Data-Driven Batch Scheduling

John Bent
High Performance Computing
Los Alamos National Lab *
Los Alamos, NM
johnbent@lanl.gov

Timothy E. Denehy
Computer Science
University of Wisconsin
Madison, WI
tedenehy@cs.wisc.edu

Miron Livny
Computer Science
University of Wisconsin
Madison, WI
miron@cs.wisc.edu

Andrea C.
Arpaci-Dusseau
Computer Science
University of Wisconsin
Madison, WI
dusseau@cs.wisc.edu

Remzi H.
Arpaci-Dusseau
Computer Science
University of Wisconsin
Madison, WI
remzi@cs.wisc.edu

ABSTRACT

In this paper, we develop data-driven strategies for batch computing schedulers. Current CPU-centric batch schedulers ignore the data needs within workloads and execute them by linking them transparently and directly to their needed data. When scheduled on remote computational resources, this elegant solution of direct data access can incur an order of magnitude performance penalty for data-intensive workloads. Adding data-awareness to batch schedulers allows a careful coordination of data and CPU allocation thereby reducing the cost of remote execution.

We offer here new techniques by which batch schedulers can become data-driven. Such systems can use our analytical predictive models to select one of the four data-driven scheduling policies that we have created. Through simulation, we demonstrate the accuracy of our predictive models and show how they can reduce time to completion for some workloads by as much as 80%.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Batch processing systems*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms

Design, Measurement, Performance

1. INTRODUCTION

Scheduling batch workloads in a distributed environment is challenging. So challenging, in fact, that the data needs of these workloads are mostly ignored by current batch schedulers. Batch schedulers can ignore these data requirements by transparently transforming distributed environments to resemble the home environments of executing programs. In doing so, they enable a technique, remote I/O, that allows the programs to access remote data directly.

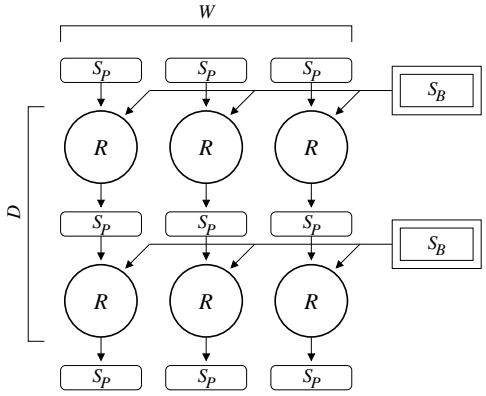

Figure 1: A Canonical Batch-Pipeline Workload. Circles represent jobs, double-edged rectangles represent batch data volumes, and single-edged rectangles are private volumes in this example workload.

Instead of considering data requirements, batch schedulers are CPU-centric in that they consider only the computational needs of workloads. Data movement in these scheduling systems therefore happens as an unplanned side-effect of job placement. As a job executes and initiates I/O operations, only then does data flow. For many years, this approach has worked well and many important problems in genomics, video production, simulation, document processing, data mining, electronic design automation, financial services, and graphics rendering have been solved using the increased computational power offered by batch computing.

However, two recent trends now threaten this technology. First, recent innovations in grid computing allow users, and batch schedulers, access to an increasingly distributed set of remote computational resources [9]. The second trend is that datasets are increasing in size and this growth has been observed to outpace the corresponding increase in the ability of computational systems to transport and process data [10]. Techniques which once worked well for CPU-intensive workloads in a local environment can suffer orders of magnitude losses in throughput when applied to data-intensive workloads in remote environments [5, 12].

*LANL Technical Information Release: LA-UR 09-02116

DADC'09, June 9–10, 2009, Munich, Germany.

As Denning noted in early computer systems that memory management and process scheduling must be considered in a coordinated fashion [7], here we make the similar observation that batch scheduling systems cannot schedule their workloads independently of storage management.

We offer one approach to this problem: *data-driven batch scheduling strategies* which consider both the data and CPU requirements of batch workloads. We tailor our approach to address the situation in which a user has data stored on a home storage server and has access to a remote compute cluster in which CPUs are plentiful but storage is scarce. Our contribution is to define different scheduling strategies for this scenario and to develop a predictive model that allows appropriate strategy selection such that a scheduler can minimize a workload's total time to completion.

To do so, we first codify some simplifying assumptions about batch workloads into a new abstraction, a canonical batch-pipeline workload. Using three representative canonical workloads and defining four possible scheduling strategies, we examine the effect of changing workload and environmental characteristics and identify potential pitfalls for each of the scheduling strategies. Using simulation, we quantify these effects by analyzing several performance metrics such as CPU utilization, wide-area network traffic, and the total completion time of the workload. We then formalize predictive analytical models for each of the four possible scheduling strategies and demonstrate high levels of accuracy in their predictive abilities. Finally, we show that across the entire set of experimentation that the models usually predict the "best" strategy and more importantly that they *never* predict the "worst."

Background. Batch computing refers to a system of computing in which a user does not interactively dispatch programs for execution but rather delegates this responsibility to a batch scheduler. The batch scheduler then in turn dispatches the programs for execution, monitors their status, and returns their output to the user.

To use batch scheduling systems, users describe the jobs they want executed along with instructions about how to execute these jobs. Although a batch workload might consist of only a single job, it is more typical for many such jobs to be described together and submitted to the batch system within a single workload.

There are several reasons why a user might submit several jobs simultaneously instead of just a single job. One such reason might be that the user has now automated, into a single batch submission, work that was previously done interactively. For instance, this work might consist of running a single job to produce some output data, then running a child job to perform some transformation of the data produced by the first job.

Users often submit multiple parent-child job structures simultaneously such that their workloads consist of a two-dimensional structure of some number of vertical sequences of jobs [20] called a *batch-pipeline* workload. Each vertical sequence in a workload, a *pipeline*, is comprised of the same set of jobs. The difference between each pipeline is that each is initiated with either different input parameters or files (*e.g.* performing a parameter sweep).

A set of pipelines is a *batch.* A *volume* is storage allocated to hold particular data. In a batch-pipeline workload, there are three types of data which result in different data sharing behaviors. *Endpoint* data refers to the unique inputs to each pipeline as well as to their output data which must be extracted from the compute system and returned to the user. *Pipeline* data refers to the data passed from parent to child. By definition, pipeline data is not output data and can be discarded by the system and not returned to the user. Finally, *batch* data is data which is read-shared across multiple pipelines.

We now present our target scenario: using remote computational resources. Although many users have local access to sufficient computational resources, there will always exist some class of users whose needs exceed local availability.

Although wide-area sharing of untrusted and arbitrary personal computers is a possible platform for batch workloads [19], we believe that a better platform for these types of throughput-intensive workloads is one or more clusters of managed machines, spread across the wide area. We assume that each cluster machine has processing, memory, and local disk space available for remote users, and that each exports its resources via a CPU sharing system. An obvious bottleneck of such a system is the wide-area connection, which must be managed carefully to ensure high performance.

Simplifying Assumptions. Our focus is the scheduler's ability to formulate a data allocation and job placement plan. Due to the large number of variables involved, we concentrate on the base case of executing a single workload on a single compute cluster and assume three additional simplifications. First, we assume that the scheduler has access to detailed and accurate information about the workload and about the compute infrastructure. Second, we assume that the workloads are uniform in structure; we refer to such uniform batch-pipeline workloads as *canonical*. Finally, we assume that the compute clusters consist of homogeneous machines.

