Orchestrating the Deployment of Computations in the Cloud with Conductor

Alexander Wieder *Pramod Bhatotia* *Ansley Post*[1] *Rodrigo Rodrigues*[2]
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

When organizations move computation to the cloud, they must choose from a myriad of cloud services that can be used to outsource these jobs. The impact of this choice on price and performance is unclear, even for technical users. To further complicate this choice, factors like price fluctuations due to spot markets, or the cost of recovering from faults must also be factored in. In this paper, we present Conductor, a system that frees cloud customers from the burden of deciding which services to use when deploying MapReduce computations in the cloud. With Conductor, customers only specify goals, e.g., minimizing monetary cost or completion time, and the system automatically selects the best cloud services to use, deploys the computation according to that selection, and adapts to changing conditions at deployment time. The design of Conductor includes several novel features, such as a system to manage the deployment of cloud computations across different services, and a resource abstraction layer that provides a unified interface to these services, therefore hiding their low-level differences and simplifying the planning and deployment of the computation. We implemented Conductor and integrated it with the Hadoop framework. Our evaluation using Amazon Web Services shows that Conductor can find very subtle opportunities for cost savings while meeting deadline requirements, and that Conductor incurs a modest overhead due to planning computations and the resource abstraction layer.

1 Introduction

Cloud computing gives programmers access to instantaneous, and practically unlimited computational resources. This allows users and organizations to adapt the computational power they use according to their needs, without requiring them to invest in IT infrastructure. This ease of scalability has made cloud computing popular among end users and a subject of excitement in research and industry. Users have the opportunity to transfer computations into the cloud, enabling applications that were previously impossible or too expensive to perform locally.

These new opportunities, however, bring new challenges. In the past, organizations invested in building and maintaining an IT infrastructure. Given that investment, they could estimate how long a certain computation would take (or its feasibility). In the new cloud computing era, however, it is possible to spend an almost unbounded amount of money on computational resources. This changes the nature of the equation, since organizations can balance the monetary cost of a computation with how long it takes to complete it. Ideally, a customer could invest the exact amount that is needed to complete the required computation within the preferred deadline.

The situation is complicated by the fact that cloud computing providers offer many different services. For example, EC2 currently provides eleven different types of virtual machine instances, and it is unclear how a computation's performance will change if run on different instance types. In addition to the rental of a virtual machine, cloud providers also offer a variety of storage options, in addition to the storage available from the rented virtual machines. Finally, cloud providers charge for data transfer across different systems, particularly between the cloud and the outside world. These factors make it hard to calculate the exact cost of a cloud deployment. Furthermore, the need to account for the possibility of failures (and the cost for recovering from them) and the emergence of spot markets, which allow bidding for resources, aggravate the complexity of making best use of cloud services.

This paper presents Conductor, a system that enables cloud customers to make better decisions about which cloud services to select, and orchestrates the execution of MapReduce computations on the cloud automatically. Conductor therefore frees the customer from having to understand the trade-offs between different services, devising an optimal execution plan, and deploying that plan. For a given MapReduce computation, Conductor lets customers specify optimization goals, such as minimizing monetary cost or completion time, and leverages automated optimization techniques to determine an execution plan that best meets these goals. The system then deploys the plan by invoking the appropriate cloud services at various points in the execution and migrating data among them. Finally, at deployment time, Conductor detects deviations from the expected plan, such as those due to mispredictions of job performance or spot prices, and adapts by recomputing the plan and adjusting the deployment.

[1] Currently at Google Inc.
[2] Currently at CITI/Universidade Nova de Lisboa

The design of Conductor addresses several interesting technical challenges with novel techniques. For example, Conductor was able to cope with the heterogeneity of cloud services, which sometimes combine both a storage and a computation service under the same interface, by designing a resource abstraction layer that separates storage from computation and provides a unified interface to every service of the same class. This abstraction is important to make the planning stage feasible by only needing to consider the generic abstractions offered by the resource abstraction layer. Furthermore, the abstraction layer enables the deployment of computations without the need to worry about lower-level interface details of each specific service.

We implemented a Conductor prototype, which comprises several components: a module for determining the optimal plan for deploying MapReduce jobs, a resource abstraction layer that maps the unified computation and storage interfaces to the suite of services offered by Amazon Web Services, and an extension to the Hadoop framework that interacts with both the planning module and the resource abstraction layer. Our evaluation shows that Conductor's automatic management succeeds in finding and deploying efficient execution plans, and discovers non-trivial opportunities for cost and time savings, while incurring a modest overhead.

The remainder of the paper is organized as follows. In Section 2 we lay out today's challenges when using cloud services and state the goals of our system. In Section 3 we overview how Conductor automates the deployment of computations and in Section 4 we describe how we can formally model MapReduce computations to automatically determine optimal deployment plans. In Section 5 we describe the design and implementation of Conductor, and in Section 6 we present evaluation results with our prototype. We present an overview of related work in Section 7, and conclude the paper in Section 8.

