STEAMEngine: Driving MapReduce Provisioning in the Cloud*

Michael Cardosa, Piyush Narang, Abhishek Chandra
University of Minnesota
{cardosa,piyang,chandra}@cs.umn.edu

Himabindu Pucha, Aameek Singh
IBM Research - Almaden
Email: {hpucha, Aameek.Singh}@us.ibm.com

***Abstract*—**

MapReduce has gained in popularity as a distributed data analysis paradigm, particularly in the cloud, where MapReduce jobs are run on virtual clusters. The provisioning of MapReduce jobs in the cloud is an important problem for optimizing several user as well as provider-side metrics, such as runtime, cost, throughput, energy, and load. In this paper, we present an intelligent provisioning framework called STEAMEngine that consists of provisioning algorithms to optimize these metrics through a set of common building blocks. These building blocks enable spatio-temporal tradeoffs unique to MapReduce provisioning: along with their resource requirements (spatial component), a MapReduce job runtime (temporal component) is a critical element for any provisioning algorithm. We also describe two novel provisioning algorithms—a user-driven performance optimization and a provider-driven energy optimization—that leverage these building blocks. Our experimental results based on an Amazon EC2 cluster and a local Xen/Hadoop cluster show the benefits of STEAMEngine through improvements in performance and energy via the use of these algorithms and building blocks.

I. Introduction

The growing data deluge has inspired significant interest recently in performing large-scale data analytics, for tasks such as web indexing, document clustering, machine learning, data mining, and log file analysis. MapReduce [1] and its open-source implementation, Hadoop [2], are emerging as a popular paradigm for such data analytics, given their ability to scale-out to large clusters of machines. This growing interest from users of MapReduce is suitably matched by enterprises/service providers hosting massive scale infrastructure for MapReduce. Several enterprises including Facebook, Yahoo, and Microsoft run their own shared infrastructure, akin to a private cloud, where different MapReduce jobs within the enterprise are simultaneously executed. Similarly, MapReduce offered as a service in the public cloud (e.g., Amazon Elastic MapReduce [3]) shows great promise. Often in such cloud environments, server virtualization is used for providing multi-tenancy and isolation. In this paper, we consider such a virtualized cloud platform, wherein each node of a MapReduce cluster is associated with a virtual machine (VM) which is then placed on a virtualized physical machine in the data center. VMs from different customers and/or different MapReduce applications share the set of physical machines in the data center.

Common to both the consumers and providers of such a MapReduce service is the need to optimize their deployments. For instance, the end-user of a MapReduce service typically cares about minimizing cost for a given MapReduce job while satisfying their performance requirements. Similarly, from the cloud operator's perspective, the desired objective is to impact the bottom-line via optimizing system-wide goals such as maximizing system throughput, minimizing energy consumption and load balancing. Such optimizations on different metrics for end-user or provider require different algorithms, however, at their core they all leverage some common resource provisioning constructs for achieving their objectives: for a consumer, this involves choosing the optimal number of VMs to run its job; for a provider, this involves placing VMs from different jobs on physical machines optimally.

A key challenge, however, is optimizing these metrics in a dynamic environment in an intelligent and automated fashion. As new MapReduce jobs arrive and old ones finish or the performance of an executing job varies with time (e.g., due to stragglers and failures [4]) or as the workload and system utilization change, the deployment needs to be optimized in response to those changes. Our work explores this key issue of dynamic and adaptive resource provisioning for MapReduce. As discussed later in the section, MapReduce offers unique opportunities which when leveraged can lead to significant optimization opportunities.

A. Research Contributions

We propose the STEAMEngine provisioning framework which contains a set of tools that optimize MapReduce deployments in the cloud. The framework is a two-tier architecture. The first tier is an extensible library of provisioning algorithms that optimize for various user and provider side optimizations – e.g. performance optimization for a user or an energy optimization for a cloud provider. These optimizations are built into the underlying resource management infrastructure and surfaced to users as additional features of the service. The second tier is an extensible set of common *building blocks* – components that provide information about the job and the cloud environment which serves as input to the provisioning algorithms, and also actuate the provisioning decisions taken by the algorithms. This layered architecture is based on the insight that many of these provisioning algorithms require similar information about the job and/or the cloud infrastructure (e.g., the estimated time of completion of the job or load on a physical server), and also employ similar provisioning

*This work was supported by NSF Grant CNS-0643505

978-1-4577-1950-9/11/$26.00 ©2011 IEEE

mechanisms (e.g., scaling the size of the virtual cluster). Based on these readily available building blocks, optimizations can be quickly developed to provision and/or continually optimize the MapReduce deployment in the cloud.

Concretely, in this paper, we showcase two provisioning algorithms built over two building blocks. We describe (i) an automated end-user-side provisioning algorithm that dynamically *optimizes performance* of a job—meeting a job runtime deadline while minimizing cost, and (ii) a provider-side provisioning algorithm that *minimizes system energy consumption* in the presence of MapReduce job arrivals and departures. In the absence of STEAMEngine, the performance optimization will need to be performed manually by the user, while the energy minimization algorithm will devolve into performing inefficient VM placement using spatial fitting only.

These algorithms leverage two building blocks (i) *Job Profiling* that exploits the *predictable and equitable behavior* of a MapReduce job to predict its completion time based on its input data and cluster size, and (ii) *Cluster Scaling* that exploits the *ease of scaling* in a MapReduce job by dynamically changing the size of the cluster running the MapReduce job, to alter its runtime.

We have evaluated STEAMEngine on both Amazon EC2 [5], as well as a local Xen cluster. Our results show that our end-user performance optimizing algorithm, running on Amazon EC2, enabled MapReduce jobs to meet their given deadlines even with inaccurate initial information. Further, our energy optimization algorithm enabled energy savings of up to 14% in our local testbed (Section IV).

