Technical Report

TR 10-023

STEAMEngine: Driving MapReduce Provisioning in the Cloud

Michael Cardosa, Piyush Narang, Abhishek Chandra, Himabindu Pucha, and Aameek Singh

September 28, 2010

STEAMEngine: Driving MapReduce Provisioning in the Cloud*

Michael Cardosa, Piyush Narang, Abhishek Chandra, Himabindu Pucha†*, Aameek Singh*†
University of Minnesota †*IBM Research - Almaden*
{*cardosa,piyang,chandra*}*@cs.umn.edu* {*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 a 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 resource 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 6-node Xen/Hadoop cluster show the benefits of STEAMEngine through improvements in performance and energy via the use of these algorithms and building blocks.

1 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 [17] 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.

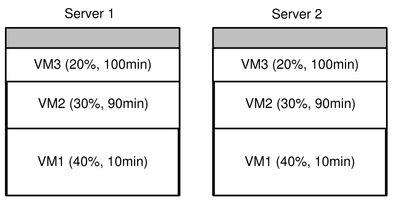
This growing interest from users of MapReduce is suitably matched by enterprises/service providers hosting massive scale infrastructure for MapReduce. Several enterprises including Facebook, Yahoo, and Microsoft run their own shared infrastructure, akin to a private cloud, where different MapReduce jobs within the enterprise are simultaneously executed. Similarly, MapReduce offered as a service in the public cloud (e.g., Amazon Elastic MapReduce [1]) shows great promise. Often in such cloud environments, server virtualization is used for providing multi-tenancy and isolation. For the scope of this paper, we also assume virtualization, 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 its deployment for system-wide goals such as maximizing system throughput, minimizing energy consumption, and load balancing, as new MapReduce jobs arrive and old ones finish. While optimizing different metrics important to the MapReduce end-user/provider require different algorithms, at their core they all leverage common resource provisioning 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. Our work, thus, explores this key issue of resource provisioning for MapReduce, both within and across multiple jobs.

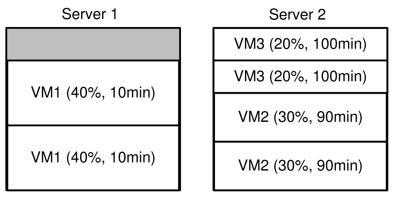
1.1 Spatio-Temporal Tradeoffs

Resource provisioning for Internet applications in a public or a private cloud setting is well studied [35, 11, 12]. However, 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

*This work was supported in part by NSF Grant CNS-0643505.



(a) Spatially-efficient VM placement



(b) Space-time-efficient VM placement

Figure 1: Spatio-Temporal tradeoff in MapReduce provisioning.

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" 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 [37, 38, 25, 34] 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 algorithm, 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.

1.2 Research Contributions

To exploit spatio-temporal tradeoffs as described above, a key requirement is the ability to predict and alter the temporal behavior of a MapReduce job based on its spatial resource allocation. Further, irrespective of the optimization objective, we argue that any MapReduce provisioning algorithm can benefit from this characteristic unique to the MapReduce paradigm. To that end, we present a MapReduce provisioning framework, *STEAMEngine*[1], that makes two main contributions.

First, it provides *building blocks* that expose the temporal behavior of MapReduce jobs by exploiting unique MapReduce characteristics. Further, these building blocks expose a generic API to easily enable a variety of provisioning algorithms (Section 2).

- Our *Job Profiling* building block exploits the *predictable and equitable behavior* of a MapReduce job to predict its completion time. Each node in a MapReduce cluster operates on equal sized blocks which contain similar content (records). For a homogeneous cluster, this results in nearly equitable resource consumption and time required to process each block. Thus, by utilizing job profiles available for different data set sizes or cluster sizes, this building block models and predicts the job runtime. Further, it improves this prediction by observing the job progress over time.

- The *Cluster Scaling* building block exploits the *ease of scaling* in a MapReduce job, i.e. dynamically in-

[1]STEAM is an acronym for Space-Time based Elastic and Agile MapReduce.

creasing or decreasing the size of the cluster running the MapReduce job, to alter the time required to finish a job. For example, if a new node is added to a cluster, MapReduce can automatically schedule tasks on the new node to take advantage of the increased cluster capacity, thereby reducing the completion time. This building block models and predicts the impact on job runtime if the cluster size is modified by accounting for the current job state and the change in resources.

Our next contribution is the design and implementation of two *novel MapReduce provisioning algorithms* based on these building blocks, that demonstrate the importance of considering spatio-temporal tradeoffs, and illustrate the utility of the building blocks (Section 3).

