have enough tokens, she can still submit her job but will have to accept a later deadline. These assumptions hold in *monopolistic* scenarios where there is a single cloud provider, e.g. the high performance computing clusters at universities or private clouds at enterprises. In fact, the university computing cluster that we used for our experiments provides an environment that satisfies all these assumptions. We plan, as part of future work, to explore the design of a pricing mechanisms for multiple (public) cluster providers.

6. EVALUATION

We evaluate MRM from three perspectives: the accuracy of its job processing time estimates, the predictability that it offers in job finish times, and the service differentiation that it enables for delay-sensitive jobs.

6.1 Evaluation of processing time estimation

Experimental methodology. To understand the performance of various estimation methods, we ran experiments on a Hadoop cluster provisioned across 40 servers; these 40 servers were

Bin	% Jobs	# Maps	# Reduces
1	38	1	0
2	16	[2-6]	0
3	14	[7-30]	0
4	8	[31-60]	6
5	6	[61-120]	12
6	6	[121-180]	15
7	4	[181-240]	18
8	4	[241-360]	24
9	4	[361-480]	30

Table 1: Workload

part of a larger shared compute cluster. Each server had 4 cores, and was configured with 2 Map and 2 Reduce slots (the default Hadoop configuration). The servers had 12 GB memory, 30–60 GB storage, and 1 Gbps maximum inter-server bandwidth. Our experiments shared the network with jobs submitted by other users to the shared compute cluster, and we had no control over the 40 servers allocated to us. Hence, the network topology changed across our experiments. This scenario matches the resource acquisition model prevalent in today's cloud services (e.g., Amazon AWS [2], Google [27]), in which users are more likely to setup MapReduce clusters on-demand (and hence get different sets of servers to work with at different times).

We use four types of MapReduce jobs in our experiments—*grep* (search for a pattern), *sort*, *wordcount* (wc), and *pi-estimator* (PiE). For *grep* and *sort* jobs, we generate input workloads using the *loadgen* class that comes with the Hadoop distribution. The *loadgen* class takes two parameters, *keepmap* and *keepreduce*, that control the percentage of records retained at the end of the Map and the Reduce phases. These two parameters help us control the map and reduce reduction factors for these jobs. *Grep* jobs have *keepmap=0.1%*, and *keepreduce=100%*, and hence, have huge reduction in data size at the end of the Map phase. *Sort* jobs have *keepmap* = *keepreduce* = 100%. *Wordcount* jobs compute the frequency of words in a document. The input data size for these jobs is proportional to the number of Map tasks because each Map task processes 64 MB of randomly generated text data. A *pi-estimator* job uses the Monte Carlo method to estimate the value of π. Compute-intensive *PiE* jobs differ from each other in the number of map tasks, and the number of Monte Carlo simulation steps. These four job types, consistent with the previously published evaluations on MapReduce (or Hadoop) clusters [16, 6], vary in terms of the balance between CPU and I/O in their execution.

Using previously published workload statistics for shared MapReduce clusters [34], we created a benchmark workload with jobs' sizes (number of Map and Reduce tasks) sampled from 9 different bins. Table 1 shows the distribution of jobs across bins, and the number of Map and Reduce tasks for jobs in each bin. For jobs belonging to bins 2-9, the number of Map tasks is not fixed (unlike bin 1); instead, we sample it uniformly from a range. This particular workload mix reflects a scenario where users submit lots of small experimental jobs, but also submit some large production jobs (as is the case at Facebook and Yahoo [34, 5]). The jobs in bins 7–9 contain many more Map tasks than the available Map slots; the over-subscription factor for the largest jobs is 6.

Method	Map	Reduce
OLS	0.99	0.96
Ridge	0.99	0.97
CCA	0.99	0.97
Kridge	0.99	0.94
KCCA	0.99	0.98
GP	0.99	0.98
Bin	0.92	0.87

Table 2: R^2 metric

We ran 10 experiments, with each run consisting of 100 jobs with the inter-arrival time between jobs sampled from a Poisson distribution. These 10 runs were split into three different load scenarios – *low*, *medium*, and *high* loads. The average job inter-arrival times were 100 and 200 seconds in the low load cases, 50 seconds under medium load, and 16 and 32 seconds under high load. The sizes of the 100 jobs in each run were as per Table 1, and we had (roughly) 25 jobs of each of the four job types. We divided the data from our 10 runs into training and test sets with each set containing five runs at different loads