To reduce the number of variables involved in developing a data-driven batch scheduler, we introduce the concept of canonical batch-pipeline workloads. Generally speaking, a canonical workload is a batch-pipeline workload in which all jobs have similar runtimes, all batch volumes are of the same size, all endpoint and pipelines volumes are of the same size, each pipeline is a single straight line (*i.e.* no multiple job dependencies), and each job at the same depth in the workload reads from the same batch volume.

For the sake of exposition, we introduce a new term, *private data*. Previously, when discussing batch-pipeline workloads, we referred to three different data types, batch, endpoint and pipeline. Both endpoint and pipeline data are accessed by only a single pipeline whereas batch data are shared across many. We therefore combine endpoint and pipeline data into an abstraction which we term private. By combining these data types we reduce the number of variables needed to describe the workloads. The trade-off is that it becomes harder to differentiate between endpoint-intensive workloads and pipeline-intensive workloads.

A canonical workload is shown in Figure 1. Canonical workloads simplify data-driven scheduling because they can be represented using only five variables. W and D define their width and their depth. S_B is the size of each batch volume and S_P is the size of each private. Finally, the random variable R describes the distribution of job runtimes; $\bar{R}$ is the mean runtime and $SD(R)$ is the standard deviation. Additionally, we define the total number of jobs in the workload, $J = D \cdot W$, the total amount of batch data, $S_{TotB} = D \cdot S_B$, and the total amount of private data, $S_{TotP} = (D{+}1) \cdot W \cdot S_P$.

The compute cluster is assumed to be homogeneous and is represented using five variables. The number of compute nodes is N_{CPU}, the total amount of storage (*i.e.* the sum of the storage from each node) is S, the rate of failure is F, the bandwidth between the cluster and the home (remote) storage site is BW_R, and the local bandwidth within the cluster is BW_L.

2. SCHEDULING STRATEGIES

Using the eleven variables describing the workload and the environment, a batch scheduler can choose from several scheduling strategies. Each choice may lead to a different traversal order through the workload with various throughput implications. The objective of the scheduler is to maximize the throughput of the workload (*i.e.* minimize the total time to completion).

Pitfalls. When choosing a strategy, the scheduler tries to avoid two potential pitfalls which may reduce throughput. The first is the underutilization of the available CPUs, which can happen in two different situations. The first occurs when the size of private volume limits the number of jobs which can run concurrently (*i.e.* the number of allocatable private volume input and output pairs is fewer than the number of CPUs). In this case, it is necessary to impose *concurrency limits* on the workload, resulting in idle CPUs. CPU underutilization may also be caused by the imposition of *barriers* within the workload. In a situation in which the maximum number of batch volumes which can be concurrently allocated is fewer than the total number of batch volumes, the scheduler can use barriers to ensure that only a subset of the batch volumes is accessed at any time. If the individual pipelines exhibit variability in their runtimes, these barriers will result in idle CPUs. The second potential pitfall is the need to remove and then subsequently *refetch* batch volumes during the execution of a workload such that the same data traverses the wide-area network multiple times.

Defining Possible Scheduling Strategies. As shown in Figure 2, we have identified four possible scheduling strategies.

The All Strategy. In the unconstrained case, *All*, shown in the left column in Figure 2, every volume in the workload (*i.e.* all private and batch) fits within the total available storage. In such a scenario, the planning is straightforward as no possible schedule can result in adverse effects. Formally, All is possible in a canonical batch-pipeline workload whenever the total of all batch data (S_{TotB}) and all private data (S_{TotP}) fits within the total amount of cluster storage (S):

$$S_{TotB} + S_{TotP} \leq S. \quad (1)$$

Figure 2 shows a workflow traversal using the All strategy for a workload of width and depth three. This traversal assumes that jobs are synchronized such that jobs which begin executing at approximately the same time also complete at approximately the same time. Further assumed is that there are at least three compute nodes so that each pipeline may execute concurrently. Given these assumptions, the traversal will proceed such that the executing pipelines remain approximately, but not perfectly, in synchrony.

The traversal begins with the initial state in which no jobs are executing and no volumes are allocated, as shown in the far left picture. In the next figure, the scheduler has allocated all volumes and has begun executing the first job in each of the three pipelines. As the first job finishes, its child can immediately begin executing because all necessary volumes have already been allocated. This continues until the final job finishes and the volumes are de-allocated (not shown). Notice that because all volumes can be concurrently allocated there are no limits on concurrency (*i.e.* there are always three jobs executing), there are no barriers imposed, and there is no refetching of batch volumes. This is true for only the All strategy; as we examine more constrained strategies, we will begin to see barriers, concurrency limitations, and refetching of batch data.

The AllBatch Strategy. If all of the batch volumes fit without sufficient remaining space for all of the private volumes, an *AllBatch* strategy is possible as long as at least one pipeline can simultaneously allocate two private volumes (*i.e.* each job must have access to both its private input and ouput volumes). The minimal allocation for this strategy is shown in Figure 2. AllBatch is formally possible whenever the sum of all batch data and the data needed for two private volumes fits within available storage:

$$S_{TotB} + 2S_P \leq S. \quad (2)$$

As all batch volumes can be simultaneously allocated, no barriers need to be imposed, nor will any batch volumes need to be refetched. However, concurrency limits may cause an underutilization of the computing capacity if fewer than N_{CPU} pipelines can simultaneously execute. To explain this we use another derived variable, C, which refers to the number of pipelines which can concurrently execute.

A workflow traversal for AllBatch is shown in Figure 2 for the maximally constrained case in which only a single pipeline can execute. This traversal shows the loss in utilization as only a single job can execute at any given time. Each pipeline must execute in its entirety before another can begin, as shown in the transition from the fourth to the fifth picture. Notice the depth-first traversal that results from this constrained schedule.

The Slice Strategy. Figure 2 shows the minimum allocated volumes for the Slice scheduling strategy, which is possible whenever an entire horizontal *slice* of the workload can be simultaneously scheduled. A horizontal slice of the workload has a storage requirement of one batch volume and one private volume for each pipeline *plus* at least one more private volume so that at least one job has access to both of its private volumes. More formally, Slice is possible whenever

$$S_B + S_P \cdot W + S_P \leq S. \quad (3)$$

Because the entire horizontal slice will execute before the workload descends, no batch refetch is necessary when using Slice. However, barriers are possible and concurrency limits may occur when the storage remaining after allocating an entire horizontal slice allows only a subset of the pipelines to allocate a second private volume for their outputs.

A workflow traversal for the Slice allocation is shown in Figure 2 for the maximally constrained case. Notice that in the steady state when all pipelines are allocated the maximum number of executing jobs is 1. This is not true for the entire duration however, as two jobs are able to execute concurrently at the very beginning and end of the workload. As the workload ramps up, an increasing number of private volumes are allocated. In this case, the maximum number of private volumes that can be allocated is four. In the steady state, after each of the three pipeline has begun, one private volume for each pipeline is allocated leaving space for only one additional private volume thereby allowing only a single job access to both its input and output private volumes and limiting concurrency to one.