B. Unique Spatio-Temporal Tradeoffs in MapReduce

STEAMEngine is built specifically for MapReduce. While resource provisioning for Internet applications in a public or a private cloud setting is well studied [6, 7, 8], we argue that applying that work directly to MapReduce provisioning misses out on an important opportunity. As opposed to "always-on" Internet applications, MapReduce jobs are inherently batch jobs with a bounded runtime. Thus, MapReduce VMs, in addition to being characterized by their spatial properties (e.g., CPU, memory), have a temporal component as well. This temporal nature of MapReduce VMs leads to unique *spatio-temporal tradeoffs* when performing resource provisioning, as discussed below.

For instance, most end-users provisioning these "always-on" Internet applications allocate sufficient resources to meet their performance requirements and need not incorporate the time duration these resources will be online into their cost calculations. However, for a MapReduce job, the cost incurred by a job is dependent on both the time required for job completion, and the resources allocated for the job. Further, the time required for job completion is inversely related to the allocated resources. Hence, resource allocation for optimizing end-user cost for a MapReduce job must account for both the spatial and temporal characteristics of a job.

Similarly, from a provider's vantage point, while there has been significant work in workload placement for traditional applications [9, 10, 11, 12] via efficient spatial placement of VMs, leveraging those concepts for placing MapReduce jobs is not sufficient. Along with their resource requirements (spatial component), MapReduce job runtime (temporal component) is a critical element for any provider-side resource placement, as illustrated by the following example.

Example 1: Take an example placement for an energy conservation objective, where servers can be shutdown or put into a sleep state to conserve energy if they are idle, so that the goal is to minimize the total uptime of the servers in the system. Consider a cloud instance (Figure 1) running three MapReduce jobs J_1, J_2, J_3, each with two VMs and utilizing 40%, 30% and 20% of physical server resources respectively. Further assume that the runtime of the jobs is 10, 90 and 100 mins respectively. Most traditional placement algorithms only consider the resource utilization (the *spatial* component). Such algorithms may use two physical servers to achieve spatially efficient packing, as shown in Figure 1(a). This placement will result in a total uptime of the servers being 200min. Incorporating the runtime information (*temporal* component) in the placement algorithm will allow choosing a better placement (Figure 1(b))—two tasks of J_1 on one server and the rest on the second server, thus *time-balancing* each server better. This placement will result in a total uptime of 110min, thus using 45% less energy than the first placement.

To the best of our knowledge, our work is the first to exploit this spatio-temporal tradeoff for MapReduce provisioning, both within and across MapReduce jobs.

II. STEAMEngine: Architecture and Building Blocks

Running MapReduce in a virtualized cloud environment requires the cloud service provider to provision VMs that form the MapReduce cluster for each job. In this model, each VM serves as a *node* in the MapReduce cluster. The VM type (CPU, memory, storage) and the number of VMs is chosen by the user submitting the job[1]. These should be optimally picked based on the desired performance and cost objectives for the job and are the only control points for the user to optimize their job. However, currently tools that inform the user for making such decisions intelligently are lacking and users rely on ad hoc decisions based on prior experience or trial and error.

Once the number of VMs is picked, the cloud provider retains complete freedom in placing these VMs among its physical server and storage resources. This placement of VMs is a key lever that can control the optimization of the cloud environment[2] and the choice of the algorithm is dictated by the specific objective chosen by the cloud operator, e.g., maximizing throughput, balancing load or minimizing energy

[1]The VM type is typically selected from a fixed set of available VM types (e.g., the VM instances in Amazon EC2)

[2]To optimize individual jobs, another lever is the choice of MapReduce configuration parameters, e.g. number of concurrent mappers/reducers per node. In this paper, we focus exclusively on the VM placement lever and aim to address integration of the two in future work.

(a) Spatially-efficient VM placement (b) Space-time-efficient VM placement

Fig. 1. Spatio-Temporal tradeoff in MapReduce provisioning.

consumption. In order to optimize MapReduce provisioning, these placement algorithms needs to account for the spatio-temporal tradeoffs described earlier.

We argue that while the optimization logic needs to be tailored specifically to the needs of the objective, *all* provisioning algorithms will benefit from leveraging common opportunities provided by MapReduce. In this section we present some common *building blocks* motivated by the different opportunities for the provisioning algorithms to build on. These building blocks are intended to be used by the cloud provider as well as users, when appropriate, in order to make smarter provisioning decisions based on their specific objectives.

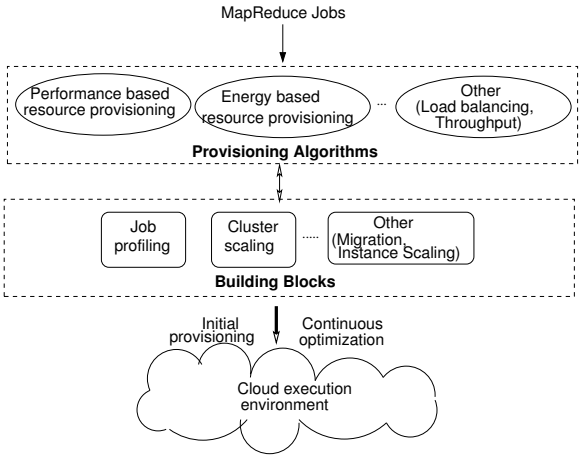

Fig. 2. STEAMEngine Framework

Figure 2 shows the proposed STEAMEngine framework. End-users submit MapReduce jobs by specifying the job characteristics (e.g., data size and VM type). The job is placed in the cloud using an appropriate provisioning algorithm based on the chosen objective, e.g. a user-driven performance optimization (Section-III-A), or cloud provider-driven energy optimization (Section-III-B). To exploit the opportunities provided by MapReduce, these algorithms require some common information and mechanisms which are provided by the building blocks, that are described next.

Note that the provisioning algorithm may be executed both at the time of initial provisioning of the job when it is first submitted as well as in a continuous fashion while the job is executing in order to optimally adapt to the changing characteristics of the cloud.

A. Job Profiling

The Job profiling building block is designed to expose the spatio-temporal tradeoffs offered by MapReduce jobs, by estimating job runtime as a function of the resources allocated to the job. This runtime information of a job enables the cloud operator to optimize performance, cost or energy of the execution environment at the time of initial provisioning, while also providing information to reprovision to achieve continuous optimization. Additionally, this information can be leveraged by the user to pick or dynamically modify the number of VMs in the MapReduce cluster. We consider two complementary profiling techniques to provide such estimates: (i) an *online profiling* technique which captures the progress of an executing MapReduce job to continuously estimate and update its expected runtime, and (ii) a *bootstrap profiling* technique that relies on historical data from prior runs as well as execution of sample jobs to estimate the runtime of a MapReduce job before it starts executing.