2 Problem Statement

In this section we detail some of the challenges that cloud customers face and describe the goals of Conductor.

2.1 Challenges

The following are several examples of challenges that arise when deploying a computation in the cloud.

Service and provider diversity. Cloud customers must choose among a variety of services with different price and performance characteristics. For example, for its EC2 service alone, Amazon offers eleven different types of VM instances. Furthermore, the diversity of these offerings is increasing, as new providers emerge and existing providers introduce new services.

Hybrid deployments. A special case of provider diversity is when cloud customers make use of their own local infrastructure, which can be augmented by the use of cloud services. Local infrastructures have different characteristics from cloud services, namely that the use of a local infrastructure does not incur additional costs, but provides access to a limited amount of storage and computational power.

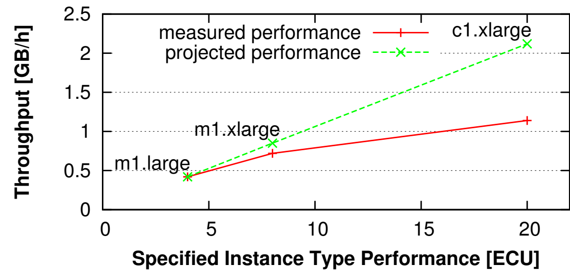


Figure 1: Specified and measured performance for three different EC2 instance types.

Dynamic pricing. The pricing for cloud services can vary over time as providers adjust their pricing models, or when new providers join the market. In addition, spot markets for cloud computing services have been recently introduced in EC2, bringing new opportunities and challenges. In particular, customers may try to use predictions of the evolution of spot prices to obtain cost savings, but may also have to adjust their choices at deployment time in case their predictions are not met.

Mispredictions. To estimate the cost of alternative deployment strategies, customers need to predict the performance characteristics of different services. This is challenging for several reasons. First, the information that is available about these characteristics can deviate from the performance that is actually observed. To illustrate this, we measured the performance of various types of EC2 instances, and compared it to the estimated performance that Amazon reported in terms of a unit they call ECU.[1] The results in Figure 1 show a consistently increasing throughput divergence between the projected and measured application performance. Second, the performance characteristics of a given cloud service can vary dramatically over time [20]. For instance, network throughput might drop due to congestion, not only within the data center, but also on the path between the user and the cloud provider. Also, since often multiple virtual machine instances are hosted on a single physical machine, a level of interference among virtual machines that is higher or lower than normal can lead to degraded or improved performance, respectively.

Faults. Cloud providers have also started to offer services with different reliability characteristics, for instance, with discounted prices for storage services with

[1] For this simple experiment, we use the same setup and application that is used in our evaluation in Section 6, and we configured Hadoop to fully exploit the parallel processing capabilities that each instance type offers.

lower replication factors. These reliability levels are of particular importance in long-running computations, or computations that store intermediate results in these storage services. An example of this are Pig [13] programs, which compile down to multi-staged MapReduce computations, in which the result of one stage is used as the input to the subsequent stage. In this case, when intermediate results become unavailable due to data loss, they must be recomputed by re-executing all previous stages. Therefore, the cost of this recovery depends on the number and complexity of the previous stages, and generally increases as the computation progresses, making more reliable storage options more and more useful [9].

Tightly coupled data and computation. A computation that is deployed on the cloud will make use of several types of resources, namely CPU, storage, and bandwidth. Even though cloud computing providers offer separate services and pricing options for some of these, like storage, most services end up tightly coupling these various categories. For instance, compute services like EC2 associate a virtual disk with each VM instance, which can be used for storage. Also, when performing a computation, one must take into account the cost of transferring the input data to where the computation is performed. This tight coupling complicates the task of deciding which services to use in several ways: it may hide opportunities for making use of resources, such as taking advantage of virtual disks to avoid having to pay for S3 storage, and it precludes simplistic resource management approaches, such as always using the cheapest offering – such a strategy could lead to increasing the overall cost or the completion time, e.g., because of the cost of transferring the data between computation and storage locations.

2.2 Goals

Our goal is to build a system that overcomes these challenges by automating the process of choosing which cloud services to make use of, and by deploying computations on the cloud according to that choice.

Aside from addressing the aforementioned challenges, the system should ideally meet the following goals.

Transparency. Customers should obtain the benefits of the system without having to modify their computations. In particular, they should be able to leverage different types of services without having to adapt their applications to the interface that is provided by that service.

Efficiency. The customer should be able specify certain goals like minimizing cost or execution time, and the system should not only find a good solution according to those metrics, but also impose low overheads both in the planning stage and also at deployment time.

Adaptivity. The system must be able to react at deployment time to mispredictions or changes in the characteristics of the deployment, so that user goals are not jeopardized by these events.