However, during the ramp up phase, not every pipeline has begun and there is additional space. Since four private volumes can be allocated at any time, this allows two jobs to execute at the very beginning of the traversal. The same effect occurs at the end after the first pipeline finishes. None of its private volumes need to remain allocated, thereby freeing storage, allowing an additional private volume to be allocated and an additional job to execute.

Finally, notice that although there may be CPU underutilization due to barriers (not shown in Figure 2) and due to concurrency limits (as shown), the Slice strategy avoids refetching any batch data. Because every private volume from a particular slice is concurrently allocated, the workflow is able to completely exhaust all jobs at a particular depth before descending. This exhaustion of a depth allows the Slice strategy to avoid any batch volume refetch even when it is maximally constrained. Also notice the breadth-first traversal that occurs using this strategy.

The Minimal Strategy. The most constrained possible allocation, as shown in Figure 2, is *Minimal*, in which only a *partial* horizontal slice of the workload can execute before storage is exhausted and the workload is forced to move deeper. A Minimal strategy is possible as long as at least one job can have access to its batch volume and both of its private volumes:

$$S_B + 2S_P \leq S. \quad (4)$$

Minimal suffers from all three pitfalls; it will definitely refetch batch volumes, and it may suffer underutilization of computation due to either concurrency limitations or barriers. Each batch vol-

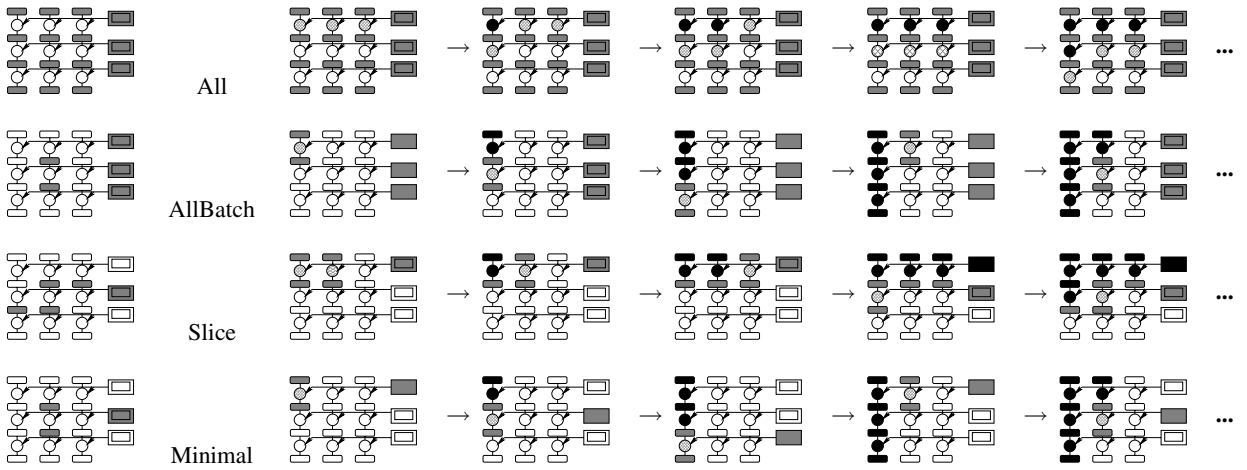

Figure 2: Minimum Allocations and Workflow Traversals. The pictures in the left column show the minimum number of volumes that must be simultaneously allocated for each scheduling strategy. The sequences on the right illustrate a workflow traversal for each strategy. Pending jobs and pending volumes are unfilled, executing jobs and allocated volumes are shaded, and finished jobs and finished volumes are filled in black.

ume will be refetched at least once for every partial horizontal slice of the workload that executes. The expected concurrency, in a Minimal allocation may also be constrained. By definition, whenever the Minimal allocation is possible and the Slice allocation is not, the maximum concurrency of the workload will necessarily be smaller than the width of the workload. However, whether this concurrency limit actually results in an underutilization of CPU depends on the number of nodes in the compute cluster.

Figure 2 shows a traversal for the Minimal strategy in the maximally constrained case. This traversal appears similar to that for the AllBatch strategy except that each batch volume must be removed before the workflow can descend such that each batch volume must be fetched for every pipeline. Notice that, as was the case for the AllBatch strategy, the Minimal strategy with concurrency constraints results in a depth-first traversal.

The Remote Strategy. Finally, there exist workloads whose combination of batch and private volume sizes is sufficiently large that none of these strategies is possible. These workloads could still execute if they do not attempt to cache their batch data or to buffer their private data, but rather use remote I/O. We refer to this as the Remote strategy. Although this approach is feasible, we do not consider it further because we are more interested in the challenges involved when data allocations are possible but constrained.

Determining Possible Strategies. Using the formulae for determining when each strategy is possible reveals that multiple scheduling strategies may be possible for any particular workload. Figure 3 plots these formulae for different values of workload width and workload depth. The areas under each line indicate where each strategy is possible for all values of batch volume size (S_B) on the x-axis and private volume size (S_P) on the y-axis. These volume sizes are shown as a percentage of the total storage, S. They are presented in tabular format to show the effect of increasing workload depths and widths.

There are several points to notice in these graphs. The first thing to notice is that in all cases, the maximum batch volume approaches the full 100% of S but that the maximum private volume is only 50% of S. This is due to the fact that an executing job must have access to only a single batch volume but to two private ones.

Further, we see that each strategy is affected differently by increasing values of workload depth and width. Comparing across the top row as the workload width increases, we observe that only the possible areas for the AllBatch and Minimal strategies are robust to increasing workload widths. Conversely, by looking at the left-most column of graphs, we see that only the Slice and Minimal strategies are robust to increasing the depth of the workload.

These graphs allow comparison of the different possible areas for each of the four strategies which can aid in the scheduler's choice. Notice initially that only in the base case in which W and D are both 1 are the possible areas the same for each strategy. For each other possible combination of W and D, the areas are different. For very large data sets in which the batch volume size exceeds the total cluster storage or where two private volumes do, there will be no allocations possible. For smaller volume sizes, in some cases there may only be one possible strategy. For example, for workloads with both a large private size and a large batch size, only the Minimal strategy is possible. Conversely, for small values of S_P and S_B, all strategies are possible. The key observation here is that when multiple possibilities exist, the scheduler needs some additional criteria by which to choose a scheduling strategy.

Winnowing the Strategies. To simplify our evaluation, we remove the All strategy from consideration. All is not interesting since the entire set of volumes can be allocated and the scheduler need make no storage allocation decisions. This problem reverts to the already addressed problem of making job placement decisions solely in regards to available computational resources. Note that another posible strategy, AllPrivate, has not been discussed here due to space constraints. See [4] for a full discussion of it as well as for an explanation of how it is a strict subset of, and is strictly worse than, the Slice allocation.

3. PREDICTIVE ANALYTICAL MODELS

To determine which strategy is *preferable* when multiple are possible, we develop analytical models for predicting the runtimes for each of the scheduling strategies. These models are relatively simple; they consist of fewer than 500 lines of codes and use only the eleven workload and environment variables as defined in Section 1.