1) Online profiling: In order to estimate a job's runtime during its execution, we use an online profiling technique. This technique provides us with fine-grained information about the progress of a running job, and enables updating the estimates of its runtime on the fly. These estimates may need to be updated either because the performance of a job may vary due to failures or stragglers, or we may not have sufficient historical information to make accurate estimates initially. Further, the model used for estimation may be inaccurate, e.g., if it does not account for I/O or network bottlenecks in the system.

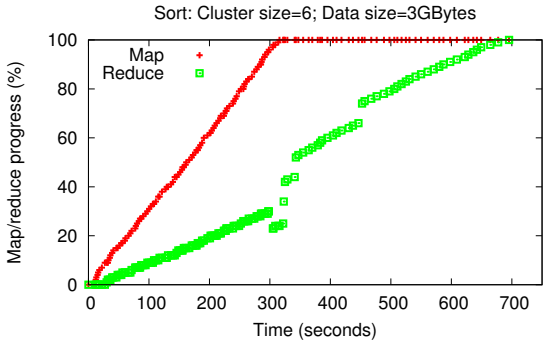

Fig. 3. Online progress of a MapReduce sort job.

As part of this profiling technique, we use the current

progress of the job itself to predict its runtime. To illustrate, Figure 3 shows the online progress of the Map and Reduce tasks for sort benchmark. The details of the experimental setup are provided in Section IV. Our results suggest that the progress of the Map phase is linear. On the other hand, the residual Reduce phase, once the Map tasks are done, shows a bursty, though piecewise linear progress. Thus, by measuring the current progress of a job, we can extrapolate its total runtime.

In our implementation, at any point of the job execution, we extrapolate the Map progress linearly to obtain the Map completion time estimate. For the residual Reduce time, we start by using the bootstrap estimate (described below) and once the Map phase is over, it is also linearly extrapolated similar to the Map phase. Note that any failures or stragglers in MapReduce jobs will adversely impact these estimates. For such scenarios, as part of our future work we intend to employ techniques similar to ones proposed in [13] and adjust the slope using exponentially weighted moving average.

2) Bootstrap profiling: While the runtime estimates obtained via online profiling allow us to track a job's progress and update the estimates at runtime, we also employ a bootstrap profiling technique to provide an initial estimate for a job's runtime even before it starts executing. Such an initial estimate can help in provisioning appropriate resources for a job a priori, and also minimize the need for reprovisioning at runtime. As part of this technique, we model the runtime of a job as a function of datasize and number of (virtual) nodes in its cluster. This model is developed using past observations of runtimes of a similar job—in most environments, multiple instances of the same MapReduce application, e.g., pagerank, are executed repeatedly, so such observations are easily available—or using extrapolation of running the job on a much smaller data set. Hence this approach is suitable at job bootstrap time (before the submitted job begins execution).

Conceptually, MapReduce jobs are data-dependent and inherently parallel, and hence, the runtime of a MapReduce job is dependent on two important factors: (i) the size of the input data, and (ii) the amount of parallelism, corresponding to the size of the (virtual) cluster available to the job for its execution. This suggests that we can model the job runtime T as a function of these two quantities:

$$T = f(D, n),$$

where D is the input data size, and n is the number of nodes (VMs) in the cluster. The function f is likely to be heavily dependent on the job characteristics – whether it is CPU/memory/disk-intensive, and whether it is Map/Reduce-heavy, etc. One potential approach to estimate job run time is to capture this function f for a given MapReduce job class[3].

To show that obtaining such a model is feasible for realistic MapReduce applications, we ran three MapReduce benchmarks—Pi, Wordcount, and Sort—with varying input data sizes and cluster sizes. These benchmarks were chosen as representative of different classes of MapReduce jobs (e.g., Pi is compute-intensive, while wordcount and sort are memory- and I/O-intensive). The details of the experimental setup are provided in Section IV. Due to space constraints, we only present the results for sort as the conclusions are similar for the other benchmarks.

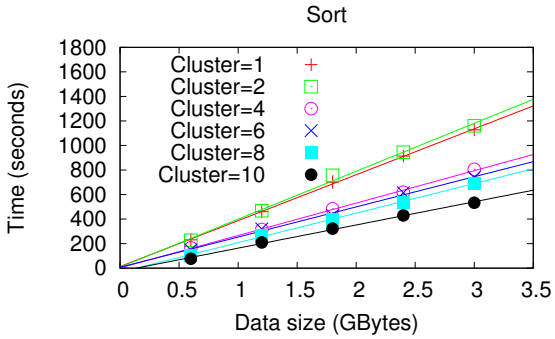

Fig. 4. Data size dependence

Figure 4 shows the runtime of the benchmarks as a function of data size (the lines in the figure are fitted to the data points). As shown in the figure, for a given cluster size, the runtime increases linearly with the input data size. This result implies that if we have prior observations about a job, and we get a new job instance with a different data size, we can estimate its runtime with a linear extrapolation. In fact, this property leads to another profiling optimization. In the absence of historical observations, we can run the job on a subset of the data in a staging area and extrapolate the results to the actual input data size to get an estimate of the expected runtime.

Our bootstrap profiling algorithm works as follows. For each job class, we keep a database of historic observations of job runtimes (separate for Map and residual Reduce phases) along with the corresponding input data sizes and cluster sizes. Upon arrival of a new job, if the profiling data includes an exact match for the cluster size and the data size specifications of the new job, the runtime is simply calculated using the value(s) stored in the database. In the absence of an exact match, we extrapolate the runtime estimate for the given datasize from the stored values of datasize and cluster size values. In the case when no relevant values are available in the database, short runs are used to obtain values for small data and cluster size combinations and then extrapolated from there.