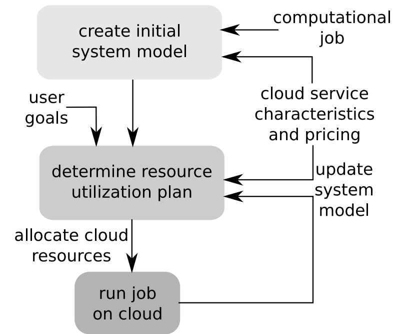


Figure 2: System overview

Flexibility. As cloud services keep evolving, it should be easy to incorporate new services or modifications to an existing service into the system, to allow customers to rapidly take advantage of them.

3 System Overview

To address the aforementioned challenges and goals, we built *Conductor*, a system that simplifies planning and deployment of jobs on the cloud by choosing which services to make use of, according to customer-defined goals, and deploying computations on the cloud according to that choice. In this section we present an overview of the design of Conductor.

As a starting point, a customer outsourcing a computation provides Conductor with the following input: (1) a computation to be executed in the cloud, (2) a set of cloud services that could be used for executing the computation, and (3) a set of goals to optimize the execution for (e.g., minimizing execution time for a given budget).

Given these inputs, Conductor starts by finding an execution plan that best meets the goals specified by the customer. Once the plan is devised, Conductor deploys and executes this plan, and, if necessary, adjusts it to changing conditions, such as variance in network performance due to congestion or degraded virtual machine instance performance due to interference with other instances on the same physical machine.

At a high level, this functionality is achieved through the following sequence of steps (as depicted in Figure 2).

1. Model both the computation and the set of services available from cloud computing providers, their cost, and their performance.
2. Automatically determine an optimal execution plan by using a solver.
3. Deploy the planned execution and monitor the execution to identify conditions that constitute possible deviations from the original model.

4. Upon detecting deviations, feed the new conditions back into the model, compute a new plan, and alter the deployment accordingly.

The next sections detail each of these steps.

4 Modeling Computations and Services

We model computations and the service offered by multiple cloud providers using dynamic linear programming. We chose to use dynamic linear programming because the pricing schemes of cloud services as well as the performance of many data-parallel distributed computations can be expressed as linear dependencies. Furthermore, powerful tools to solve linear programs are available. However, not every computation can easily be modeled using dynamic linear programming, and hence, we had to restrict the domain of our approach. Next, we explain this restriction, and, in subsequent sections, we detail how we model services, computations, and costs.

4.1 Restricting Computation Types

In order to automatically create a linear programming based model for a given job, it is necessary to know what are the individual processing phases, their computational cost, and the data flow patterns among them. Since this is difficult to predict without analyzing the computation, we restrict ourselves to a specific class of computations, namely MapReduce jobs [2]. MapReduce is very generic and has gained widespread adoption, and therefore, by focusing on this model, we cover a very broad and increasingly important set of large-scale computations. Furthermore, MapReduce computations follow a predefined data flow pattern, which makes it feasible to build a generic dynamic linear programming model. Our description assumes the reader is familiar with MapReduce; the original MapReduce paper supplies the necessary background [2].

An alternative to restricting to MapReduce would be to let programmers manually specify these job characteristics, as proposed by other work [5]. Another possibility would be to focus on recurring jobs, where the first run would be monitored to extract the model that would be used in subsequent runs. The core of our system would not have to be changed to accommodate these methods.

4.2 Modeling Cloud Service Offerings

One of the challenges in choosing the best set of services to make use of stems from the fact that each service may coalesce resources of different types, namely storage and computation, which we then need to consider separately when determining the best set of services to make use of. To address this, our model for cloud service offerings breaks down each service into the following three separate types of resources, a subset of which may be provided by that service: computation, storage, and communication. This allows us to have fine-grained control over which services to utilize according to the application needs for each of these resources.

Formally, the model for cloud service offerings considers a set of m distinct cloud services (e.g., Amazon EC2, Amazon S3), $F_1, ..., F_m$, that provide a set of resources of different types. The idea behind our model is to explicitly consider the storage and computation capability of each service, and model communication implicitly. This is because, in contrast to storage and computation, communication cannot stand on its own and be allocated independently, but instead always *connects* other resource instances, which can either be storage or computation. Each service is also associated with a certain price that the customer is charged for using it. For a service that has different prices for allocating a resource for the first time and maintaining a resource, we additionally annotate the communication resources connected to that service to reflect this pricing scheme. For instance, a cloud storage service might charge customers for storage capacity consumed in a period of time, and additionally for network traffic and I/O operations when uploading data to (or downloading from) the service. In this case, in our model the storage service is annotated with the time-based storage cost, and the per-use cost is modeled as a communication cost. Modeling the per-use cost is possible in our case since we can precisely control how data upload and download is mapped to individual I/O operations and how much data is transferred in each operation on average. Thus, given the average amount of data transferred per I/O operation, we can translate the per-operation costs to per-byte costs and incorporate them as communication costs in our model.

Conductor generates the model automatically from a description of cloud service offerings that contains information about service cost, performance characteristics and other properties. In Figure 3 we show a simplified example of a service description in a simple, human-readable XML-based format that Conductor takes as input. These descriptions of public cloud services could be published by the providers themselves or by third parties, while a user would only have to manually specify his privately owned resources (if any).