- Our end-user provisioning algorithm tries to *optimize performance* of a job—meeting a job runtime deadline while minimizing cost. It leverages Job Profiling to pick the minimum number of VMs required to meet the deadline, and Cluster Scaling to adjust the runtime if it deviates from the initial prediction. Without STEAMEngine, such performance optimization will need to be performed manually by the user.

- Our provider-side provisioning algorithm *minimizes system energy consumption* as new MapReduce jobs arrive, and old ones finish. Job Profiling provides the runtime estimate of an incoming MapReduce job, and thus the uptime estimate of its VMs. These VMs are then intelligently co-placed to minimize the cumulative up time (*CMU*) of the system. Cluster Scaling helps correct deviations from initial predictions. In the absence of STEAMEngine, this energy minimization algorithm will devolve into performing inefficient VM placement using best-fit spatial packing.

We have evaluated STEAMEngine on both Amazon EC2 [3], as well as a local 6-machine 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 4).

Finally, our STEAMEngine framework is extensible and can easily accommodate other building blocks as well as provisioning algorithms. For instance, VM migration and instance scaling can be used as additional building blocks, while provisioning algorithms can be developed for dynamic load balancing or QoS-based optimization (Section 6).

2 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. The VM type (CPU, memory, storage) and the number of VMs is chosen by the user submitting the job[2]. 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 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 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. Thus, as part of our STEAMEngine framework, in this section we present a unifying framework for MapReduce provisioning that provides 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.

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-3.1), or cloud provider-driven energy optimization (Section-3.2). To exploit the opportunities provided by MapReduce, these algorithms require some common information and mechanisms which are provided by the building blocks.

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.

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

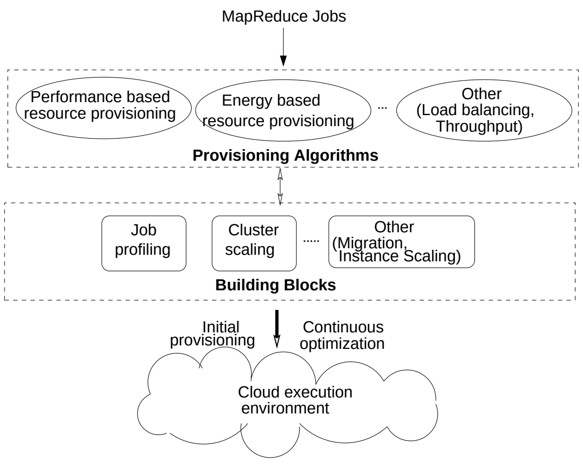

MapReduce Jobs
Performance based resource provisioning
Energy based resource provisioning
Other (Load balancing, Throughput)
Provisioning Algorithms
Job profiling
Cluster scaling
Other (Migration, Instance Scaling)
Building Blocks
Initial provisioning
Continuous optimization
Cloud execution environment

Figure 2: STEAMEngine Framework

2.1 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: (i) an offline 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, and (ii) an online profiling technique which captures the progress of an executing MapReduce job to update these estimates in a more fine-grained manner.

2.1.1 Offline profiling

The goal of the offline profiling technique is to estimate the runtime of a job before the job begins execution. This estimate 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 (due to Observation 1 below).

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. The goal of the offline profiling is to capture this function f for a given MapReduce job class. Note that, for the scope of this work, we do not model the impact of varying VM types; each job runtime model is associated with a MapReduce job class, and a fixed VM type.

To show that building such a profile 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 4. Due to space constraints, we only present the results for sort as the conclusions are similar for the other benchmarks.

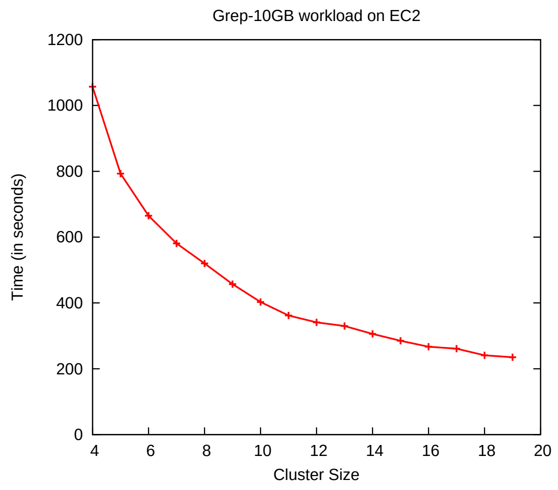
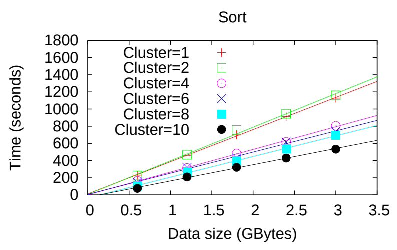
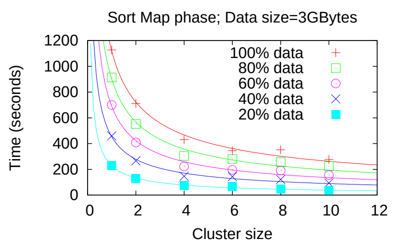