Notice that many of the low level characteristics, such as the disks and buffer caches, of the cluster environment are *not* considered in these predictive models. As we will see, this simplification may cause some absolute inaccuracies in the model's ability to predict absolute runtimes but does not adversely affect the model's

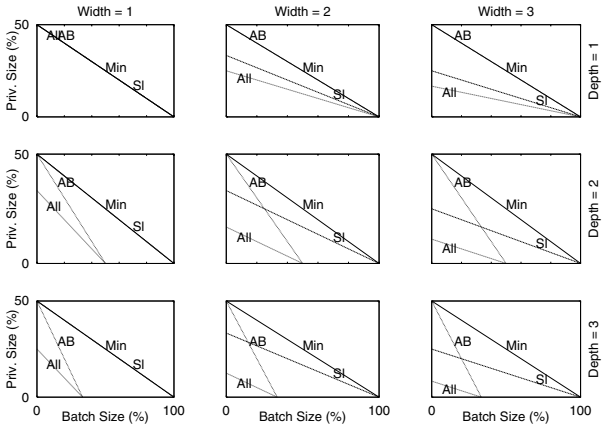

Figure 3: Possible Volume Sizes. These graphs show, as the area under the lines, when each scheduling strategy is possible for varying batch and private volume sizes. The names of the strategies have been abbreviated. The x-axis is the size of the batch volumes and the y-axis is the size of private, both shown as a percentage of storage.

ability to predict the relative performance of the different strategies. As this relative performance is what is used for predictive scheduling, this loss of absolute accuracy is a reasonable trade-off.

To predict the total time to completion for these workloads, we compute how many phases will be required to run these workloads. The number of these phases is generally the total width of the workload divided by our anticipated degree of concurrency. Each phase then consists of executing some number of pipelines.

The runtime for each phase is computed to be the total amount of compute time within that phase plus the time needed to access the total amount of private and batch data for a pipeline within that phase. Notice that our model assumes that the runtimes for all pipelines within a phase can be computed as a single number. Also it does not consider any costs due to contention that may be incurred depending on the degree of concurrency. As we will see in our evaluation, these simplifications may cause the model to underpredict the absolute runtimes but does not effect its relative predictions comparing across strategies.

Predicting Runtime for All. Specifically for All, the total time to completion includes the time for a cold phase, T_{Cold}, during which the batch data is fetched from the home storage server, and some number of warm phases, N_{Warm}. Each warm phase takes time T_{Warm}, during which the batch data is fetched from a local cache within the compute cluster. Thus, the total time for a workload is

$$T = T_{Cold} + N_{Warm} \cdot T_{Warm} \tag{5}$$

The runtime for the cold phase consists of the sum of the time to fetch the batch data from the remote node, T_{ColdB}, the time to read the private data T_P, the time to write the private data T_P, and the compute time, T_{CPU}:

$$T_{Cold} = T_{ColdB} + T_P + T_P + T_{CPU}. \tag{6}$$

The time to read the batch data remotely, T_{ColdB}, is the dividend of the total amount of batch data, S_{TotB}, and the serial bandwidth of sending data through the remote and local networks, BW:

$$T_{ColdB} = \frac{S_{TotB}}{BW} \tag{7}$$

The serial bandwidth to read data remotely is

$$BW = ((BW_R)^{-1} + (BW_L)^{-1})^{-1} \tag{8}$$

The private input data is read from the home node for the first job in each pipeline and from the local cluster for subsequent jobs. The time to read the private data, T_P, is the sum of the time to read one private volume from the remote storage server and the time to read the remaining $D-1$ private volumes from the local cluster:

$$T_P = \frac{S_P}{BW} + \frac{S_P \cdot (D-1)}{BW_L} \tag{9}$$

The time to write the private data is the same except that it is the last private volume, not the first, that is written to the home server.

To predict the total compute time, T_{CPU}, we multiply the average compute time, $\bar{R}$, by the number of jobs in each pipeline, D:

$$T_{CPU} = \bar{R} \cdot D \tag{10}$$

Because All does not require any barriers, all jobs can run as soon as their job dependencies are satisfied. Because no jobs must wait to execute, their runtime variability is *not* considered within the predictive model.

The expected number of concurrently executing jobs is

$$C = min\,(W, N_{CPU}) \tag{11}$$

depending on whether sufficient compute nodes are available to run all W pipelines. The number of warm phases, N_{Warm}, is

$$N_{Warm} = \left\lceil \frac{W}{C} \right\rceil - 1 \tag{12}$$

which subtracts the initial cold phase from the total number of phases required to run all W pipelines at a concurrency of C.

Each warm phase consists of the time to fetch the batch data from the local cache, T_{WarmB}, plus the times to read the private input, T_P, write the private output, T_P, and compute, T_{CPU}:

$$T_{Warm} = T_{WarmB} + T_P + T_P + T_{CPU} \tag{13}$$

The last estimate needed to predict the runtime for All is T_{WarmB}, the time to fetch the batch data in the warm phase after it has already been cached in the local cluster. This is computed as the dividend of the the total amount of batch data, S_{TotB}, and the local bandwidth, BW_L:

$$T_{WarmB} = \frac{S_{TotB}}{BW_L} \tag{14}$$

Predicting Runtime for AllBatch. The algorithm for predicting the total runtime for workloads scheduled using AllBatch is similar to that for All. The sole difference lies in computing the expected number of concurrently executing jobs, C. In this case, the number of executing jobs is also limited by the number of input and output private volumes that can be allocated after accounting for all batch data:

$$C = min\left(W, N_{CPU}, \left\lfloor \frac{S - S_{TotB}}{2S_P} \right\rfloor\right) \tag{15}$$

Predicting Runtime for Slice. The model for predicting runtimes for Slice differs from the model for All in two ways. First, the expected number of executing jobs for Slice is limited by the number of output private volumes that can be allocated after allocating all the volumes necessary to hold a single horizontal slice of the workload. Therefore, we define the expected concurrency in the steady state to be

$$C' = \left\lfloor \frac{S - S_B - S_P \cdot W}{S_P} \right\rfloor \tag{16}$$

In order to predict the runtime for Slice, we need to account for periods of execution during which the number of executing jobs can exceed the steady state value. The number of executing jobs may be greater at both the beginning and end of the workflow due to the fact that less storage is needed to allocate a horizontal slice of the workload. Therefore, to estimate the runtime, we must compute the *average* concurrency between these maximum and steady state values. This complexity was ignored for the other strategies because the number of executing jobs is more consistent throughout the traversal of the workload. Because of this effect, the width

	W	D	S_{TotB}	S_{TotP}	$\bar{R}$	$SD(R)$
Batch	350	5	883 GB	420 GB	5000 s	500 s
Private	350	5	150 GB	5250 GB	5000 s	500 s
Mixed	350	5	225 GB	1050 GB	5000 s	500 s

Table 1: Synthetic Workload Parameters.

and the depth of the workload influence the average concurrency. For small widths and depths, this additional concurrency at the beginning and end of the workflow makes a larger contribution.