B. Cluster scaling

Our cluster scaling building block is inspired by the elasticity offered by MapReduce. As discussed above, the runtime of a job is inherently dependent on its cluster size. So, if the estimated performance of a job begins to fall behind the initial estimate, or if the cloud operator has enough spare capacity to add more nodes to a job's cluster, it is possible to dynamically expand the cluster on demand. Similarly, the

[3]Note that, for the scope of this work, we do not model the impact of varying VM types; hence each job runtime model is associated with a MapReduce job class, and a fixed VM type.

Configuration	Mean (s)	Improvement
4 nodes, none added	615.84	—
8 nodes, none added	353.95	42.53%
4 nodes, add 4 at 0%map	343.14	44.28%
4 nodes, add 4 at 25%map	464.49	24.58%
4 nodes, add 4 at 50%map	514.24	16.50%
4 nodes, add 4 at 75%map	585.29	4.96%

TABLE I
IMPACT OF CLUSTER SCALING DURING MAP PHASE.

#Workers (= #reducers)	Residual Reduce Time (s)	Total time (s)
3	1503	5703
5	932	3661
7	714	2874
10	516	1987

TABLE II
IMPACT OF NUMBER OF REDUCERS ON JOB RUN TIME.

cloud provider may also "scale down" a cluster by removing nodes from a cluster when a job may want to reduce its cost, or when the operator needs to reclaim some of the nodes added as part of an earlier "scale-up" operation. To leverage this scaling opportunity, the provisioning algorithm needs information about the impact of scaling on job runtime, which is provided by the cluster scaling building block.

The job profiling building block can already estimate the complete job runtime for different cluster sizes, however, cluster scaling building block needs to couple this information with the progress the job has already made, and the impact of scaling it after the job begins. Additionally, it has to account for the differences between the Map and the Reduce phases of the job.

1) Scaling for Map Phase: We begin with an understanding of the impact of scaling on the map phase using the following setup: A wordcount job with 6 GBytes of data is executed from beginning to end on a 4 node and an 8 node physical cluster. We then experiment with starting the same job on a 4 node cluster, and adding 4 more nodes at different points in the map stage. Our results (Table II) demonstrate that benefit from cluster scaling is higher if the amount of map phase remaining is higher. Another interesting observation is that the amount of improvement when the job is executed with 8 nodes is quite close to when 4 nodes are added after the job is launched on 4 nodes. This occurs due to two reasons – first, our startup overhead of starting new VMs is very small, which makes scaling very efficient. Second, even though the newly added 4 nodes do not contain any local data, and they must fetch their input data from the original 4 nodes over the network, fetching their input data from other nodes across the network has negligible overhead in our setting (since network bandwidth and disk bandwidth on our LAN connected nodes are comparable).

Based on the above results, the cluster scaling building block for the map phase estimates the impact of scaling by leveraging the job runtime vs. cluster size model and the map progress over time model built by the job profiling building block.

2) Scaling for Reduce Phase: Unlike the map phase, the reduce phase of a MapReduce job is more complex to scale dynamically. In MapReduce and specifically the Hadoop implementation, the number of reducers for a job is a static parameter set when the job begins executing. This is because as soon as the job starts executing, the intermediate key space is statically broken up into partitions which are equal in number to the set number of reducers. Ability to dynamically change these partitions would allow dynamic scaling. While such re-partitioning would have non-zero overheads, as our next experiment shows, there are significant potential performance benefits of reduce scaling. In this experiment, for a wordcount job for a 10 GB dataset, we varied the number of nodes in the cluster and set the number of reducers to the number of nodes for each run. As can be seen from Table II, ability to use all nodes in the cluster for reduce operations significantly reduces the time spent for the reduce phase going from over 5700 seconds for a 3 node cluster to under 2000 seconds for 10 nodes.

A way to work around this limitation would be to possibly set a larger number of reducers than the number of nodes in the cluster at job start time, thus potentially allowing for future cluster scaling. However, this causes an additional overhead since if the number of reducers is larger than the number of nodes, the reduce tasks are serialized on those nodes and since each reduce task has to wait for all map tasks to finish before completing, it causes an unnecessary slowdown. As an example for a similar wordcount job for a 3 node cluster, setting the number of reducers to 25 performed 43% poorer in residual reduce time as compared to using 3 reducers.

Fixing the implementation to allow re-partitioning of the key space and thus, allow scaling reduce jobs dynamically is an important problem and part of our future work. In the current version of STEAMEngine, however, we only support cluster scaling for the map phase.

III. STEAMENGINE: PROVISIONING ALGORITHMS

This section presents two STEAMEngine provisioning algorithms that leverage the profiling and scaling building blocks. The first technique demonstrates how the building blocks could be used to meet performance goals from a cloud user's perspective, while the second presents an energy management algorithm based on the framework that can be employed by a cloud provider to reduce their system-wide energy consumption.

As part of each, we will first introduce an *initial provisioning* algorithm that allocates MapReduce jobs as they arrive by starting virtualized clusters across servers in the data center. Then, we will present a *continuous optimization* algorithm that adaptively changes the VM allocation during the execution of these jobs in order to optimize for the desired metric (performance or energy).

A. User Optimization: Performance

Problem setup: Given a MapReduce job and a deadline for its completion, the end-user provisioning algorithm seeks to meet the deadline while minimize user costs (we equate this to minimizing number of VMs assigned for the job).

Key idea: This provisioning algorithm must make effective use of profiling data to adapt the amount of resources allocated to the job if it seems to possibly miss the deadline. The job progress is accelerated by scaling the cluster to a larger size.

1) Initial Provisioning: We assume that a user specifies the data size for the submitted MapReduce job, the VM Type corresponding to the resource requirements for the VMs in the cluster, and the desired deadline by which the job must be completed. For the initial provisioning, given the data size, bootstrap profiling is used to estimate the number of VMs required to meet the deadline. This serves as an initial estimate which can be changed during continuous optimization.

2) Continuous Optimization: As the job executes, its progress may deviate from our initial estimate, and we need to continually optimize its resource allocation in order to meet the desired deadline.