Figure 4: 10 GB Grep job runtimes on EC2 as a function of cluster size.

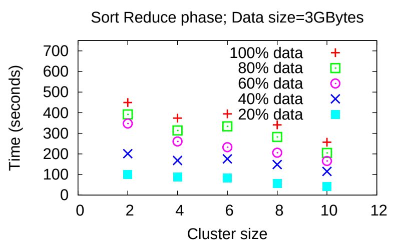
Dependence on data size: Figure 3(a) shows the runtime of the benchmarks as a function of data size (the lines in the figure are fitted to the data points). As shown in the figure, for a given cluster size, the runtime increases linearly with the input data size (Observation 1). This result implies that 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



(a) Data size dependence



(b) Map phase dependence on cluster size



(c) Reduce phase dependence on cluster size

Figure 3: Relation between job runtime, input data size, and cluster size for sort job.

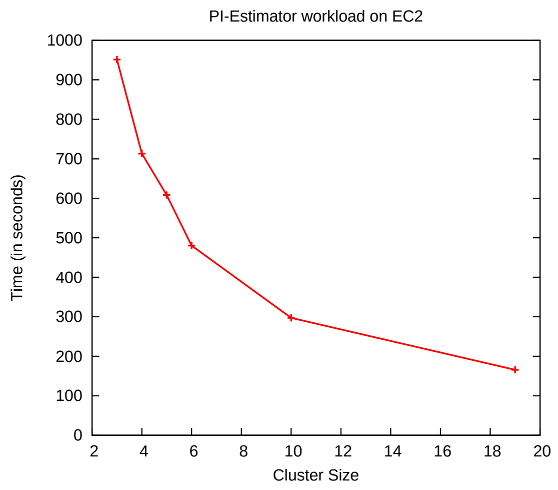


Figure 5: 8.5M-sample PiEstimator job runtimes on EC2 as a function of cluster size.

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.

Dependence on cluster size: To understand the dependence on cluster size, we consider the time to complete the Map phase (the time to complete all the map tasks) and the residual Reduce phase (the remaining Reduces after the Map phase has completed) for these benchmarks separately (Figures 3(b) and 3(c) and the map-heavy Grep and PiEstimator workloads in EC2 in Figures 4 and 5, respectively). As we can see, for a given data size, the Map phase runtime shows an inverse relation ($1/n$) to the cluster size (Observation 2). This is because the Map phase is fairly data-parallel for most applications, with each map task working independently on its own data block. On the other hand, the Reduce phase depends a lot on the application, and could be insignificant (e.g., for Pi and Grep) or could involve a fair amount of data movement and output data writing, which could result in non-linear overheads (e.g., for sort, as shown in the figure). Thus, to capture the relation to cluster size, both the Map and the residual Reduce phases need to be modeled separately, and the total runtime of a job will be determined by combining the two (Observation 3).

Based on the above results, our offline 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, if the database contains multiple values corresponding to the same cluster size (but different data sizes) as the incoming job, we can perform a linear extrapolation from these values (based on Observation 1 above). If we don't have multiple matches for the cluster size of the incoming job, but have values corresponding to the same input size, then we extrapolate for the Map and residual Reduce phases based on Observations 2 and 3 above. 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. Later, this estimate is combined with the online progress estimation to account for any errors in offline profiling.

Building block interface: In order to access this building block, STEAMEngine provides the following two APIs:

`getOfflineEstimate`(job, dataSize, clusterSize) which returns the estimated runtime for that job, and

`getClusterSize`(job, dataSize, desiredRuntime) which returns the estimated cluster size required to finish the job within $desiredRuntime$. These APIs are accessible to both cloud provider as well as the cloud user. The cloud provider can use this estimate to decide VM placement in a spatio-temporal manner and the cloud user can use this API to determine its initial cluster size.