The exact effect seen is that the first and last n jobs can all execute concurrently, where n is strictly greater than the number of jobs expected to concurrently execute in the steady state. This value of n is found by determining how many jobs can have both their input and output private volumes allocated after allocating a single batch volume:

$$n = \left\lfloor \frac{S - S_B}{2S_P} \right\rfloor \tag{17}$$

Therefore, we estimate the average expected concurrency for Slice to be the weighted average of n and $C^{'}$. The number of jobs to be executed at a concurrency of n is n at the beginning of the workflow plus another n at the end. The remainder of the $(J - 2n)$ jobs will be executed at the expected steady state, $C^{'}$. Therefore, the expected concurrency, C, for Slice is

$$C = min\left(W, N_{CPU}, \frac{2n^2 + (J - 2n) \cdot C^{'}}{J}\right) \tag{18}$$

The second difference in the Slice model results from the possibility of barriers. Because each job may need to wait for its sibling jobs to complete, the mean job time, $\bar{R}$, is insufficient. Instead, we use an incremental value between the mean time and a crude estimate of the longest predicted job time, $\bar{R} + SD(R)$. We weight this adjustment based on the likelihood of barriers in the workflow, which is approximated by determining the maximum number of batch volumes that can be concurrently allocated, N_B. In other words, the amount of time a job may wait for its siblings is inversely proportional to N_B.

For Slice, N_B is the number of batches that fit after allocating a single volume for each pipeline and a second volume for as many additional jobs as can execute:

$$N_B = \left\lfloor \frac{S - S_P \cdot W - C^{'} \cdot S_P}{S_B} \right\rfloor \tag{19}$$

Note that because Slice executes as many pipelines as possible after allocating a single batch, N_B will usually be 1 except for very small batch volume sizes or when storage is plentiful. Finally, we merge this wait time into the total compute time for each pipeline:

$$T_{CPU} = \left(\bar{R} + \frac{SD(R)}{N_B}\right) \cdot D \tag{20}$$

Note that the model does not assume anything about the actual runtime distribution of the workload. Even though the runtimes within our synthetic workloads fit a normal distribution, the model is concerned only with estimating the cost of barriers and does so with a crude estimate for the runtime of the longest job. Again, our models are concerned only with relative, and not absolute, accuracy. We feel this simplification is reasonable in that it achieves the desired behavior by penalizing strategies that allow barriers relative to the likelihood of encountering them.

Predicting Runtime for Minimal. The predictive model for Minimal is similar to that of Slice with one major difference. Effectively, Minimal splits the workload into N_{Cycles} subworkloads and schedules each using Slice at full CPU utilization (*i.e.* each job has both private volumes allocated). The predicted runtime is therefore the sum of the predicted runtimes for each of the subworkloads using Slice:

N_{CPU}	S	F	BW_R	BW_L
50	250 GB	0.0	12 MB/s	1 MB/s

Table 2: Compute Environment Parameters.

$$T = \sum_{i=1}^{N_{Cycles}} T(Slice, SubWorkload_i) \tag{21}$$

Notice that in the base case in which Slice can itself execute at full CPU utilization, the number of subworkloads for Minimal is one and the two strategies are identical.

The number of subworkloads is found by dividing the number of pipelines by the expected concurrency:

$$N_{Cycles} = \left\lceil \frac{W}{C} \right\rceil \tag{22}$$

Finally, the expected concurrency is found by determining how many jobs can have concurrent access to both their input and output private volumes after allocating a single batch volume:

$$C = min\left(W, N_{CPU}, \left\lfloor \frac{S - S_B}{2S_P} \right\rfloor\right) \tag{23}$$

Finally, the predicted runtime for Minimal is the sum of the predicted runtimes for each of the subworkloads using Slice:

$$T = \sum_{i=1}^{N_{Cycles}} T(Slice, SubWorkload_i) \tag{24}$$

Predicting the Effect of Failure. The failure rate, F, of the compute environment is *not* considered within any of the individual models. Rather, it is considered as an external effect and is modelled *almost* identically across each of the models. Specifically we take the predicted runtime of the workload and multiply it by the failure rate to determine the number of failed jobs. We then estimate the runtime of these remaining jobs. We repeat until no jobs are expected to fail.

The failure model is applied differently in regard to batch data. When we estimate the runtime for the failed jobs for those strategies which retain batch volumes (All and AllBatch), we adjust the model such that it does not run any cold cycles but rather assumes that all batch data is already cached. Conversely, for Slice and Minimal, which proactively remove batch, the failure model correctly considers that all batch data must be refetched.

4. EVALUATION

To evaluate our predictive models, we define three representative workloads and a representative compute cluster, and run multiple experiments in which we vary different characteristics of the workloads and the environment. Using simulation, we compare the modelled predictions to the actual observed measurements.

Simulation Environment. Our discrete event simulator consists of three parts: the base compute platform, a distributed file system, and a scheduler which dictates job and data allocations. The base compute platform is a simulated set of interconnected computers with disks, buffer caches, and networks. These machines form a distributed file system by binding into a cooperative cache and exporting storage allocation mechanisms. Finally, a data-aware batch scheduler gathers both workload and environmental information and creates a data-driven plan for workload execution. A thorough description and verification of the simulator is available in [4].

Having previously eliminated All and AllPrivate as uninteresting strategies in Section 2, we now consider only AllBatch, Slice and Minimal. To examine the differing performance profiles of these strategies and to evaluate the accuracy of our predictive models, we schedule these strategies for each of three representative workloads across the range of workload and environmental characteristics.

The three synthetic workloads that we use are based on our previous batch workload profiling study [20] and are constructed such that one is *batch-intensive*, one is *private-intensive* and the third is *mixed*, being neither batch- nor private-dominant but having middling values for each. The precise values used in each of these workloads are shown in Table 1. The job runtimes are drawn from a normal distribution with mean $\bar{R}$ and standard deviation $SD(R)$. For each workload, we set the compute time such that a single pipeline, if executed locally at the home storage server, would perform I/O for approximately 50% of its total execution time.

We choose these workloads such that, for the batch-intensive workload, only the Slice and Minimal strategies are possible, and for the private-intensive workload, only the AllBatch and Minimal strategies are possible. For the mixed workload, the AllBatch, Slice, and Minimal strategies are possible but All and AllPrivate are not. Finally, the values used to define the environment are listed in Table 2. Although the absolute values are small relative to many current systems, what is more important is the ratio which does reflect the continuing trend of application data exceeding the capacity of the remote execution environment.

Predictive Accuracy. Having crafted a set of representative workloads and a compute environment in which their execution is data constrained, we now examine the relative performance of the different strategies as we vary each of the eleven variables defining the workloads and the environment. We do not however individually examine each of these variables; some are evaluated as their ratio to another, such that we now examine eight experiments derived from our eleven variables. However, due to space constraints, we will not show results for each of these experiments; the full set of experiments is available in [4].

Interpreting the Results. For many of these experiments, we show a set of graphs comparing the observed simulated throughput and the throughput estimated by our predictive models. Looking ahead to Figure 4, which will be discussed in more depth below, we see the graphs on the left show the effect on the batch workload, the middle on the private and the right most on the mixed workload. Within each set of graphs, the top row shows the measured throughput for the workloads as achieved in our simulated system. The middle row shows the throughput as estimated by our predictive models. Finally, in the bottom row, we quantify the predictive accuracy of our model.

The measurements in these graphs sometimes abruptly end; this can be seen clearly for the throughput of Slice for the mixed workload (the middle graph on the right). Notice that the throughput first drops and then entirely disappears. The reason for this is that at a certain width, Slice is no longer possible.