Processing time estimation accuracy. We use the *Coefficient of determination* (R^2) to capture the *goodness of fit* of various techniques including out GP model; R^2 corresponds to the proportion of data variability that is explained by the model [25]. An R^2 value close to 1 denotes a good fit. Table 2 shows the R^2 metric for different methods. We combined the estimate for the Shuffle and Reduce phase into a single estimate (labelled "Reduce") for simplicity. The method labeled "Bin" represents the approach by Verma et al. [29] in which they assume that the same job is executed several times, and use the mean and variance of processing time from multiple runs to estimate a job's processing time. This assumption does not hold for our workload (we have similar not exactly the same job) and hence, the "Bin" method has lower R^2 value. While the other methods provide a good fit (R^2 values close to 1) for both Map and Reduce phase durations, we get slightly higher accuracy for Maps than Reduces. These results are similar to previously reported results for regression and CCA based methods [13, 12], and demonstrate that several techniques can be used for estimating MapReduce job processing times.

This result demonstrates that our approach of using prior statistics of job executions can provide reasonably accurate processing time estimates. It may be possible to further improve upon these results by matching the training data to the current system conditions can improve accuracy, but this may require detecting when the operating load has changed significantly and updating the analytical models. Hence, there exists a *accuracy vs. complexity* trade-off with regards to the processing time estimation.

PI estimation accuracy. In addition to estimating processing times, the predictability that MRM can offer also cru-

Run	OLS		GP		Bin	
	Map	Reduce	Map	Reduce	Map	Reduce
16	1	0	1	1	2	2
32	7	0	7	1	5	3
50	9	4	7	4	3	2
100	2	0	2	0	2	4
200	1	1	0	0	1	0
All	4	1	3.4	1.2	2.6	2.2
PI length (s)	222	107	112	44	530	53

Table 3: PI error and average PI length

cially depends on its ability to estimate an accurate and tight prediction interval. To understand the impact of using PI on predictability, we can compute whether the actual execution time of the map and reduce phase of a jobs falls within its PI. If it does not, then we have under-estimated the execution time and this can lead to a deadline violation. Table 3 shows the PI error percentage (i.e. percentage of jobs whose execution time does not fall inside the PI), with 95% PI, for the five test runs. Across individual runs (with 100 jobs each), we see less than 9% and 5% PI error in the worst case for Map and Reduce phase durations. The PI error numbers for maps are higher than expected in two cases; with 95% PI, we would expect close to 5% violations. However, when aggregated across the 500 jobs in the test data, the PI error rate is close to 1% for Reduce phase duration, and 4% for Map phase duration. Table 3 also shows the PI error for the "Bin" method [29]. Even though the R^2 accuracy for this method was not good (see Table 2), the average PI error is within the 5% bound.

Table 3 also shows the average *PI length* for the Map and Reduce phase duration. Comparing the PI length with the measured job execution times, we found that GP bounds are much tighter, especially for shorter jobs, compared to OLS bounds. The average PI length from GP is 3.5 (0.8) times the total Map (Reduce) duration; the corresponding value for OLS is 13.8 (2.4). Note that the PI length of the map phase for the "Bin" method is much larger than in case of the GP or the OLS method; thus, the "Bin" method achieves a lower PI error rate at the expense of a larger PI length. For the reduce phase, the PI length for the "Bin" method is half of the PI length for the OLS method but slightly larger than the GP method. However, it's PI error rate for the reduce phase is the highest; we suspect that this is again due to the fact that our workload has similar but not exactly the same jobs. Based on the results shown in Tables 2 and 3, we conclude that our GP method is better suited than the "Bin" method for real-world scenarios where the same job is not executed repeatedly.

6.2 Deadline Predictability

Next, we evaluate MRM's ability to meet job deadlines. We use the same methodology as described above to execute runs of 100 jobs. We selected three load scenarios—low, medium, and heavy—with average job inter-arrival times of 100, 50 and 32 seconds in the three cases. We conducted six experiment runs, two for each load scenario. When testing for a particular run of 100 jobs, we use the trace from other five runs as training data.

Load	% Tardy	Median tardiness (s)	Median earliness (s)
H	18	22.9	135.3
M	1	266.4	105.6
L	1	20.1	104.8

Table 4: Deadline violation rate under no-slack. H: Heavy, M: Medium, L: Low.