2.1.2 Online profiling

While the offline profiling technique provides us with a coarse-grained estimate of the total job runtime based on the parameters available a priori, the estimate may sometimes be inaccurate. This could be either because we do not have sufficient historical information to make accurate estimates, the model used for estimation may turn out to be inaccurate, or the performance of a job may vary due to failures, stragglers, etc. As a result, we also use an online profiling technique, that provides us with more fine-grained information about the progress of a running job, and enables updating the estimates of its runtime on the fly.

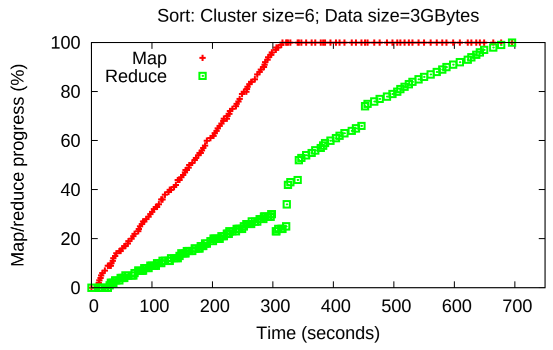


Figure 6: Online progress of a MapReduce sort job.

Figure 6 shows the online progress of the Map and Reduce tasks for the sort benchmark. 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. Based on these observations, in our current implementation, we extrapolate the Map progress linearly based on the online estimates. We start by using the offline estimate for the residual Reduce phase runtime, and once the Map phase is over, we predict it in the same fashion as we predicted the Map phase.

Interface: In order to use the online profiling building block, first the job is registered with STEAMEngine for online profiling using the `registerForProfiling`(jobId) API, where jobId is the ID of the MapReduce job. Once the job is registered, the online profiling estimate can be obtained using `getOnlineEstimate`(jobId) which returns expected map, reduce and total job runtime at that point of time based on the performance of the MapReduce job so far. This interface is available to both cloud provider and cloud users who, if desired, can then adjust the amount of resources allocated to the job using the cluster scaling building block described next.

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 1: Impact of cluster scaling during Map phase.

2.2 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 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.

2.2.1 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 2) 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.

Interface: This STEAMEngine building block provides four APIs that can be used by the cloud user or the provider – (1) `getScalingEstimate`(jobId, newClusterSize, mapProgressPoint) provides estimated runtime for the job with ID $jobId$ if the cluster is scaled to $newClusterSize$ when map progress is at $mapProgressPoint$, (2) An inverse lookup API `getNewClusterSize`(jobId, mapProgressPoint, newDesiredRuntime) which returns the new cluster size required in order to adjust the runtime of the job to $newDesiredRuntime$, (3) `scaleCluster`(jobId, newClusterSize) which uses cloud VM provisioning (or de-provisioning) to create (or delete) VMs in the cluster to bring its size to $newClusterSize$ and edits MapReduce cluster configuration to reflect the new cluster size, and (4) an exclusive cloud provider API `scaleCluster`(jobId, newClusterSize, vmToPmMap) which instead of using default VM placement on the physical machines (PMs), explicitly specifies the placement; such an API is useful when the provisioning algorithm needs to dictate how all VMs are placed during the execution of the scaling operation.

2.2.2 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 2, 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.

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

Table 2: Impact of number of Reducers on job run time.

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.

3 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 alters VM allocations during the execution of these jobs in order to optimize for the desired metric (performance or energy).

3.1 User Optimization: Performance

This end-user MapReduce provisioning algorithm optimizes job performance.

Problem setup: Given a MapReduce job and a deadline for its completion, the 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 offline profiling data to estimate the initial number of resources it should allocate to the job. Further, if such profiling data is unavailable or is inaccurate, continuous optimization via online profiling and cluster scaling must be used to accelerate the progress of the job, and meet the given deadline.

3.1.1 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, the offline profiling API `getClusterSize` is used to estimate the number of VMs required to meet the deadline.

3.1.2 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. For such continuous optimization, first the job is registered with the online profiling building block using the `registerForProfiling` API, which lets the building block monitor the progress of the job.

If the online profiling estimate of job's finish time falls beyond 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) by using the `getOnlineEstimate` API.

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 use the `getNewClusterSize` API at the current map progress point to get the additional number of VMs which need to be provisioned. Note that the `newDesiredRuntime` argument to the API needs to adequately 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 using the `scaleCluster` API of the building block. The process can be repeated if necessary *after* the new VMs have joined the cluster.