4.3 Modeling MapReduce Computations

Next, we walk through the successive steps of MapReduce computations to explain how we model them [17]. Unless mentioned otherwise, all variables in our model are positive. We express the execution as a sequence of discrete time intervals such that for each interval t the model contains the actions (e.g., process or transfer data) that can be performed in that interval. An important prac-

```
<resource>
    <property="name">
        <string> S3 </string>
    </property>
    <property="cost_get">
        <double> 1.0E-6 </double>
    </property>
    <property="cost_put">
        <double> 1.0E-5 </double>
    </property>
    <property="cost_t_store">
        <double> 2.08333332E-4 </double>
    </property>
    <property="can_compute">
        <boolean> false </boolean>
    </property>
    <property="storage_capacity">
        <int> -1 </int>
    </void>
</resource>
```

Figure 3: Simplified example of the XML-based description of the S3 storage service.

tical aspect of this model is that, to limit the size of the model that is generated, we always set an upper bound T on the time to finish the computation. T is expressed in terms of number of time intervals, which are the granularity of the execution progress. For instance, one interval could correspond to one hour of runtime.

Input to Map phase. To execute the *Map* phase, the input data from the source storage has to be uploaded to a storage service in the cloud for processing. The upload is modeled in a time-step fashion for all T intervals. For each interval t, the source storage contains $source_t$ amount of data that wasn't uploaded yet and $upload_{(i,t)}$ denotes the amount of data uploaded from the source storage to the storage service F_i. All the data uploaded by time t (denoted by $storeIn_{(i,t)}$) will be stored in F_i until the execution phase is finished. Data storage and upload is *flow preserving*, which we express by the following constraints:

$$\forall i,t: \; source_t - \sum_{i=1}^{m} upload_{(i,t)} = source_{t+1} \tag{1}$$

$$\forall i,t: \; storeIn_{(i,t-1)} + upload_{(i,t)} = storeIn_{(i,t)} \tag{2}$$

The available upload speed can be expressed in the model by adding a constraint that restricts the total amount of data that can be uploaded.

Data processing. Next, the uploaded data is processed, and the result is stored. Similar to data upload and storage, we model the actual processing per time interval t: In interval t, the uploaded data $storeIn_{(i_1,t)}$ in storage service F_{i_1} can be processed by a computational service and then the result $storeOut_{(i_2,t)}$ is stored at F_{i_2}.

The amount of data that is processed in each time interval t is bounded by the number of computing nodes that we choose to run during that interval. Also, we can only process input data in the cloud that has already been uploaded. Let $proc_{(i,t)}$ denote the amount of data which is processed by cloud service F_i in time interval t. We can therefore represent the constraints for computations as follows:

$$\forall i,t: \; \sum proc_{(i,t)} \leq nodes_{(i,t)} \cdot capacity_i \tag{3}$$

$$\forall t: \; \sum_{t'=1}^{t} \sum_{i=1}^{m} proc_{(i,t')} \leq \sum_{i=1}^{m} storeIn_{(i,t)} \tag{4}$$

Here, $nodes_{(i,t)}$ denotes the number of computing nodes rented in interval t from computing service F_i, and $capacity_i$ denotes the processing capacity of a single node for F_i.

Reduce phase. The *Reduce* phase is modeled in a similar way to the Map phase, except for the fact that we do not need to consider the data upload stage, since the *Reduce* phase takes the result of the *Map* phase as the input. Hence, in our model, in each time-step t we add possible transitions from the output storage of the *Map* phase $storeOut_{(i,t)}$ to the input storage of the *Reduce* phase $storeIn_{(i,t+1)}$.

With the current formulation, the Reduce phase can start without the Map phase being complete, which is not allowed by the MapReduce model. We enforce that the two phases do not overlap by specifying that the amount of data flowing to the next phase has to be either 0 or the full output data. We specify this property as a linear programming constraint using a *semi-continuous* variable, that can hold either 0 or the full output data size. After combining the two phases, we model the download of the final result from the output storage of the *Reduce* phase by adding transitions to the destination storage.

4.4 Execution Cost

The model must also capture the monetary cost of running the computation. The monetary cost of each phase can be expressed as the cumulative sum of the cost incurred in each interval over time T. For time interval t, the cost can be expressed as the sum of the cost incurred for uploading the data, processing the data and storing the result in a storage service. We calculate the cost for each time interval based on the amount of resources consumed per cloud service. For instance, the computation cost in time interval t is the number of machine-hours used in this interval multiplied by the price per machine-hour. Formally, we express the total monetary cost over T as follows:

Let $y_{(i,t)}$ be the number of units of cloud service F_i purchased for time interval t, and let b_i be the price per unit for F_i. The total cost C for such a configuration is

$$C = \sum_{t=1}^{T} \sum_{i=1}^{m} (b_i \cdot y_{(i,t)}) \tag{5}$$

Note that this monetary cost, as well as other characteristics captured in our model such as execution time, can be used in the objective function for optimization. Since no negative amount of resources can be purchased, we automatically have the constraint $\forall i, t: \; y_{(i,t)} \geq 0$.