If the online profiling estimate of job's finish time exceeds the given deadline, then a re-provisioning may be needed. To avoid over-reaction to small errors and also enable the online profiling to build up enough observation, we check for these violations at reasonably spaced-out execution points (e.g., every 10% of map progress). If there is indeed a need to re-provision the job, we use cluster scaling to adjust the resources for the job. As mentioned earlier, in our current implementation, we use cluster scaling during the map phase only, and its impact on the reduce phase is part of our future work.

First, we determine the additional number of VMs which need to be provisioned. Note that we need to account for overhead time of starting up additional VMs. For instance, on EC2, we experienced a startup overhead of 70 seconds to completely boot a new VM. Next, those VMs can then be added into the cluster. The process can be repeated if necessary *after* the new VMs have joined the cluster.

B. Provider Optimization: Energy

Next, we present a more complex provisioning algorithm that minimizes the total energy consumption of the cloud execution environment. Reducing energy consumption in these cloud environments is an important problem as it is a fast growing component of the operational cost in these massive scaled environments [14, 15].

Problem setup: The energy efficiency goal for a MapReduce cloud is to execute all submitted MapReduce jobs such that the total energy consumption of the physical machines is minimized. It can be assumed that as soon as all the jobs on a machine are finished, it can be put into a hibernate or sleep mode which uses a negligible amount of energy. For simplicity, we assume that all machines in the cloud data center consume an equal amount of power and do not consider fractional energy costs for a machine running at less than 100% utilization. While energy-efficient processors consume lesser power at lower utilization levels (with or without DVFS based techniques, e.g., [16]), the power variation exhibited as its utilization is varied is not significant [17]. Further, work in [18] shows that techniques that turn machines on/off can achieve higher energy savings. Under this model, the optimization goal effectively translates into minimizing the *cumulative machine uptime (CMU)* of all the physical machines in the cluster.

Key idea: As discussed in Section I-B, both the spatial resource requirements as well as the runtime of a MapReduce job need to be considered to achieve an energy-efficient allocation. In particular, as illustrated in Example 1, we would like all the machines to be spatially *well-fitted* as well as *time-balanced* [19].

1) Initial Provisioning: The submitted job specifies the size of the data for the MapReduce application, the VM Type corresponding to the resource requirements for the VMs in the cluster, and an initial provisioning size in the number of VMs desired for the virtualized cluster.

Our initial provisioning algorithm combines the notion of time balancing servers with spatially-efficient placement for a new job arriving into the system as follows: When a job J arrives, we obtain an initial estimate of its runtime T_J using the *Job Profiling* building block. Note that T_J is going to be the estimated runtime of all VMs allocated to the job J. We then define $S_{J,\delta}$ to be the set of non-empty servers such that the estimated *remaining runtimes* of all the VMs on any server $s \in S_{J,\delta}$ are within δ of T_J. This constrains the expected runtime of VMs running on any server to be within δ time units of each other, thereby limiting the *time imbalance* (TI) which is defined as the difference between the minimum and the maximum remaining runtimes of the VMs running on it:

$$TI = \max_{j=1}^{n} T_j - \min_{j=1}^{n} T_j,$$

Limiting TI to δ causes VMs on a server to finish close to each other in time, and then that server can be powered off or put into a sleep state, thereby saving power.

We then use *Best Fit* spatial placement— which aims at maximizing the utilization of spatial resources like CPU and memory— to place the VMs of the job J on a subset of servers from $S_{J,\delta}$. For this, we place VMs belonging to job J on servers in $S_{J,\delta}$ ordered by the number of VM slots available for that VM type in descending order, assuming we can fill every slot. Intuitively, we bring as many servers as close to full utilization as possible by starting with the least-utilized servers in $S_{J,\delta}$ first. If we can not place all the VMs for J on servers within $S_{J,\delta}$, we start new servers and put the remaining VMs on them as needed.

Note that δ is a system parameter that depends on the amount of time-balancing desired, and is likely to depend on various factors such as job lifetimes, number of servers in the system and their capacity, job arrival rates, etc. Intuitively, when $\delta = 0$, each job will be placed on separate servers (or with VMs on another job with identical finish time), while if

$\delta = \infty$, our provisioning algorithm reduces to a spatial-only Best Fit algorithm.

2) Continuous Optimization: As jobs progress, new jobs arrive, jobs complete, and as the collective state of the data-center changes, initial provisioning decisions may no longer be optimal. The continuous optimization algorithm adaptively addresses inefficiencies, and improves overall system energy consumption using following components:

Trigger Point: Continuous optimization is triggered when a **δ-violation** occurs: Our algorithm periodically updates the job run time prediction. If this runtime prediction deviates from the bootstrapped estimate, it checks if the servers hosting the VMs of that job violate their δ constraint. If there is a violation, the optimization algorithm leverages cluster scaling to take corrective action.

Job Selection: This step selects the most suitable job for corrective action. When a job causes a *δ-violation*, it is either finishing earlier than predicted, or later than expected. In case the job run time is lower than the predicted value, we do not correct it since we do not want to force the job to run longer. In the case when the job is taking longer than expected, we consider it as a viable candidate for accelerating its progress via cluster scaling. A *δ-violation* trigger can be caused by multiple jobs – we select the job with the longest runtime among all candidates.

Adjustment Decision: Having selected the job J for cluster scaling, we determine the magnitude of the scaling operation, and how to provision the additional VMs. If there are multiple servers that violate the δ constraint, we pick the one with the largest violation. Having picked this server, if T_0 is the runtime of the shortest job on the server, we define $T_1 = T_0 + \delta$ as the target runtime that the selected job J should be scaled to. Next, we use the cluster scaling building block to obtain the number of VMs required to reduce the runtime of J down to T_1, and we call this number V_{T_1}.

The next step is to choose where to start up these additional VMs. In case the data center does not have enough resources to provision these new VMs, we abort the scaling operation. If resources are available, we attempt to provision the VMs on servers that are already powered on and if that is inadequate, only then are suspended servers brought back online. Note that any of the scaling decisions are employed only if it reduces the overall CMU of the system, i.e. $\Delta CMU < 0$.