3.2 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 [26, 6].
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., [8]), the power variation exhibited as its utilization is varied is not significant [36]. 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 1.1, both the resource requirements as well as the expected runtime of a MapReduce job need to be considered to achieve an energy-efficient allocation. In particular, as illustrated in Example 1, to achieve a better space-time tradeoff, we would like all the machines to be spatially well-fitted (to avoid spatial wastage), as well as time-balanced (to avoid temporal wastage) [9].

3.2.1 Initial Provisioning

When a job is submitted to the system, it is placed using the initial provisioning step. 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 use `getOfflineEstimate` to obtain an estimate of its runtime T_J. 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* 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 $S_{J,\delta}$. 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. Pseudocode describing this approach for initial provisioning is shown in Algorithm 1.

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.

3.2.2 Continuous Optimization

As jobs progress, new jobs arrive, jobs complete, and as the collective state of the datacenter changes, our initial provisioning decisions could potentially be sub-optimal. Our continuous optimization algorithm addresses these inefficiencies, and improves overall system energy consumption via the following components:

Trigger Point: Continuous optimization is triggered when a δ**-violation** occurs as follows. Our algorithm periodically queries the online job profiling `getOnlineEstimate` API to update the job run time prediction. If this online run time prediction deviates from the offline profiling 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 originally 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. Since a δ**-violation** trigger can be caused by multiple jobs, we select the job with the longest runtime among all the candidate jobs amenable to cluster scaling.

Algorithm 1 PROVISIONJOBDELTA(FLOAT $delta$, JOB j, VMTYPE vt, INT $numVms$, FLOAT $estRuntime$, SERVER $servers$[])

```
Server $candidates$[] = {}
int candidateVmSlots[] = {}
int $totalVmSlots$ = 0
for each $s$ in $servers$ do
    $min$ = min($s$.getMinRuntime(),$estRuntime$)
    $max$ = max($s$.getMaxRuntime(),$estRuntime$)
    if $max - min > delta$ then
        continue;
    end if
    int $VmSlots$ = $s$.canPlace($vt$)
    if $VmSLots > 0$ then
        $totalVmSlots += VmSlots$
        $candidates$.add($s$)
        $candidateVmSlots$.add($VmSlots$)
    end if
end for
if $candidateVmSlots < numVms$ then
    return $false$
end if
Sort $candidates$ by $candidateVmSlots$ descending
for each $s$ in $candidates$ do
    Assign $VmSlots$ from $candidateVmSlots$ entry
    if $numVms \geq VmSlots$ then
        $numVms -= VmSlots$
        $DoBestFit(s, vt, vmSlots)$
        Remove $s$ from $candidates$
        if $numVms == 0$ then
            return $true$
        end if
    end if
end for
Reverse $candidates$
for each $s$ in $candidates$ do
    Assign $VmSlots$ from $candidateVmSlots$ entry
    $vmsToPlace = min(VmSlots, numVms)$
    $numVms -= VmsToPlace$
    $DoBestFit(s, vt, VmsToPlace)$
    if $numVms == 0$ then
        return $true$
    end if
end for
return $true$
```

Adjustment Decision: Having selected the job J for cluster scaling, we must now 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, to make the server time-balanced again. Next, we use the cluster scaling building block's `getNewClusterSize` API 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 using the provider-specific `scaleCluster` API which also provides the physical servers to use while placing the new VMs.

4 Evaluation

In this section, we evaluate the accuracy of our building blocks and the benefits of our provisioning algorithms which leverage these building blocks.

4.1 Methodology

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

Public Cloud: We utilized Amazon EC2 [3][3]. 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 10GB 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.

[3]We used EC2 instead of Amazon Elastic MapReduce [1] to retain control over how we optimized our provisioning.

4.2 Accuracy of Job Profiling

We first measure the accuracy of our online and offline profiling algorithms, since our cluster scaling building block builds on top of this accuracy.

4.2.1 Offline Profiling

For this experiment, we first profiled the different benchmarks by running them on 5 physical servers in our local testbed with different combinations of data size and number of VMs. These runs generated a set of 90 data points, composed of 5 data sizes combined with 6 cluster sizes for each of 3 workloads, as part of our offline profiling database. We then evaluated the accuracy of our offline profiling algorithm as follows: we conducted 3000 runs of our profiling algorithm on a randomly selected subset (60 points) of the profiling database to estimate the remaining 30 data points, and computed the estimation error. Offline profiling had an average error of 9.5% and standard deviation of 15.8%.

4.2.2 Online Profiling

We next evaluated the accuracy of our online profiling algorithm at different stages of a job progress. Here, we ran a job and at many points during its run, we estimated its Map-phase or residual Reduce-phase finish time using our online profiler, which was compared to the actual finish time.