The predictive accuracy of our models is seen qualitatively in the graphs along the bottom row in which the observed runtime of the predicted best strategy is normalized against the runtime of the actual strategy observed to achieve the highest throughput via simulation. Points at which the model mispredicts the highest throughput strategy appear as a positive value in these graphs. The height of these points is the percent of additional (unnecessary) runtime that would be incurred by a scheduler using this misprediction as compared to a perfect scheduler that always correctly identifies the best strategy. In cases in which the model correctly identifies the best strategy, this value is zero. Also shown for comparison is the normalized value of the *worst* possible strategy.

Note that we are *not* comparing the predicted throughput of the model to the observed throughput of the simulator; rather we use the relative values from the model to select which strategy is expected to achieve the highest throughput. We then compare that strategy's simulated runtime to the best simulated runtime.

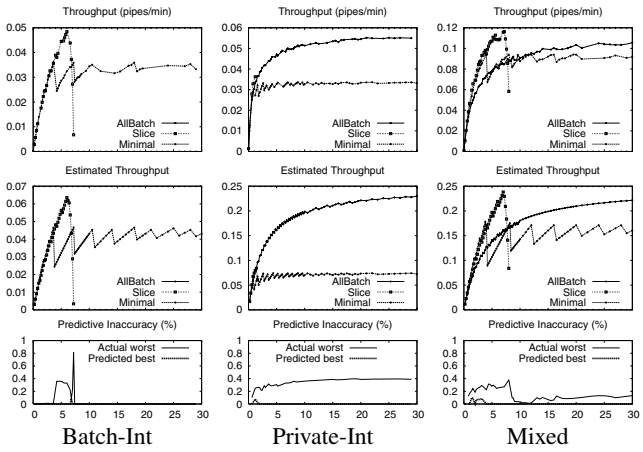

Figure 4: Sensitivity to Workload Width.

However, it is not our intention that the models be perfectly accurate. Indeed in the interests of simplification, we have ignored many system aspects in the models such as network contention, the memory subsystems of the compute nodes and the home storage server as well as the latencies for the disks and networks. These simplifications cause the model to overestimate the throughput of the workload. Our aim is not to make absolute predictions but is rather to select from the possible scheduling strategies. In other words, we are not interested in the ability of the models to accurately predict runtimes but rather in their ability to accurately predict the relative runtimes of the different strategies.

As will be seen, the models do not *always* predict the best strategy. However, as long as they only makes mistakes when the relative throughput of the expected best is close to the actual best, then the effect of this inaccuracy is minimized. Notice further that although the models do not make perfect estimates as to the predicted throughput of the strategies, it does provide perfect information as to which strategies are and are not possible. Thus, for every case in which only one strategy is possible, the model will correctly identify it. Only when multiple strategies are possible does there exist any possibility of misprediction.

For a dynamically changing environment as is typical in batch computing, these predictive models do not need to be optimal. In fact searching for optimal information in a batch system is often quixotic as the information can become stale and therefore suboptimal very quickly. For this reason, it becomes much more important not necessarily to select the best strategy but to avoid very bad ones. *As the evaluation shows, our models do so in all cases.*

Sensitivity to Workload Width. Shown in Figure 4 is the effect that increasing the width of the workload has on the performance of the three studied strategies. For these graphs, the x-axis is not the absolute width of the workload but is rather its ratio to the number of cluster compute nodes. There are several things to notice in these graphs. First, we'll discuss why each of the strategies behaves as it does for each of these workloads and then we'll discuss the ability of our model to correctly predict these behaviors.

Notice the throughput crossover points for both the batch and the mixed workloads. These crossovers occur because for "thin" workloads, Slice is able to concurrently execute on all nodes. For the batch-intensive workload, AllBatch is not possible and Minimal underperforms Slice due to the redundant use of the WAN, as shown in the right-most graph. However, as the width of the workload increases, there is a time in which Slice remains possible but only at a drastically reduced concurrency. At this point, the throughput of Minimal surpasses that of Slice. A similar effect is

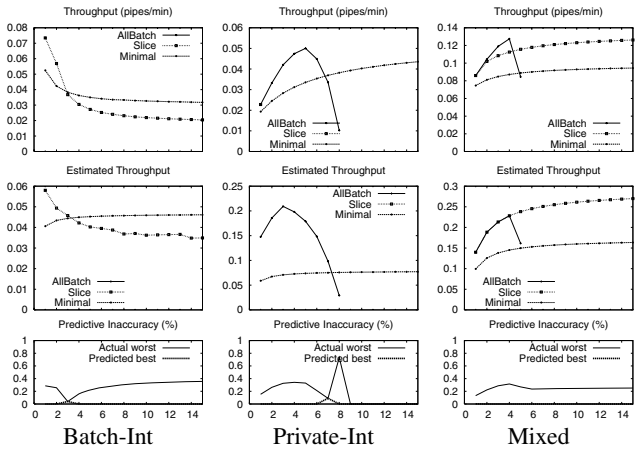

Figure 5: Sensitivity to Workload Depth.

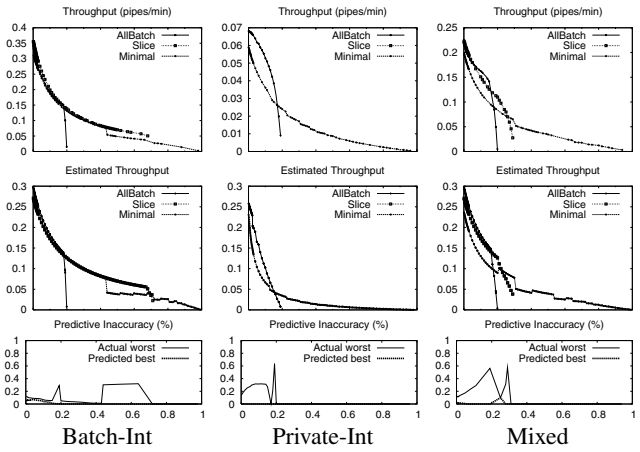

Figure 6: Sensitivity to Batch Volume Size.

shown for the mixed workload yet in this case AllBatch is possible albeit at a lower concurrency. Notice further that, as expected from the discussion in Section 2, both AllBatch and Minimal are robust to increasing workload widths while Slice is significantly more sensitive. For the private-intensive workload, no crossover effect is revealed because Slice is not possible and the other strategies are relatively constant across increasing workload widths.

The sawtooth pattern exhibited by Minimal is due to the "tail" effect. For workloads in which the expected number of concurrent jobs is not an even factor of the total workload width, the last group of jobs to execute will be smaller than the previous groups. In the batch workload for example, the maximum concurrency is approximately three times that of the number of compute nodes such that it achieves peaks in throughput at multiples of three along the x-axis.

As for the predictive accuracy, note that the models are not absolutely accurate, especially in regards to their predicted throughput values. Although close for the batch workload, they are off by a factor of five for the private workload and a factor of two for the mixed. However, in regards to the *relative* accuracy, our modeled estimates mirror closely the simulated results. In particular, to evaluate the predictive ability of the models to identify the best strategy, we examine the crossover points in these graphs. Notice that there is a perfect one-to-one correlation in the crossover points between the simulated and the modeled results. Further, each pair of correlated crossover points occurs at relatively the same position on the x-axis; in other words, our model correctly predicts both that a crossover will occur as well as where it will occur. The models consistently avoid making bad predictions. Although the worst possible prediction is sometimes eighty percent worse than the best, the models never make mispredictions greater than ten percent.