4.5 Data Migration

Since we consider multiple storage services in our model, we may choose to migrate data between them during the execution. We include migration by adding transitions in each time interval t from $storeIn_{(i_1,t)}$ to $storeIn_{(i_2,t+1)}$. These transitions express migrating input data from the storage services F_{i_1} to F_{i_2}. Similarly, we add transitions for migrating the output data $storeOut$ in each time-step. Note that the transitions for data migration go from one time-step t to the next one $t+1$, rather than staying within the same time step. This allows us to express that data migration is not completed instantly. The cost for migration can be added to the storage cost per time-step.

4.6 Resource Overlap

In our previous explanation of the model, we have assumed that each service provides only a single type of resource, either storage or computation. However, in practice, services can provide both types (and potentially other types) simultaneously. For instance, we can opportunistically store data on the virtual disk of running VMs, leveraging this spare resource at a low extra cost.

Our model accommodates this overlap of resources easily, since it distinguishes cloud services from the resources they provide. Thus, in addition to the pricing and performance characteristics we already specify for each cloud service $F_1, \ldots, F_m$, we also specify the quantities of other resources $R_1, \ldots, R_n$ each of the services offers. For instance, in this model a pure storage service like Amazon's S3 will provide only storage resources while instances of Amazon's EC2 service provide both computation and storage resources.

4.7 Dynamic Pricing

Recently, Amazon started offering spot market pricing, where customers bid the maximum price they are willing to spend to have access to unused Amazon EC2 capacity, thus paying a price tag that reflects the current supply and demand. Furthermore, Amazon allows customers to have access to the history of spot prices, so that customers can try to predict how spot prices will change and develop a bidding strategy.

We thus extend our model to include dynamic pricing in spot markets. Given our model where computations are divided into discrete time-steps, spot prices can easily be incorporated by setting the price of this service in each time-step to an estimated spot price. These estimates could potentially be derived by extrapolating past pricing patterns. In our evaluation in Section 6.5 we leverage a simple method that uses the maximum spot price of the last n hours as a basis to compute a bid. More elaborate methods [1] or methods for analyzing stock market trends could also be leveraged. However, predicting spot prices is a challenging problem in its own right and beyond the scope of this work. For the sake of simplicity, we assume *some* predictor that can produce estimates for future pricing. Let $E[b_{(i,t)}]$ denote the estimated price per unit of cloud service F_i for time interval t. Thus, the modified total cost C' can be expressed as follows:

$$C' = \sum_{t=1}^{T} \sum_{i=1}^{m} (E[b_{(i,t)}] \cdot y_{(i,t)}) \qquad (6)$$

4.8 Solving

For processing the linear program and computing an optimal execution plan, we dispatch the generated linear program to the CPLEX solver. Although the solving time is usually on the order of seconds (see Section 6.6), it is possible that in certain cases CPLEX takes significantly longer to compute the optimal solution. In such cases, a potentially non-optimal, but feasible solution can be found much faster. To avoid long delays on the deployment of jobs submitted by users, instead of waiting for the optimal solution, we bound the solving time to three minutes and use the best solution that was computed so far.

5 Job Deployment

Once Conductor finds an optimal execution plan for a model of a job and available resources, it deploys the plan by instantiating the appropriate cloud services. We next present the design of this component of Conductor.

5.1 Programming Abstractions

The deployment of a computation plan is complicated by the fact that different services may have different storage and computation interfaces, incompatible semantics, or that sometimes storage and computation are bundled together. Conductor overcomes the differences between the services by providing a uniform interface to applications. In particular, Conductor provides abstraction layers for the two basic resource types: storage and computing resources. For services with bundled resources like EC2 instances, the abstraction layers for these two different resource types allow using storage and computation independently. These abstraction layers also enable Conductor to transparently manage the resources according to the execution plan. Furthermore, the abstraction layers hide the complexity of supporting and managing the resources from the application, as depicted in Figure 4.

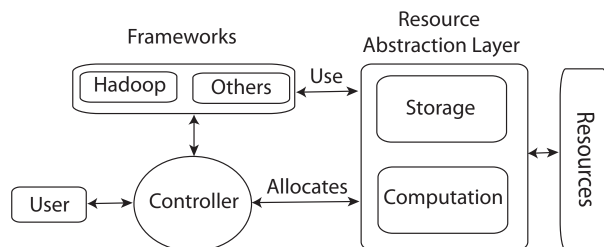


Figure 4: System overview.

Storage. The primary goal of Conductor's storage system is to provide an abstraction layer that enables applications to transparently utilize multiple different storage services as backends, and manage this usage automatically. For instance, by using Conductor's storage system, applications can transparently use both S3 and the local hard disk of virtual machines to store different parts of the data. This storage can be accessed by a client that hides from the user how and where data is stored on the backends.