Concretely, we first consider only the set S_δ of currently powered-on servers that would not incur a δ-violation if one of the new VMs were started on it. We then prioritize servers in S_δ by their estimated uptime (corresponding to their currently longest-running job), and consider placing newly added VMs onto the servers in this priority order. If we run out of servers in S_δ and still have remaining VMs to be placed, then we consider the cost of starting up a new server and adjust the ΔCMU accordingly. Based on our calculations, if ΔCMU is positive, then by adding V_{T_1} additional VMs, we would incur a penalty in longer machine uptimes, *and thus we do not perform a cluster scaling operation*. If ΔCMU is negative, however, then *we have found an energy-efficient cluster scaling operation which both increases MapReduce performance and shortens cumulative machine uptime*. At this point, we start V_{T_1} additional VMs for the job J on the selected target servers.

IV. Evaluation

In this section, we show the benefits of the STEAMEngine framework by evaluating the provisioning algorithms which leverage the framework building blocks.

A. Methodology

We used two environments in our evaluation, a public cloud setting and a local testbed.

Public Cloud: We utilized Amazon EC2 [5][4]. Our VM instances were of the m1.small type that is defined as 1 CPU core, 1.7 GB memory, 160 GB local storage, and running on a virtualized Fedora Core 8 32-bit platform. We used Amazon S3 to store our 10 GB data set that we used for many of our experiments.

Local Testbed: Our local testbed consists of 6 physical machines interconnected with Gigabit Ethernet. Each machine is a dual core 800 MHz processor with a 250 GB hard drive and 2 GB memory. Each machine is running Xen 3.2 and Debian operating system with the 2.6.24 Linux kernel. Our operating environment supports 3 different VM types, that vary in CPU and memory sizes (VM type 1: 128 CPU credits-768 MB memory, VM type 2: 128 CPU credits-640 MB memory, and VM type 3: 256 CPU credits-256 MB memory). Each machine stores VM images for each of the VM types, which are used to instantiate VMs for each job.

Workloads: Our MapReduce platform is Hadoop 0.20.1. We experiment with four different workloads as representative MapReduce applications: Sort, Grep, Wordcount, PiEstimator (Pi). In our local testbed, we assume that each application is associated with a VM type: Sort uses VM type 1, Wordcount and Grep type 2, and Pi uses type 3.

B. Performance Optimization Evaluation

We first demonstrate the benefit of our performance optimizing provisioning algorithm (Section III-A). In particular, we evaluate the benefit of the online profiling and cluster scaling building blocks in this algorithm, for which we intentionally started jobs with inaccurate cluster sizes, so as to generate runtime estimate updates and trigger cluster scaling in order to meet the given deadlines.

We implemented our algorithm in EC2, composed of our cluster scaling building block and online profiling[5]. As determined by the algorithm that monitors the Hadoop job progress, new VMs are added on the fly as needed in order to finish the job by the deadline. Important parameters in this algorithm are the frequency of trigger points, which we set to be every 10% map progress, and the VM startup overhead, which we measured as 70 seconds in EC2.

[4]We chose EC2 over Amazon Elastic MapReduce [3] (which internally also uses EC2) for greater control.

[5]We used the bootstrap estimate for the residual Reduce-phase.

Job	VMs	Orig Runtime (sec)	Deadline (sec)	Final Runtime (sec)	Cluster Scalings
Pi	4	733	500	*520	+3 VMs @ map=10%
Pi	4	733	500	480	+4 VMs @ map=10%
Grep	4	1057	600	577	+6 VMs @ map=10%
Grep	4	1057	600	585	+5 VMs @ map=10%
WdCnt	10	1987	1700	1674	+2 VMs @ map=20%, +1 VM @ map=36%
WdCnt	10	1987	1700	1698	+2 VMs @ map=20%, +1 VM @ map=66%

TABLE III
CLUSTER SCALING ALLOWS MAPREDUCE APPLICATIONS TO BE REPROVISIONED TO MEET DEADLINES IF NOT GIVEN ENOUGH RESOURCES AT RUNTIME. ONLY ONE TRIAL* DID NOT MEET ITS DEADLINE.

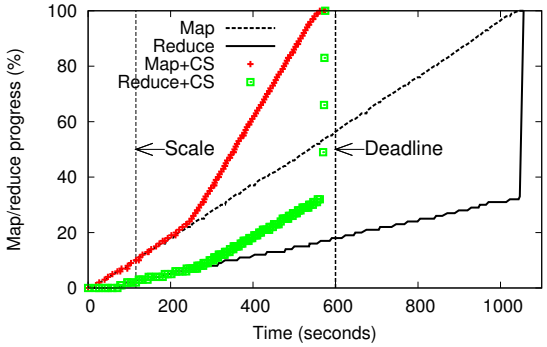

Fig. 5. 10 GB Grep job progress on EC2.

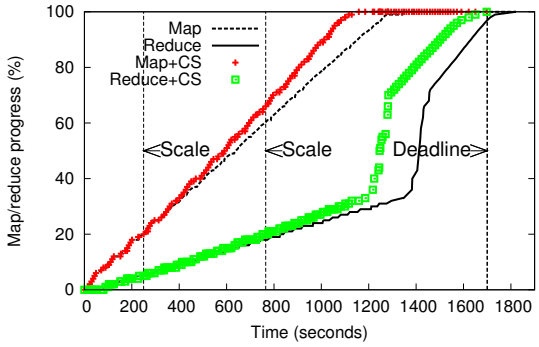

Fig. 6. 10 GB Wordcount job progress on EC2.

The full results are listed in Table III, and time series for runs of Grep and Wordcount can be seen in Figures 5 and 6 respectively.

Pi was run with 900 maps and 9500 samples per map. In the first run, Pi missed its deadline by only 20 seconds with a runtime of 520 sec, which was the only deadline violation we experienced in all of our runs. However, Pi beat its deadline by 20 seconds in the second run, when it added 4 VMs instead of just 3. Grep was run on 10 GB random data. The job met its deadline for both runs. Figure 5 shows the detailed time series of the first run.