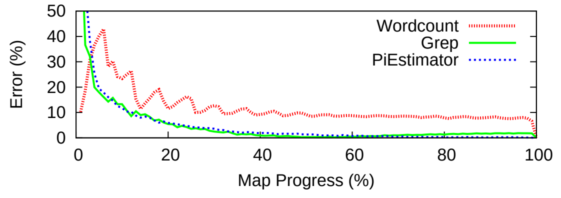


Figure 7: Error experienced while predicting the Map phase completion times for standard workloads in EC2.

Figure 7 shows the error of the online estimation of the Map phase for our standard workloads (Wordcount–10 GB data with 10 nodes, Grep–10 GB data with 4

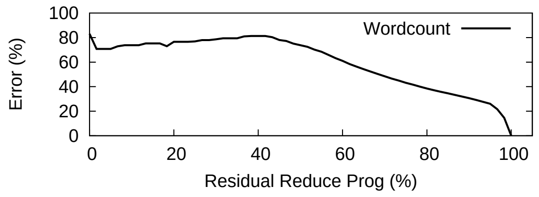


Figure 8: Error while predicting residual Reduce phase completion times for Wordcount. Staggered time series progress (as in Figure 10) contributes to high error.

nodes, Pi–8.5 million samples on 4 nodes) while they execute in EC2, averaged over 3 runs. We bootstrap with no initial data, but the estimation accuracy naturally improves as the job continues executing, as more time series data becomes available to the online profiler. For Pi and Grep, the error was $<5\%$ beyond map=22%, and $<2\%$ beyond map=38%. For Wordcount, progress is staggered at the beginning, where its low initial error appears to be a random phenomenon, and stabilizes with more readings. Overall, its error was $<20\%$ beyond map=12%, and $<10\%$ beyond map=43%.

Figure 8 shows the error of the online estimation of the residual Reduce phase. As in Figure 10, the Reduce phase progress is staggered for Wordcount, leading to larger error than the Map phase predictions. One may instead use the offline prediction for residual Reduce time instead of the online prediction if available. More complex modeling for online estimations of residual Reduce progress are necessary for higher accuracy, and we leave that to future work.

4.3 Performance Optimization Evaluation

We now demonstrate the benefit of our performance optimizing provisioning algorithm (Section 3.1). In particular, we evaluate the benefit of the cluster scaling building block in this algorithm, for which we intentionally started jobs with inaccurate cluster sizes to 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[4]. 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.

The full results are listed in Table 3, and time series for single runs of Grep and Wordcount can be seen in Figures 9 and 10, respectively.

[4]We used the offline estimate for the residual Reduce-phase.

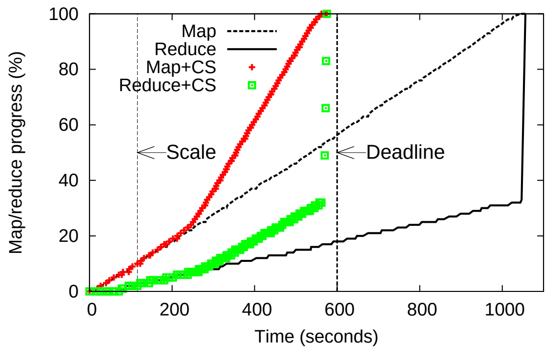


Figure 9: 10 GB Grep job progress on EC2.

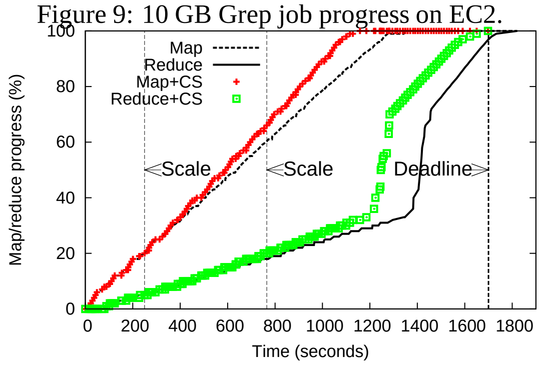


Figure 10: 10 GB Wordcount job progress on EC2.

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 9 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 10 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 as seen in Section 4.2.2. 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.

Our results from this performance optimization problem show that our building blocks, online profiling and cluster scaling, were successfully synthesized into an algorithm to enable the agility of MapReduce applications

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 3: 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.

to meet given deadlines.