We implemented Conductor's storage system as a distributed key-value storage service. The key-value interface is generic enough to support other abstractions built on top of it. For instance, there are already multiple file system implementations built on key-value stores [12]. Also, many applications and frameworks, including Hadoop, support Amazon's S3 storage service. Since S3 and Conductor's storage system provide similar interfaces, Hadoop is able to use Conductor's storage system seamlessly.

The central component in Conductor's storage system is the *namenode*, which provides a directory service for data, and manages upload, replication and migration of the data as per the execution plan generated by the controller (described in Section 5.2). The namenode maintains a mapping from file block identifiers to their locations in Conductor's storage system. These location records contain information specific to the storage backend on which a file block is stored. For instance, for a file block stored on a node's local hard disk, the location record would indicate the type of storage and include the addresses of the nodes storing that file block. (We replicate blocks in more than one node for fault tolerance and performance.)

The implementation of each storage backend is specific to the storage services it utilizes, and maps the semantics of each service to the target key-value store semantics. For instance, the local disk storage backend uses a storage daemon running on each participating node. This daemon uses Berkeley DB to store key-value pairs locally on disk. The data stored on each node can be accessed through a protocol with put, get and delete queries. For the S3 backend, in contrast, the client uses the generic S3 client APIs.

To access data in Conductor's storage service through the uniform interface provided by the storage client, the client first queries the namenode for a set of locations for the data block. In case the namenode returns multiple locations, the client fetches the file block from the closest location (by ping time) using the logic that is specific to the storage backend given in the location record.

As an optimization, when computation and storage components are co-located on the same node, Conductor allows the computation to make use of local storage without going through the namenode. For reading data, this is done by directing requests to the local storage daemon directly, which can either succeed and proceed in a very fast manner, or fail and fall back to the normal read operation, in which case we additionally install a cached copy of the data on the local node. For writing data, the optimized write is always performed locally, and then the namenode is notified so that it can transfer data to the appropriate nodes in the background.

Computation. For computation it is more difficult to create an abstraction layer for different services. For instance, it is easy to provide a MapReduce computation service (like Amazon's Elastic MapReduce service) on top of a virtual machine abstraction, but the converse is not true. Since we restrict ourselves to MapReduce computations, our abstraction of a computation resource is an instance that is capable of participating in a MapReduce computation. In particular, we only require that computation resources can be configured and automatically set up to join a Hadoop cluster and participate in MapReduce job execution. This allows us to implement this abstraction on top of different types of services, from low level VM rental to local cluster resources. For instance, in the case of Amazon's EC2, this is achieved by building a pre-configured machine image which is used by Conductor to automatically allocate EC2 instances according to the deployment plan and have them join a Hadoop cluster.

5.2 Job Controller

The job controller is a central component of the design of Conductor. It orchestrates the job execution by generating a plan to best meet the user's goals and deploying it using cloud services.

After submitting a MapReduce job to Hadoop, a user starts Conductor's job controller to manage the execution of the job. The job controller automatically generates a linear programming based execution model with the given job characteristics and resources available as described in Section 4. The model, together with the user's optimization goals, is then processed by a solver to generate a deployment plan. After the controller receives the resulting plan, it deploys it accordingly and monitors the execution progress. Deployment decisions concerning how to handle and store data (e.g., where and

when to upload and store what data) are forwarded to the storage service (described in Section 5.1), which then triggers upload and replication accordingly. Deployment decisions about the actual processing are handled by the job controller by allocating the planned number of nodes through a service-specific interface (e.g., in case of EC2, Amazon's AWS client library) and setting them up to join the computing cluster.

In order to allow a seamless interaction between Hadoop and Conductor, we extended Hadoop in several ways, as we explain next.

5.3 Hadoop Extensions

We adapted Hadoop version 0.20.2 to support Conductor's automatic management functionalities.

Location-aware scheduler. The original Hadoop scheduler makes decisions that may conflict with the execution plans determined by Conductor, thus resulting in higher cost or performance degradation. In particular, the Hadoop scheduler tries to schedule tasks on the nodes that also hold the respective input data block, and, in cases where locality cannot be exploited, it schedules tasks on non-local nodes and reads their input over the network. This flexible scheduling of tasks conflicts with Conductor, as it may violate the deployment of the execution plan generated by the controller. For example, fetching the input data from a remote site when not specified in the plan could congest network links, hinder other data transfers, or result in transfer costs that were not considered during the optimization phase. Therefore, to accurately deploy an execution plan, we must override the flexible scheduling policies of the Hadoop scheduler. In particular, we ensure by data migration and replication that data is always locally available to the task or stored on a remote location specified in the plan when it is assigned for execution by the scheduler.

The modified scheduler is integrated into Hadoop and we normally run it on a node under the customer's control. To deploy the plan accurately, the scheduler maintains task queues for each computing resource (e.g., EC2) containing the tasks that are runnable for that resource. The scheduler sets tasks runnable when their input data is either stored locally to that resource or on a different storage resource specified in the plan. For instance, depending on the plan, a task is set runnable on EC2 nodes when its input data finishes uploading to EC2 nodes or to the S3 storage service. The scheduler then assigns runnable tasks from the corresponding queues to nodes. This mechanism ensures that during scheduling no actions are performed that were not considered in the plan, which might have negative impact on runtime or cost.