Wordcount was run on the same 10 GB random data. This is our reduce-heavy workload, where 28% of the total expected job runtime would be accounted for in the residual Reduce phase. The results as seen in Figure 6 show that cluster scaling was performed twice to meet the deadline. This shows the ability of cluster scaling to overcome inaccuracies exhibited in the online runtime estimations. The graph also illustrates the impact of the VM startup overhead: after the first cluster scaling, the 70 sec overhead of starting new nodes corresponded to map=26%, and therefore the first trigger point after the initial cluster scaling was at map=36% and increments of 10% for further trigger points thereafter. This is why the second scaling is done at 36% in one case and 66% in the second case.

C. Energy Optimization Evaluation

In this section we show the benefits of using our energy optimization algorithm (Section III-B), both in the initial provisioning and continuous optimization stages. Since this is a provider-side optimization, these experiments were conducted on our local testbed, where we could control VM allocations.

1) Benefit of Initial Provisioning: We first show the benefit of using initial provisioning based on accurate bootstrap profiling (our bootstrap profile database contains run times for the chosen jobs resulting in an exact match in our runtime estimation) by comparing it to the initial provisioning with spatial best fit, thereby demonstrating the benefit of exploiting the spatio-temporal tradeoff.

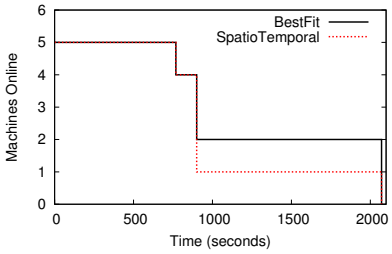

Fig. 7. The spatio-temporal algorithm uses fewer machines, has a lower CMU and thus saves energy.

In this experiment, we had 3 MapReduce jobs—(1) Sort job with 3 VMs and data size 950 MB, resulting in a runtime of 768 seconds; (2) Wordcount job with 8 VMs and data size 4 GB, resulting in a runtime of 900 seconds; (3) a Pi job with 6 VMs and 77.5 million samples, resulting in a runtime of 2071 seconds. We compared a spatial best fit algorithm against a spatio-temporal algorithm that incorporated the runtimes of the jobs in its VM placement decisions. Our results showed that while both schemes utilized 5 physical machines in total, the total uptime of the servers in the spatio-temporal technique was lower by 920 seconds: 5549 sec vs 6469 sec (a savings of

14%). The number of physical machines online for both the spatial best-fit and our spatio-temporal algorithms can be seen in Figure 7.

2) Benefit of Continuous Optimization: We now evaluate the benefit of continuous optimization after initial provisioning is done, but when conditions change during their execution.

In this experiment, we have 2 Pi jobs; the first job has 3 VMs, 900 maps, 9950 samples/map, and takes 1545 seconds while the second one has 6 VMs, 1386 maps, 4000 samples/map and takes 990 seconds. We introduce an error in the runtime estimate to induce cluster scaling in the experiment. Specifically, we estimate the runtime of the first Pi job to be the same as the second Pi job. Thus, our initial provisioning algorithm co-places these two jobs on the same physical machines, and then incurs a higher CMU due to the error in the estimation. However, STEAM with continuous optimization detects the error in runtime estimate using online profiling when the job is 10% completed, which triggers a δ violation, which in turn triggers cluster scaling to correct the violation (as described in Section III-B). The results can be seen in Table IV. The cluster scaling logic adds 1-2 additional VMs for the longer running job to remove the δ violation, resulting in a total runtime of 1104 seconds on average for the longer job. This results in a CMU savings of 335 seconds (11%), averaged over three runs.

Summary of Results: Our results show that our building blocks, job profiling and cluster scaling, were successfully synthesized into two provisioning algorithms to enable the agility of MapReduce applications to meet user-specified deadlines and provider-specified energy goals respectively.

V. Related Work

Resource provisioning in MapReduce. Recent work [20] investigating resource sharing across MapReduce jobs (e.g., Yahoo's capacity scheduler and Facebook's fairness scheduler) focuses on fairly allocating resources across jobs, and not on how the jobs are co-placed. Quincy [21] is a framework for scheduling concurrent jobs to achieve fairness while improving data locality. The authors note as part of their future work that Quincy can be further improved by leveraging information similar to that provided by STEAMEngine's job profiling and cluster scaling components. Sandholm et al [22] presented a resource allocation system that enables resource allocation to be varied across different job stages resulting in performance improvement while meeting the cost budget. Our end user provisioning algorithm is complementary to this approach, and could be employed to obtain cost budget required as input to this algorithm. Work in [23] notes that Amazon Spot Instances can be leveraged to dynamically improve the performance of a MapReduce job. This finding is similar to that of our cluster scaling building block. However, our work presents a more comprehensive framework with a set of building blocks that can be exploited by different provisioning algorithms.

Our energy minimizing provisioning algorithm addresses a growing concern regarding energy consumption in MapReduce. Recent work [24] shows the energy inefficient use of resources within a Hadoop job, and proposes a new data layout that enables turning off nodes to save energy, while trading off performance in the process. Work in [25] studied the impact of different parameters of a Hadoop job and cluster such as replication level, input size, etc. to understand how they impact the energy consumption. Our work, however, focuses on the opportunities to save energy across multiple jobs rather than within a single job. Investigating an integration of these two approaches is an interesting avenue for further work.

MapReduce optimizations. The growing popularity of MapReduce has also spurred a large body of work on improving the Hadoop implementation (e.g., [26, 4]). These can leverage STEAMEngine's building blocks beyond resource provisioning problems. Work in [27] proposes an approach similar to our job profiling to determine optimal configuration parameters for a MapReduce job. Mantri [13] also uses profiling to detect outliers in a MapReduce job, and proactively take corrective action. As part of our future work, we would like to investigate if more detailed models [28, 29] that predict MapReduce completion time could potentially enhance the accuracy of our job profiling building block.

Resource allocation in virtualized environments. A large body of work has explored application placement in a virtualized data center to minimize energy consumption [9], perform load balancing [12, 10] or for server consolidation [11]. These approaches essentially focus on achieving spatial efficiency when placing applications and deal with temporal variations by continually adjusting the placement using VM migrations. In contrast, our algorithms are proactive in nature exploiting the runtime estimates of MapReduce jobs based on their inherent parallelism. Steinder at al [6] also investigated resource allocation for heterogeneous virtualized workloads driven by high-level performance goals, while we consider a broader set of metrics such as energy and cost.