4.4 Energy Optimization Evaluation

In this section we show the benefits of using our energy optimization algorithm (Section 3.2), 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.

4.4.1 Benefit of Initial Provisioning

We first show the benefit of using initial provisioning based on accurate offline profiling (our offline 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.

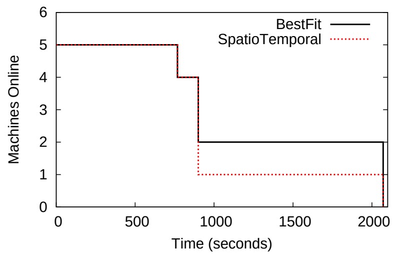


Figure 11: The spatio-temporal algorithm uses fewer machines, has a lower CMU and thus saves energy.

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 11.

4.4.2 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 3.2). The results can be seen in Table 4. 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.

5 Related Work

Resource provisioning in MapReduce. Recent work has investigated the problem of sharing resources across several MapReduce jobs while achieving different objectives [33]: Yahoo's capacity scheduler consists of different job queues, and each queue receives its capacity when it contains jobs, while unused capacity is distributed among other queues. Facebook's fairness scheduler ensures fairness among different jobs, and also provides resource guarantees for production jobs. These schedulers' main focus is to fairly allocate resources across jobs, and not on how the jobs are co-placed. Quincy [23] 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

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 4: Cluster scaling (CS) used for time balancing to achieve energy savings. TI corresponds to Time Imbalance.

can be further improved by leveraging information similar to that provided by STEAMEngine's job profiling and cluster scaling components. Further, we believe that STEAMEngine's building blocks can be leveraged in the non-virtualized MapReduce clusters considered in these cases to similarly optimize different metrics of interest. For example, the problems of placement and choice of the number of maps and reduces to optimize provisioning in this setting are analogous to the problems studied in this paper for the virtualized setting.

Sandholm et al [32] presented a resource allocation system that uses priorities to offer different service levels to jobs over time. This enables resource allocation to be varied across different job stages, and even within a job, resulting in overall performance improvement while the cost budget is met. 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 [15] 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. Also, while the above algorithms focus on optimizing specific metrics such as performance and fairness, our work presents a framework with a set of common building blocks that can be exploited by different provisioning algorithms to achieve different objectives.

Our energy minimizing resource provisioning algorithm addresses an important MapReduce issue. Recent work points to the growing concern regarding energy consumption of MapReduce; Work in [19, 29] used MapReduce as one of the workloads in evaluating the power consumption of datacenters. Further, recent work [28] 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 [14] 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 interesting work on improving the Hadoop implementation, and its applicability (e.g., [16, 39]). STEAMEngine's building blocks can be leveraged beyond resource provisioning problems. Work in [5] proposes an approach similar to our job profiling to determine optimal configuration parameters for a MapReduce job. Mantri [4] uses an approach similar to our online profiling to detect outliers in a MapReduce job, and proactively takes corrective action. As part of our future work, we would like to investigate if more detailed models [22, 30] that predict MapReduce completion time could potentially enhance the accuracy of our job profiling building block, and experiment with other applications including scientific data analysis [27] to understand enhancements required.

Resource allocation in virtualized environments. A large body of work has explored application placement in a virtualized data center to minimize energy consumption [37], perform load balancing [34, 38] or for server consolidation [25]. 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 [35] 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.

Energy Management. A number of resource allocation techniques (e.g., [10, 7, 13]) leverage the stateless nature of certain workloads (such as web requests) to end and restart request execution on a different physical machine that saves energy. Such techniques are not suitable for MapReduce jobs due to the stateful nature of MapReduce VMs. Recent work [31, 24] has explored integrating system level power management policies with virtualization technologies. They focus on developing hooks into virtualized systems to better integrate power management, and hence is complementary to our techniques.

Parallel Processing. Finally, parallel job scheduling in the context of massively parallel supercomputers is a well studied area [21, 20], 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.

6 Extending STEAMEngine

STEAMEngine is easily extensible for plugging in new building blocks and provisioning algorithms.

Other Building Blocks: While profiling and cluster scaling are useful building blocks for both cloud providers and users, there can be other building blocks that are only useful for one of the two. An example of a provider-side building block is **migration.** This building block is based on VM migration capability and can be used to migrate a MapReduce VM to another physical machine for several reasons: to improve a map task's data-locality, to load-balance, or to consolidate for energy management. However, since MapReduce applications are data-intensive, to leverage this opportunity, it is important to understand the overhead of migration, which is dependent on the storage model used by the cloud environment. For instance, the commonly used live VM migration relies on a shared storage model, while traditional MapReduce clusters use a local storage model (data on local disks of compute nodes). Capturing the migration overhead and determining the right storage model would be critical in designing this building block.