The most adversarial scenario for MRM to achieve predictability is when no job offers any slack. We refer to this setting as *no-slack*. As discussed in Section 5, we use our PI based approach to ensure that deadline violations are under a threshold β. For our experiments, we use two-sided 95% PI, and hence, MRM is configured to keep the violation rate below 2.5%.

Table 4 shows that MRM achieves violation rate below the 2.5% threshold for the medium and low load scenarios. However, for heavy load, the violation rate is 18%. For jobs that finished before their deadline, we show the median earliness value in seconds.

To analyze the cause for the high rate of deadline violations in the high load scenario, Figure 8 plots the difference between the estimated and the actual total execution time for the map and reduce phases for the 18 jobs that missed their deadline in the heavy load run. This figure points to MRM under-estimating processing times as the main reason for a higher deadline violation rate. Instead of under-estimating the processing time of at most 2.5% of jobs (because we use two-sided 95% PI), during the heavy load run, MRM under-estimated the processing time of either the map phase or the reduce phase or both for 18% of the jobs.

We find that the main reason for MRM's under-estimation of processing times for 18 jobs in the heavy load case is the high re-execution rate of map and reduce tasks. For the low and medium load runs, a total of 410 and 377 map tasks are re-executed across 100 jobs, receptively; hence, average failure rate is 4.1 and 3.77 per job. For the heavy load case, 808 map tasks are re-executed. Thus, failure rate for map tasks is twice as high. The situation is worse for reduce tasks: a total of 4, 5, and 22 reduce tasks are re-executed across the 100 jobs when under low, medium, and heavy load, respectively. We had Hadoop's straggler mitigation algorithm turned on during our experiments, and we speculate that the high task re-launch rate for the heavy load case was due to stragglers.

Figure 8: **Error in processing time estimates: no-slack, heavy load**

One solution for MRM to improve predictability under high task re-launch rate, and more generally when the processing time estimate is incorrect, can be to track the actual processing time of tasks of a job as they finish [21, 20]. For large jobs with lots of tasks, MRM can then detect if a job's progress is slower than expected (e.g., due to a high rate of task re-launches or incorrect processing time estimate). MRM can leverage this information in two ways: (1) it can compute the expected finish time of this job and inform the user about the additional delay beyond the deadline in completing the job, and (2) it can take the new estimate for the processing time for this job when computing the feasible deadlines for new arrivals. The latter is important to prevent cascading violations, i.e., a delayed job causing deadline violations for jobs waiting for it to finish. Note that cascading violations did not occur in our heavy load experiment. We plan to pursue this enhancement as part of our future work.

6.3 Service Differentiation

We evaluated a scenario where we force a set of randomly selected jobs to give slack, and verified that this helps other jobs (as deadline sensitive jobs) get earlier deadlines. This means that the slack by delay tolerant jobs can improve the service for delay sensitive jobs in Map-Reduce. However, to study the impact of the pricing function on MRM's service differentiation, we use trace-driven simulations. We use simulations here since our experimental setup is currently implemented to only take a trace of job arrival times as input. We have not yet implemented the framework in which we dynamically select the deadline for a job based on the price vs. deadline function at that point in time.

Simulation methodology. We extract processing time statistics from an experiment run with moderate load on a 40 node cluster. We then simulate errors in processing time estimates by adding Gaussian noise (with standard deviation $\sigma = 4.8$ seconds) to the actual processing times extracted from our traces. The error standard deviation is equal to 10% of the average job duration in our traces. We add a negative error value to the processing time estimate of (roughly) 95% of the jobs, and a positive error value for the rest. This is equivalent to using the 95% PI upper limit of processing time for computing deadlines in our experiments (see Sections 5 and 6). Although the run had only 100 jobs, we run the simulation for 500,000 jobs by entering the jobs again in a random order 5000 times. The arrival process is Poisson with 50s, 100s and 200s inter-arrival for low, medium and high load cases, respectively.