Sensitivity to Workload Depth. Similar to varying the workload width is varying the depth as is shown in Figure 5. As expected from the discussion in Section 2, the performance of AllBatch is highly sensitive to the depth. As the depth increases, the total amount of batch data also increases, thereby allowing less storage in which fewer pipelines can execute.

Some less obvious results are also seen in these experiments. Notice in the batch workload that there exists a crossover point between the Slice and Minimal strategies. This cross-over point correlates with measurements of the CPU utilization which are not shown here due to space constraints. For small depths, Slice can achieve higher levels of concurrency than at greater depths. The reason for this discrepancy is that the level of concurrency for Slice is different at the beginning and end of the workload's execution than it is in the middle as was discussed in Section 3.

Notice again that our predictive accuracy is very high; we never make a misprediction that is more than five percent from the identified best. Although our absolute predictions are inaccurate, we correctly identify every crossover and do so for each close to their precise location on the x-axis.

Sensitivity to Volume Size. We also examine the effect of changing the batch volume size as shown in Figure 6. Here the x-axis is the ratio of the batch volume size to the total amount of storage. Because we vary here one of the aspects that distinguishes the three workloads, we effectively homogenize them such that the results as compared across workloads are similar.

As expected, increasing the size of the batch volumes has the largest effect on AllBatch. The other strategies are effected but to a lesser degree as they need allocate only a single batch volume at a time. Notice however that after AllBatch is no longer possible for large values of batch volume sizes, there still exist interesting relative performances between the Slice and Minimal. For the batch workload, Slice is possible long after AllBatch is not, and while it remains possible it outperforms Minimal due to the heavy cost in the batch-intensive workload of transferring redundant data over the WAN. Yet ultimately, Minimal is able to continue after Slice is not. The effect is similar in the mixed workload but Slice becomes impossible more quickly here due to the larger sizes of private volumes in this workload as compared to the batch workload.

Here the predictions of the model are particularly striking in their accuracy; the slopes and crossover points are visually very similar. Qualitatively observing the measured values in the bottom graphs, we see again that the model never predicts worse than ten percent and successfully avoids making bad predictions.

Varying the private volume size yields results very similar to those produced by varying the batch volume size as both homogenize the workloads. As such and due to space constraints, these results are not shown here. However, the models are even more accurate here as they do not make a single misprediction.

Sensitivity to Computation Time. The amount of time spent within each job doing computation and not I/O is another workload characteristic that affects the relative performance of the different strategies. Although this experiment is not shown due to space constraints, it does reveal an interesting crossover point between AllBatch and Slice which is predicted correctly by our model. AllBatch is constrained here by having concurrecy less than the number of CPUs such that it makes progress but underutilizes the CPUs. Conversely, Minimal has a higher concurrency but must refetch the batch data. As the compute time increases relative to the time it takes to refetch the batch data, the relative refetch penalty incurred

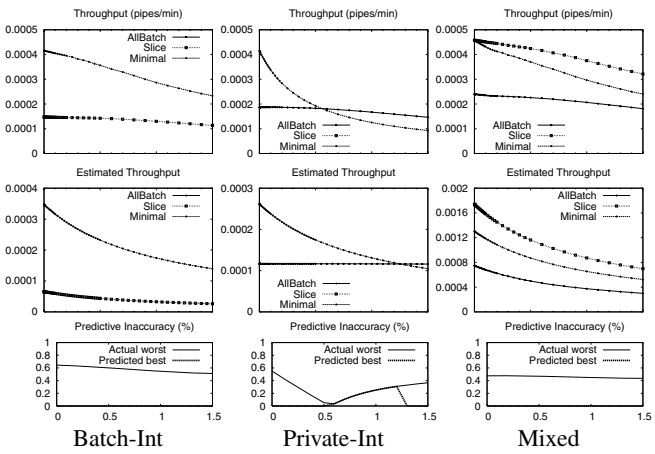

Figure 7: Sensitivity to Runtime Variability.

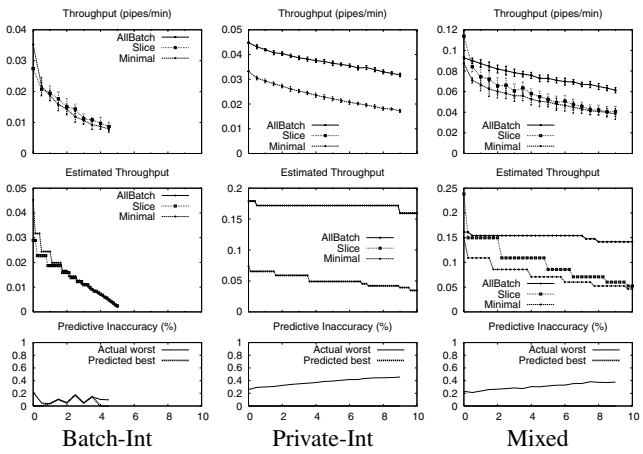

Figure 8: Sensitivity to Failure Rate.

by Minimal is reduced while the underutilization penalty incurred by AllBatch remains constant. Thus, for increasing values of compute time, we observe that the performance of Minimal improves relative to that of AllBatch.

Sensitivity to Runtime Variability. Similar to the effect of changing the compute time is changing the variability across job times. Although this experiment is not shown due to space constraints, our models make entirely correct predictions for the relative throughputs of the different strategies.

However these results are somewhat uninspiring due to the small influence that compute time has on the overall runtime of each pipeline. Therefore, we repeat this experiment with modified workloads such that compute time is a much larger relative value. As mentioned earlier, the compute time is set such that it comprises approximately half of the total execution time for a pipeline run locally at the home storage server. For these modified experiments, we increase it such that its relative proportion increases to approximately 99% (*i.e.* five million seconds).

With such a large value for compute time we can more readily see its influence on the different strategies as shown in Figure 7. As expected, a large variance disproportionately penalizes strategies which impose barriers. This is seen as the relative performance of AllBatch which imposes no barriers is mostly constant while the performance of Minimal and Slice which do impose barriers degrades much more rapidly.

Here again, the accuracy of our model is not perfect. Although it correctly identifies the crossover point, it misses its precise location on the x-axis. Although this miss results in approximately a 30% difference in throughput, notice the model does successfully avoid making mispredictions with inaccuracy exceeding 50% in both the batch-intensive and private-intensive workloads.

Sensitivity to Network Bandwidths. The relative performance of the wide-area and local networks can also influence the achievable throughputs of the various scheduling strategies, but these results are not shown due to space constraints. Intuitively as we consider each strategy's constraints, it is clear that the refetch penalty incurred by Minimal is most sensitive to this. This intuition is substantiated in the experiment as the performance of Minimal relative to the other strategies improves more dramatically as the bandwidth to the remote storage server grows relative to the local bandwidth.

This intuition is modelled correctly as well. Although a poor strategy selection could result in throughput differences as high as 60%, our model consistently remains within three percent.