Storage system. The second extension to Hadoop is to add support for Conductor's storage system. This support is required for Hadoop jobs to process input data stored on Conductor's storage system and write output data to it. Hadoop supports multiple storage options via *file system drivers* that implement a file system abstraction. In order to make Conductor's storage system usable by Hadoop, we implemented a file system driver that translates file system specific calls (e.g., open, close, read, write) into the key-value store operations (e.g., get, put, delete) that are supported by Conductor's storage system.

In our implementation, we split files into smaller *chunks* that are stored as key-value pairs in Conductor's storage system. Additionally, for each file we store *inodes* that list the chunks that constitute the file content. Our implementation reuses to a large extent the Amazon S3 file system driver, since S3 has a similar key-value storage interface. In contrast to the S3 driver, which does not allow any locality in scheduling tasks, the driver for Conductor's storage system implements the functionality required by the Hadoop scheduler to perform location-aware scheduling. More precisely, we provide methods for the scheduler to retrieve the location of a task's input data, and, based on that information, set it to runnable. The driver also interacts with Conductor's storage system to provide hints about which data block should be uploaded or replicated with higher priority.

5.4 Adapting to Dynamics

The job controller monitors the execution progress. If the observed performance for a particular resource (e.g., EC2 instances) significantly deviates from the expected characteristics upon which the model and the deployment plan was based, the job controller adapts the deployment by creating an updated model, recomputing the plan, and deploying it accordingly.

In a similar way, Conductor reacts to other system dynamics that might change during runtime, such as dynamic pricing for resources in spot markets. Conductor re-creates a model based on the current system state and the properties of the resources, including the changed ones. Similarly to the initial model, this model is transferred to the solver daemon to determine an execution plan and deploy it.

6 Evaluation

In this section we evaluate our Conductor prototype by using it to deploy several computations on the cloud using Amazon's Web Services (AWS) in scenarios that can be difficult to handle manually or require non-obvious deployment strategies.

Our evaluation tries to answer the following main questions: (1) Can Conductor realize potential cost and time savings when deploying MapReduce jobs, both in cloud-only and hybrid deployments? (2) Can Conductor

adapt to unexpected conditions at deployment time, including the unpredictability of spot market prices? (3) What are the overheads introduced by Conductor?

6.1 Experimental Setup

In all experiments, the plan generated by Conductor exclusively makes use of large instances (`m1.large`) for processing on Amazon's Elastic Compute Cloud (EC2). These instances are equipped with 7.5GB of memory, a 850GB virtual hard disk, and 4 EC2 Compute Units, where one Compute Unit is equivalent to a 1.0-1.2GHz AMD Opteron or 2007 Intel Xeon CPU. In addition to the large EC2 instances, we also allow Conductor to use extra large EC2 instances (`m1.xlarge`). However, in the scenarios we consider, the extra large instances are never chosen in the generated deployment plans since they offer a cost-performance ratio that is slightly worse than for large instances. For hybrid deployments, we additionally use a local cluster of five machines, each equipped with an AMD Athlon64 dual core CPU running at 2GHz and 2GB of memory. Additionally, some experiments make use of S3 for storage.

The application we use for our evaluation is a k-means clustering analysis. We use the k-means clustering implementation in MapReduce that is available as part of the Apache Mahout package. The input to the job consists of 40 million randomly generated points, summing up to 32GB of data. Additionally, we use a set of 10 thousand reference points for the clustering process. For this application, the large EC2 instances we used and our local cluster nodes both achieved an average processing throughput of 0.44GB/h per node. Our approach can be applied to other applications and resources as well when their characteristics are specified.

Unless mentioned otherwise, the network bandwidth between the customer and the cloud is set to 16MBit/s (2MB/s) and the client has a predetermined deadline for job completion of 6 hours. In all experiments, the input data and the Hadoop Jobtracker were located on a node in our local cluster, and the output was also downloaded to our local cluster. We used the prices of Amazon's AWS as of July 2011. For tracking the cost of cloud resource usage in each experiment, we instrumented our prototype implementation to account for all operations over cloud resources. We chose this internal accounting approach over Amazon's accounting because it enabled us to track the per experiment cost and at a much finer granularity.

6.2 Savings in Public Clouds

First we evaluate Conductor's ability to deploy an execution plan that realizes potential cost and time savings in a scenario where the customer deploys a computation entirely on the cloud.

In this scenario, the customer has several options for deploying the computation using AWS, which we test in our experiments:

- *Hadoop S3.* Upload data to S3 and then instruct a Hadoop cluster running on EC2 instances to access data directly from S3.
- *Hadoop upload first.* Upload data directly to single EC2 instance running HDFS. Upon completion, start more EC2 instances to join the cluster and use HDFS for inputs and outputs.
- *Hadoop direct.* Set up the HDFS cluster on the client side, and instruct the EC2 instances to read and write to this HDFS cluster.