$\alpha = 2$			$\alpha = 1.5$			$\alpha = 1.1$		
H	M	L	H	M	L	H	M	L
1.3	2.2	2.5	0.01	0.01	0.01	0	0	0

Table 5: **% Tardy jobs; H: 50 s, M: 100 s, L: 200 s**

Load	$\alpha = 2$		$\alpha = 1.5$		$\alpha = 1.1$	
	median	95^{th}	median	95^{th}	median	95^{th}
H	0	37.9	14.7	50.2	35.9	65.9
M	0	9.6	14.7	24.6	35.9	40
L	0	0	14.7	14.7	35.9	35.9

Table 6: **Slack by delay-sensitive jobs (in seconds)**

We choose half of the jobs as delay-sensitive jobs, with the other half considered delay-tolerant. For any delay-sensitive job, we assume that users desire the earliest possible deadline, but they are willing to pay at most α times their processing time p_j (assuming 1 token buys 1 unit of processing time). We show results for three different values of α—2, 1.5, and 1.1. The different values of α capture various scenarios for the purchasing power of users. For delay-tolerant jobs, we model users as wanting to pay only p_j, the minimum possible price under MRM. We set $\kappa = 2$, which means that delay-tolerant jobs offer a slack at least equal to the average job duration even in an empty queue. On the other hand, when $\alpha = \kappa$, we expect delay-sensitive users to provide zero slack when the queue is empty.

Results. Table 5 shows the percentage of tardy jobs for different values of α. With larger α, where delay-sensitive jobs choose smaller slacks, the violation rate is higher. However, the interesting point to note is that the scenario with higher load has fewer violations. This is because load-aware pricing increases the slack in the system by adjusting the pricing function to the system load. Hence, the price increases for higher loads, because of which jobs are forced to provide larger slacks; the numbers in Tables 6 and 7 confirm this. Higher slack makes MRM more robust to processing time estimation errors, and this leads to fewer deadline violations.

Tables 6 and 7 show the slack offered by delay-sensitive and delay-tolerant jobs for different values of α. As we would expect, we see that delay-sensitive jobs offer lower slack than delay-tolerant jobs both in the median and 95^{th} percentile values. This is true even when $\alpha = 1.1$, the scenario in which delay-sensitive jobs offer higher slack due to lower purchasing power. Thus, MRM's pricing function discourages delay-tolerant jobs from offering lower slack, relative to the slack offered by delay-sensitive jobs. The second term on the RHS of (13) contributes to a higher price (that delay-tolerant jobs cannot afford) and hence, prevents them from offering lower slack (and jumping ahead of jobs waiting in the queue).

Another metric of interest is how much jobs are over charged because MRM over-estimated their processing time. Table 8 presents the ratio of the amount that a job paid over its real resource usage. These numbers are the same for all load val-

Load	$\alpha = 2$		$\alpha = 1.5$		$\alpha = 1.1$	
	median	95^{th}	median	95^{th}	median	95^{th}
H	43.9	56.1	43.9	60.6	43.9	68.4
M	43.87	43.9	43.87	44.03	43.87	43.9
L	43.87	43.9	43.87	43.9	43.87	43.9

Table 7: Slack by delay-tolerant jobs (in seconds)

Type	$\alpha = 2$		$\alpha = 1.5$		$\alpha = 1.1$	
	median	95^{th}	median	95^{th}	median	95^{th}
DS	2.4	2.98	1.8	2.23	1.31	1.64
DT	1.2	1.49	1.2	1.49	1.2	1.49

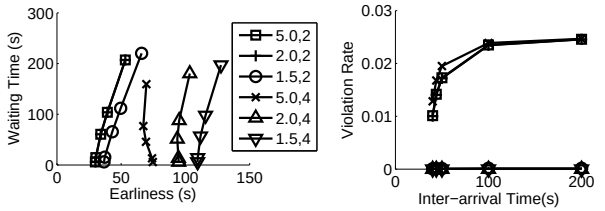
Table 8: Premium paid over resource usage; DS: Delay-sensitive, DT: Delay-tolerant

ues; we omit the mean statistics because they are just 2% above the medians. For $\alpha = 2$, we expect delay-sensitive jobs to pay twice their resource usage but they actually paid about 2.4 times. One way to address this issue is to reimburse the extra amount charged.

Comparison against other schedulers. Using simulations, we compare MRM against FCFS and a non pre-emptive priority scheduler that prioritizes delay-sensitive jobs over delay-tolerant jobs. As discussed in Section 3, FCFS can provide predictability but no service differentiation. The priority scheduler provides the best possible service differentiation without pre-emption, but it achieves this at the expense of predictability for delay-tolerant jobs.