Sensitivity to Failure. Finally, the effect of failure is examined in Figure 8 in which we increase the failure rate along the x-axis. Due to the randomness by which we induce simulated failures these experiments are not deterministic. We therefore run each point for thirty iterations and show the means and standard deviations.

Each failure event "resets" a compute node in the system such that any running job on that compute node is evicted and all data contained on that node's disk and memory is erased. As discussed in Section 3, failures disproportionately effect Slice and Minimal because failed pipelines must refetch all batch data whereas in AllBatch the batch data is not removed and failed pipelines can be rerun using the already locally cached batch data. This behavior is seen most clearly in the mixed workload in which Slice achieves the highest throughput for low rates of failure but then quickly drops below AllBatch as the rate of failure increases.

Although the predictive model does suffer some small inaccuracy in the batch-intensive workload, it does so only within the bounded variability. The predicted slopes are accurate and the major crossover point in the mixed workload is predicted accurately.

Discussion. Our models do not provide absolute accuracy but do provide accurate relative predictions. Across our experiments, we have seen a wide range of runtimes for the different strategies and many instances in which mispredictions could result in over 80% additional runtime compared to the identified best strategy yet our models consistently predict strategies with runtimes within 5% of the identified best and never predict runtimes more than 30% greater.

One remaining question concerns the validity of our simplifying batch-pipeline workloads into a canonical structure. We assert that an understanding of canonical workload scheduling is beneficial for scheduling non-canonical workloads as well. For example, imagine a workload with "small" batch volumes at the ends and a "large" batch volume in the middle. By definition, this is not a canonical workload but by applying our scheduling knowledge we can avoid lost work that might arise otherwise. An oblivious scheduler which did not use our planning techniques but did observe capacity constraints might allow many pipelines to initially execute while the batch data is small. When the traversal would reach the large volume however, not all of the allocated pipelines would be able to remain allocated and their forward progress would necessarily be lost. This lost progress could be avoided however by making the workload appear canonical by increasing the size of all batch and private volumes to their maximum respective sizes and then using our predictive models to correctly identify the allocation limits for the workload.

5. RELATED WORK

Backfilling techniques in parallel scheduling which use small programs to fill holes left when a program does not use all available resources [14] should be relevant for batch schedulers. One difference however is what constitutes a small program; for parallel scheduling, a small program is one which uses a small number of processors whereas for data-driven batch scheduling, a small program is one which uses a small amount of storage.

Gang scheduling with memory considerations [2] is a policy for capacity-aware scheduling of parallel programs where memory is scarce. This is similar except that our scarce resource is disk storage. Another difference is that gang scheduling schedules across multiple parallel programs while batch schedulers schedule multiple jobs within a single batch-pipeline workload. Additionally, batch schedulers have the luxury of refetching batch data when remote storage is scarce; gang scheduling considers no such secondary backing store for memory (swapping to disk is particularly severe in parallel program as it interferes with the synchronization among the job's threads).

Another approach to coordinate allocations of storage and CPUs in parallel scheduling is shown in [18] in which multiple reservations are attempted and then are used only if all are successful. In fact, this is the approach we propose for batch schedulers: to only execute jobs after both claiming necessary CPUs and storage.

Within the grid community, there is an increasing awareness of the growth of datasets [1, 8, 10, 11, 21] and a correspondingly increasing interest in the coordinated scheduling of data and computation. Stork [12] creates mechanisms for the controlled transfer of datasets across wide-area networks and, like our system, is explicitly designed for moving an awareness of batch inputs and endpoint outputs into the scheduling framework. However, Stork is a procedural approach where users add explicit data-movement jobs into their workloads whereas we propose a declarative approach in which users provide the necessary information and the scheduler makes data-movement decisions.

Many approaches have been proposed for caching and locating batch datasets [3, 6, 13, 15, 16, 17]. These approaches complement our work here as they pertain to the more persistent question of what happens to batch data *following* the completion of the workload, whereas here we have addressed the question of how to access the batch data *during* the workload's execution.

6. CONCLUSIONS

We define four scheduling allocations for data-intensive batch workloads and evaluate them across a range of workload and environments. We have further provided analytical predictive models with which a data-aware batch scheduler can choose the appropriate data allocation with a goal towards minimizing the total time to completion. Finally, using simulation, we have demonstrated the accuracy of our predictive models and have shown how their use can lead to throughput improvements as high as 80%.

Allocating CPUs is easy, but may cause an overallocation of storage; allocating data is easy, but may cause an underutilization of CPU; allocating both while adversely affecting neither is the challenge for modern batch schedulers.

7. REFERENCES

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *SOSP '95*, pages 109–26, Copper Mountain, CO, December 1995.

[2] A. Batat. Gang scheduling with memory considerations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 109, Washington, DC, USA, 2000. IEEE Computer Society.

[3] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger, and F. Zini. Evaluation of an Economy-Based File Replication Strategy for a Data Grid. In *In Proceedings of 3nd IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid'2003)*, Tokyo, Japan, May 2003.

[4] J. Bent. *Data-Driven Batch Scheduling*. PhD thesis, University of Wisconsin, Madison, May 2005.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit Control in a Batch-Aware Distributed File System. In *NSDI '04*, pages 365–378, San Francisco, CA, March 2004.

[6] A. L. Chervenak, E. Deelman, I. Foster, A. Iamnitchi, C. Kesselman, W. Hoschek, P. Kunst, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A framework for constructing scalable replica location services. In *SC '02*, Baltimore, MD, November 2002.

[7] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[8] I. Foster and P. Avery. Petascale Virtual Data Grids for Data Intensive Science. GriPhyn White Paper, 2001.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[10] J. Gray and A. S. Szalay. Scientific Data Federation: The World-Wide Telescope. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, pages 95–108. 2003.

[11] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[12] T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, 2004.

[13] H. Lamehamedi, Z. Shentu, B. K. Szymanski, and E. Deelman. Simulation of Dynamic Data Replication Strategies in Data Grids. In *International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[14] D. A. Lifka. The anl/ibm sp scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[15] S.-M. Park, J.-H. Kim, Y.-B. Ko, , and W.-S. Yoon. Dynamic Data Grid Replication Strategy based on Internet Hierarchy. In *Second International Workshop on Grid and Cooperative Computing (GCC'2003)*, Shanghai, China, December 2003.

[16] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing (HPDC)*, Edinburgh, Scotland, July 2002.

[17] A. Romosan, D. Rotem, A. Shoshani, and D. Wright. Co-Scheduling of Computation and Data on Computer Clusters. In *Scientific and Statistical Database Management Conference (SSDBM 2005)*, Santa Barbara, California, June 2005.

[18] Q. Snell, M. Clement, D. Jackson, , and C. Gregory. The Performance Impact of Advance Reservation Meta-Scheduling. *Job Scheduling Strategies for Parallel Processing*, 1911, June 2000.

[19] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[20] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and Batch Sharing in Grid Workloads. In *HPDC 12*, pages 152–161, Seattle, WA, June 2003.

[21] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, N. Tammineedi, and S. L. Scott. FreeLoader: Scavenging Desktop Storage Resources for Scientific Data. Technical report, Computer Science and Mathematics, Oak Ridge National Laboratory, 2005.