Parallel Processing. Finally, parallel job scheduling in the context of massively parallel supercomputers is a well studied area [30, 31], and shares interesting similarities and differences with our work. "Space slicing" in these parallel machines enables packing as many jobs as possible in the given set of resources, "malleable" parallel jobs resemble the elastic nature of MapReduce; and similar to our observation, estimating job completion time can potentially aid in scheduling decisions in these systems as well. We exploit properties of VMs to enable easy cluster scaling, and while many of these algorithms are focused on performance or fairness, we also support metrics such as energy management and cost.

VI. Conclusions

Intelligent provisioning of MapReduce jobs in a virtualized cloud environment enables end-users/providers of a MapReduce service to effectively optimize their deployments. Our work identified the spatio-temporal opportunities unique to the MapReduce paradigm, and proposed STEAMEngine, a provisioning framework to leverage these opportunities. STEAMEngine consists of provisioning algorithms to optimize both user-side as well as platform-side metrics through a

Expt	Initial TI (s)	Target δ (s)	TI after CS (s)	CS VMs Added	CMU w/CS (s)	CMU no CS (s)	CS CMU Savings
Run 1	504.6	200	143.1	1	2832.0	3122.0	9.3%
Run 2	593.7	200	34.8	2	2638.1	3126.2	15.6%
Run 3	568.1	200	145.9	1	2835.9	3062.5	7.4%
Avg	555.5	200	84.74		2768.7	3103.6	10.8%

TABLE IV
CLUSTER SCALING (CS) USED FOR TIME BALANCING TO ACHIEVE ENERGY SAVINGS. TI CORRESPONDS TO TIME IMBALANCE.

set of common building blocks—Job Profiling and Cluster Scaling—that estimate and alter the temporal characteristics of the MapReduce job. Our work describes two such novel provisioning algorithms—a cloud user-driven performance optimization and a cloud provider-driven energy optimization. Our evaluation shows that our performance optimizing algorithm, running on Amazon EC2, enabled MapReduce jobs to meet their deadlines even with inaccurate initial information. Further, our energy optimization algorithm saved up to 14% energy in our local cluster when simultaneously executing multiple jobs.

Finally, STEAMEngine is easily extensible for plugging in new building blocks and provisioning algorithms. For instance, a building block based on scaling VM instances up/down (e.g., dynamically adding CPU resources to a VM instance) can also help optimize MapReduce deployments. Migrating VMs between servers can be a helpful building block for a cloud provider. Similarly, provisioning algorithms with other optimization objectives such as dynamic load balancing across resources, or QoS-driven optimization can leverage STEAMEngine's building blocks.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. of OSDI*, 2004.

[2] "Hadoop," http://hadoop.apache.org.

[3] "Amazon EMR," http://aws.amazon.com/elasticmapreduce/.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of OSDI*, 2008.

[5] "Amazon EC2," http://aws.amazon.com/ec2.

[6] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. M. Chess, "Server virtualization in autonomic management of heterogeneous workloads," in *Integrated Network Management*, 2007, pp. 139–148.

[7] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, 2001.

[8] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing server energy and operational costs in hosting centers," in *Proceedings of ACM SIGMETRICS*, 2005.

[9] A. Verma, P. Ahuja, and A. Neogi, "pMapper: Power and Migration Cost Aware Placement of Applications in Virtualized Systems," in *ACM Middleware*, 2008.

[10] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," in *Proceedings of NSDI*, 2007.

[11] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *Proceedings of 10th IEEE/IFIP Network Ops and Management Symp. (NOMS 2006)*, 2006.

[12] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proceedings of IEEE/ACM Supercomputing*, 2008.

[13] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, and B. Saha, "Reining in the outliers in map-reduce clusters," in *Proceedings of OSDI*, 2010.

[14] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, 2008.

[15] C. L. Belady, "In the data center, power and cooling costs more than the it equipment it supports," *Electronics Cooling Magazine*, vol. 13, no. 1, pp. 24–27, 2007.

[16] T. Burd, T. Pering, A. Stratakos, and R. Brodersen, "A Dynamic Voltage-Scaled Microprocessor System," in *IEEE ISSCC*, 2000.

[17] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu, "Delivering Energy Proportionality with Non Energy-Proportional Systems—Optimizing the Ensemble," in *USENIX HotPower*, 2008.

[18] E. M. Elnozahy, M. Kistler, and R. Rajamony, "Energy-efficient server clusters," in *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, 2002.

[19] M. Cardosa, A. Singh, H. Pucha, and A. Chandra, "Exploiting Spatio-Temporal Tradeoffs for Energy Efficient MapReduce in the Cloud," Dept. of CSE, Univ. of Minnesota, Tech. Rep. 10-008, Apr. 2010.

[20] "Scheduling in hadoop," http://www.cloudera.com/blog/tag/scheduling/.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.

[22] T. Sandholm and K. Lai, "Mapreduce optimization using dynamic regulated prioritization," in *ACM SIGMETRICS/Performance*, 2009.

[23] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: Using spot instances for mapreduce workflows," in *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[24] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," in *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[25] Y. Chen, L. Keys, and R. H. Katz, "Towards Energy Efficient MapReduce," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-109, Aug 2009.

[26] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of NSDI*, 2010.

[27] S. Babu, "Towards automatic optimization of mapreduce programs," in *ACM SOCC*, 2010.

[28] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson, "Statistics-driven workload modeling for the cloud," in *Proceedings of 5th International Workshop on Self Managing Database Systems (SMDB)*, 2010.

[29] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of mapreduce pipelines," in *Proceedings of IEEE ICDE*, 2010.

[30] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," *Springer-Verlag Lecture Notes In Computer Science*, vol. 949, 1995.

[31] D. G. Feitelson, "Job scheduling in multiprogrammed parallel systems," *IBM Research Report*, vol. RC 87657, 1997.