Figure 9(a) shows the average waiting time for delay-sensitive jobs under different schedulers for five different average job inter-arrival times when $\alpha = 2$ and $\kappa = 2$. (Later we compare for different values of α and κ.) The bars represent the 5^{th} and 95^{th} percentiles for the waiting time. Under heavy load, average job inter-arrival times of 40, 44, and 50 seconds, delay-sensitive jobs experience queueing delays under FCFS, whereas MRM leverages the slack offered by delay-tolerant jobs to provide differentiated service to delay-sensitive jobs. As expected, the priority scheduler achieves the shortest waiting time, but as shown in Figure 9(b), its deadline violation rate is unacceptable.

The reason for reduction in the waiting time for delay-sensitive jobs under MRM or priority scheduler is that they can jump ahead of (delay-tolerant) jobs waiting in the queue. We did a pairwise comparison between MRM and FCFS in terms of the number of delay-sensitive jobs that jump ahead of some job(s). For example, when average job inter-arrival time is 40s, by comparing the completion order of jobs under FCFS and MRM, we observed that 35.4% of the delay-sensitive jobs jumped ahead of some job(s) under MRM. However, it is possible for a delay-sensitive job to jump ahead of some other job(s), only if there is at least one delay-tolerant job waiting in the queue when the delay-sensitive job arrives. When restricted to delay-sensitive jobs that saw at least one delay-tolerant job in the queue upon arrival, we find that 47.7% of delay-sensitive jobs jump ahead of earlier jobs when compared with the FCFS schedule.



(a) Wating time vs Earliness (b) Violation Rate

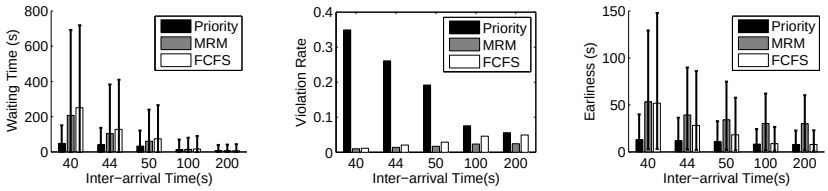
Figure 10: Changing the purchasing powers (legend numbers are α and κ)

Achieving this predictability and service differentiation is not free, but it affects the earliness of the jobs. Figure 9(c), shows that MRM has larger earliness than other policies. The earliness for FCFS policy increases by load because the prediction interval method is conservative (as discussed in Section 5). The difference between MRM and FCFS decreases for smaller inter-arrival times (higher load) as for higher loads the slack provided by deadline tolerant jobs are filled by deadline sensitive jobs and they cannot run early. The earliness for Priority queue does not increase for higher loads as much as FCFS does because deadline sensitive jobs in Priority queue see smaller queue as they only see deadline sensitive jobs ahead of them.

Changing the purchasing powers. We compared the waiting time of delay-sensitive jobs and earliness of MRM in Figure 10(a) for different values of α and κ. [10] Each line represents an experiment with different job inter-arrival times and larger waiting times are for smaller inter-arrival times. The diagram shows that there is a trade off between waiting time for delay-sensitive jobs and earliness of jobs in the system. This trade-off can be adjusted by parameters α and κ that affect the purchasing power of delay-sensitive vs delay-tolerant jobs. Note that for $\kappa = 2$, the lines for $\alpha = 5$ and $\alpha = 2$ overlap with each other. This is because delay-sensitive jobs already provide no slack for $\alpha = 2$ and increasing α to 5 does not help, instead we should increase prices by larger κ to force delay-tolerant jobs provide more slack.

Now we explain the curves in Figure 10(a) by justifying the trend of earliness vs. load (waiting time). Earliness mostly comes from the fact that delay-tolerant jobs give slack and there is no delay-sensitive job to fill the slack. We believe that there are two opposing load-dependent forces affecting earliness: 1) As the price function is load-dependent, the price increases as load goes up. So fewer delay-sensitive jobs afford to jump ahead of others, which causes larger earliness for higher loads (waiting time) 2) In low load, there are scenarios that even if a delay-tolerant job gives large slack, no delay-sensitive job arrives to fill its slack, so it finishes early. So we expect smaller earliness for higher loads. We see that the first case is dominant in all curves except for

[10] Here, we fixed the purchase power of delay-tolerant jobs to 0.5. Someone may also fix κ or α instead.