Another example is an **instance scaling** building block. While the cluster scaling building block for STEAMEngine provides the capability for users or cloud providers to *scale-out* or *scale-in* resources allocated to the MapReduce job, instance scaling could enable *scale-up* or *scale-down* of the VM instance allocated to the cluster nodes. As an example, if a VM needs more CPU, it can be dynamically allocated using virtualization capabilities like *shares* in VMware or *CPU weight* in Xen. Additionally, other techniques like DVFS [8] can be used to reduce the clock frequency of the CPU, reducing the energy consumption, thus scaling down the VM.

Other Provisioning Algorithms: In this paper, we described two provisioning algorithms—a cloud user driven performance optimization and a cloud provider driven energy optimization. There are many other provisioning optimizations possible that can leverage one or more of the same building blocks to achieve other objectives. For example, a **dynamic load balancing** algorithm may attempt to alleviate hotspots in the data center by identifying the VMs with most remaining runtime and either shutting them down on congested servers and cluster scaling them on underutilized servers or using a VM migration building block to migrate them to other servers. Similarly a **QoS-driven optimization** algorithm may obtain online estimates of job runtimes for high priority jobs and scale them to larger cluster sizes while possibly scaling down low priority jobs.

7 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 a set of common building blocks—Job Profiling and Cluster Scaling—that estimate and alter the temporal characteristics of the MapReduce job. STEAMEngine also comprises of provisioning algorithms that leverage these building blocks to optimize desired metrics. 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 6-machine Xen cluster when simultaneously executing multiple MapReduce jobs.

References

[1] Amazon EMR. http://aws.amazon.com/elasticmapreduce/.

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

[3] Amazon EC2. http://aws.amazon.com/ec2.

[4] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., and Saha, B. Reining in the outliers in map-reduce clusters. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[5] Babu, S. Towards automatic optimization of mapreduce programs. In *ACM Symposium on Cloud Computing (SOCC)* (2010).

[6] Belady, C. L. In the data center, power and cooling costs more than the it equipment it supports. *Electronics Cooling Magazine 13*, 1 (2007), 24–27.

[7] Bohrer, P., Elnozahy, E. N., Keller, T., Kistler, M., Lefurgy, C., McDowell, C., and Rajamony, R. The case for power management in web servers. *Power aware computing* (2002), 261–289.

[8] Burd, T., Pering, T., Stratakos, A., and Brodersen, R. A Dynamic Voltage-Scaled Microprocessor System. In *IEEE ISSCC* (2000).

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

[10] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing energy and server resources in hosting centers. In *Proceedings of Symposium on Operating System Principles* (2001).

[11] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001).

[12] Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., and Gautam, N. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS Conference* (New York, NY, USA, 2005), ACM, pp. 303–314.

[13] Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., and Gautam, N. Managing server energy and operational costs in hosting centers. In *ACM SIGMETRICS* (2005).

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

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

[16] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. Mapreduce online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2010).

[17] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI* (2004).

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

[19] Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)* (2007).

[20] Feitelson, D. G. Job scheduling in multiprogrammed parallel systems. *IBM Research Report RC 87657* (1997).

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

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

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

[24] Kansal, A., Liu, J., Singh, A., Nathuji, R., and Abdelzaher, T. Semantic-less coordination of power management and application performance. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower)* (2009).

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

[26] Koomey, J. G. Worldwide electricity used in data centers. *Environmental Research Letters 3*, 3 (2008).

[27] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing* (2010).

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

[29] Lim, K., Ranganathan, P., Chang, J., Patel, C., Mudge, T., and Reinhardt, S. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)* (2008).

[30] Morton, K., Friesen, A., Balazinska, M., and Grossman, D. Estimating the progress of mapreduce pipelines. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)* (2010).

[31] Nathuji, R., and Schwan, K. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of ACM SIGOPS symposium on Operating systems principles (SOSP)* (2007), pp. 265–278.

[32] Sandholm, T., and Lai, K. Mapreduce optimization using dynamic regulated prioritization. In *ACM SIGMETRICS/Performance* (2009).

[33] Scheduling in hadoop. http://www.cloudera.com/blog/tag/scheduling/.

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

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

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

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

[38] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of Symposium on Networked Systems Design and Implementation)* (2007).

[39] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *OSDI '08: Eighth USENIX Symposium on Operating System Design and Implementation* (2008).