(a) Waiting time of delay-sensitive jobs (b) Violation Rate (c) Earliness

Figure 9: Comparing the violation rate and the earliness for an M/G/1 queue with estimated job duration for $\beta = 5\%$

$\alpha = 5, \kappa = 4$. This is because for other cases, either κ is too small that does not force delay-tolerant jobs give large slack or α is too small for delay-sensitive jobs to be able to fill the slack. However, for $\alpha = 5, \kappa = 4$, the second force is dominant because delay-tolerant jobs give such large slack that is only filled in high load. Also α is large enough that the rise of price function does not affect this trend.

Figure 10(b) compares the violation rate for different values of α and κ. Note that when the purchasing power of delay-sensitive jobs is low comparing to κ, the violation rate is almost zero as they provide enough slack to compensate the job duration under-estimation.

7. RELATED WORK

Our work in developing MRM is related to prior work in several areas.

Computing processing time estimates. Ganapathi et al. [12] use a kernel CCA-based method to estimate a job's processing time. Our results in Section 3.2 extend their results by evaluating a larger set of approaches on a much larger dataset. Like MRM, Kavulya et al. [17] also use Hadoop-specific features to characterize jobs. To estimate the processing time of job A, they first do a N-nearest neighbor search in the feature space for jobs similar to A executed in the past, and then use *locally-weighted linear regression* to estimate A's processing time. However, unlike MRM, their goal is to detect performance problems on the cluster by comparing estimates of job processing times with actual runtimes. As such, it is crucial for their work to have fairly precise processing time estimates; MRM's estimation technique is more conservative because it attempts to bound the worst case behavior in order to avoid deadline violations.

Verma et al. [29] consider a scenario where the same job is executed several times, and use the mean and variance of processing time from multiple runs to estimate a job's processing time. As shown in Section 6.1, this strategy does not work well for previously unseen jobs. Also, in a multi-tenant environment, we observed higher variance in job processing times than reported in [29]. We believe this is because in [29] jobs are executed one at a time so there is no inter-job resource contention, and our workload is more diverse.

Similar to our approach of using prediction intervals to set deadlines, Mu'alem and Feitelson use a conservative approach for scheduling jobs on an IBM SP2 system. However, instead of estimating service times, they ask users to provide an estimate, and then recommend using twice that value for making scheduling decisions [22]. Our prediction interval based approach is a more principled way for setting conservative deadlines.

Several other approaches have been proposed for estimating the (remaining) processing time of a task/job once it starts executing [6, 21, 20]. These are not directly applicable to MRM because it needs *a priori* estimates.

MapReduce schedulers. Scheduling MapReduce-style workflows has received significant attention. Jockey [11] is closest in spirit to MRM. It provides latency guarantees and dynamic resource allocation to maximize jobs' utility. Jockey's latency guarantees achieve predictability but only for production jobs that execute repeatedly. This is because Jockey uses statistics from past runs to estimate a job's processing time. In contrast, MRM can estimate the processing time of a job that has not executed previously provided similar jobs have been executed in the past. Another key difference between Jockey and MRM is that Jockey takes the function describing the utility of a job completing at time t as input and does dynamic resource allocation to maximize this utility while in MRM, the system offers a price vs. deadline curve to user. The deadline picked by a user in MRM can be interpreted as time t at which user's utility function intersects MRM's price vs. deadline curve.

Quincy [16] uses a network flow formulation to achieve two goals—optimize the data locality of jobs and enable fair access to the cluster irrespective of job size. Zaharia et al. [34] implement delay scheduling to improve the data locality of short jobs. In both cases, the improvement in data locality and fairness that any particular job receives depend on other jobs simultaneously seeking access to the cluster and so cannot be used to offer predictability guarantees.

Verma et al. [30] evaluate three variations of the earliest deadline first (EDF) scheduler in Hadoop—1) vanilla EDF, 2) EDF with minimum allocation of Map and Reduce slots to a job to meet its deadline, and 3) EDF with dynamic reallocation of spare Map/Reduce slots. They make the same assumption as [29]—a job is executed multiple times over new datasets—to estimate job processing times, and every job selects a deadline uniformly at random from $[2p_j, 4p_j]$, where p_j is the job's processing time estimate. However, since users' choice of deadline is not restricted to a feasible

set, all three schedulers incur a large number of deadline violations. Verma et al. [30] observed 20–40% deadline violations with high levels of tardiness, when running the cluster at a load comparable to the low load scenario we considered in Section 6. In contrast, our results validate MRM's design philosophy that to achieve predictability we need to restrict deadline choices to a feasible set.

Market-based mechanisms. There is a rich history in applying market-based approaches for resource management, in particular for cluster computing environments [31, 26, 7, 27]. The key idea in these approaches is to create a market that can bring sellers (owners of computational resources) and buyers (users with jobs to execute) together. Often, prices for resources are set using either repeated auctions or trading [31, 27, 7]. These auction-based systems are best suited for *federated* clusters with multiple independent owners for resources.

Pricing in MRM is designed for clusters owned by a single entity. In such a setting, the overhead of repeated auctions can be avoided by having a centralized entity set prices [26]. Here, users are assumed to be *price takers*, and the main challenge lies in setting resource prices to achieve desired objectives—predictability and service differentiation in case of MRM.

Congestion pricing. MRM's price-deadline curve is similar in spirit to prior work on pricing to alleviate congestion in other resources such as network bandwidth [24], freeways [32], and parking spots [14]. Flat rather than per-byte pricing has emerged the dominant pricing strategy for network access, as the former has been seen to encourage greater network access. However, flat pricing is unsuitable for shared multi-tasking clusters due to its inability to differentiate jobs based on their delay tolerance. Similar to MRM's load-sensitive pricing, there have been recent proposals to price parking based on current congestion levels [4].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new service model for shared MapReduce clusters that provides both predictability of finish times and service differentiation. Our design, MRM, forces users to reveal their true finish-time preferences using a novel price-deadline curve formulation, incentivizing them to offer slack that enables the system to accommodate users with delay-sensitive jobs. Our experiments show that, despite significant performance variability in clusters, it is possible to achieve near-perfect predictability of finish times when users can be incentivized to offer slack.

MRM's design opens up many interesting directions for future work. First, for jobs with multiple MapReduce phases (e.g., Hive [3] and Pig [1] jobs), MRM would need to identify the *critical path*—the task sequence with the longest finish time—in the DAG. Second, to deal with deadline violations, MRM can offer a rebate to users when it violates their deadlines, or appropriately define service-level agreements to guarantee a certain level of predictability, thereby setting user expectations. Lastly, mechanisms to detect and penalize malicious users (e.g., those who inflate feature vectors) would be essential in a production system.

9. REFERENCES

[1] Apache Pig. `http://pig.apache.org`.

[2] Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[3] Hive. `http://hive.apache.org`.

[4] Sfpark. `http://sfpark.org/how-it-works/`.

[5] Yahoo! gridmix3. `http://developer.yahoo.com/blogs/hadoop/posts/2010/04/gridmix3_emulating_production/`.

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

[7] R. Buyya, J. Giddy, and D. Abramson. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In *Active Middleware Services*, 2000.

[8] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. the MIT Press, 2006.

[9] D. Bertsekas and R. Gallagher. *Data Networks*. PHI, 1992.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[11] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Eurosys*, 2012.

[12] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-Driven Workload Modeling for the Cloud. In *Workshop on Self-Managing Database Systems*, 2010.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordand, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, 2009.

[14] R. Garcia and A. Marin. Parking capacity and pricing in park'n ride trips: A continuous equilibrium network design problem. *Annals of Operations Research*, 2002.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

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

[17] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proceedings of Cluster, Cloud and Grid Computing*, 2010.

[18] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2008.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[20] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[21] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the Progress of MapReduce Pipelines. In *ICDE*, 2010.

[22] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[23] C. A. O'cinneide. Phase-type distributions: Open problems and a few properties. *Stochastic Models*, 15(4):731–757, 1999.

[24] A. Odlyzko. Internet pricing and the history of communications. *Computer Networks*, 2001.

[25] T. Ryan. *Modern regression methods*. Wiley, 2009.

[26] I. Stoica, H. Abdel-Wahad, and A. Pothen. A microeconomic scheduler for parallel computers. In *IPPS*, 1995.

[27] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. Using a market economy to provision compute resources across planet-wide clusters. In *IPDPS*, 2009.

[28] J. Tan, X. Meng, and L. Zhang. Delay tails in MapReduce Scheduling. In *SIGMETRICS*, 2012.

[29] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *ICAC*, 2011.

[30] A. Verma, L. Cherkasova, and V. S. Kumar. Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle. In *NOMS*, 2012.

[31] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spwan: A distributed computational economy. *IEEE Trans. on Software Engineering*, 1992.

[32] M. B. Yildirim and D. W. Hearn. A first best toll pricing framework for variable demand traffic assignment problems. *Transportation Research Part B: Methodological*, 2005.

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, A. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

[34] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.