Interestingly, all of these options are described in the Hadoop or AWS documentation, which further strengthens our motivation that there often does not exist a clear choice of how to deploy computations in the cloud.

Figure 5 shows the monetary costs for different deployment options and Figure 6 shows their overall job completion time. The first four bars compare Conductor's cost and performance to the three deployment options listed before. In this case, Conductor determined it should only use EC2 instances for storage, and the adequate number of EC2 instances to be 16. Therefore, in the *Hadoop direct* run, we also use 16 EC2 instances. For the two configurations with a distinct upload phase before processing, we use 100 EC2 instances. In the runtime comparison in Figure 6, *streamed processing* denotes the combined time required for processing the data and retrieving it from the respective storage as it is consumed, without a distinct upload phase.

From these results we make two main observations. First, that the total cost and completion time can vary significantly between the different deployment options. For certain deployment options (e.g., *Hadoop S3*) the service pricing models can result in an unexpectedly high total cost. In the case of *Hadoop S3*, the actual processing was finished in little more than one hour, but two full hours are charged for each allocated instance, resulting in a total cost roughly two times higher than for the other options. The second observation we make is that Conductor succeeded both in obtaining a cost that is very close to the cheapest alternative, and in meeting the required completion deadline. Note that the fact that Conductor only performs slightly worse than the fastest alternative is a positive outcome, given that we are comparing the performance of our implementation mostly driven by a single graduate student with Hadoop's highly optimized production code. We analyze the main overheads introduced by Conductor later in the evaluation.

Another advantage of Conductor is that it not only helps determine the best deployment scheme but also helps choose the right deployment parameters, which may be even more difficult to set than just determining, e.g., whether to use S3 or not. To show this, we validate

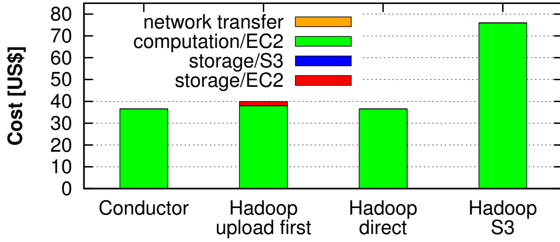


Figure 5: Monetary cost for running the job for various deployment options solely in the cloud.

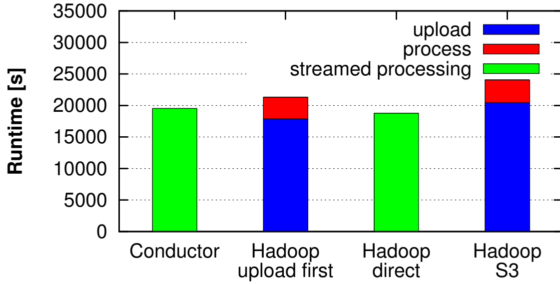


Figure 6: Job completion time of various deployment options solely in the cloud.

whether Conductor made a good decision regarding the number of EC2 instances to reserve. Therefore, we reran the experiment with five more and five less EC2 instances than those chosen by Conductor (11 and 21 instances, respectively). The results are shown in Figure 7. These show that slightly increasing and slightly decreasing the number of EC2 instances that are allocated leads to either a cost increase or missing the deadline, respectively. This validates our points that understanding the characteristics of all possible deployment options and making the right choice for a particular application scenario can be challenging, and that Conductor performs this choice automatically while incurring in modest overhead.

While this first experiment already shows some challenges in deciding which cloud services to use, Conductor still ends up resorting to one of the three deployment possibilities we had considered initially. In the next experiment we intentionally designed an even more challenging scenario where multiple different services have to be used at different times to minimize the monetary cost for the execution. To highlight this, we slightly modify the job parameters to use an upload bandwidth of 8MBit/s and a smaller set of reference points such that large EC2 instances process the input at a rate of 6.2GB/h per node.

The experimental results in Figure 8 show that neither storage option yields optimal results when used alone. Instead, the minimal cost is achieved when a mix of S3 and EC2 storage is used for storing different parts of the data at different points in the execution. In this particular example, Conductor first uploads roughly half of the in-

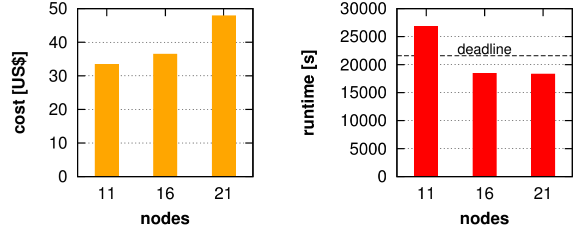


Figure 7: Monetary cost and runtime impact of deviating from the optimal deployment scheme in a cloud-only scenario.

put data to the S3 storage service, and later the remaining data is uploaded to EC2. Once an EC2 node is allocated, it starts processing the input data that was previously uploaded to S3 and the data is uploaded in parallel to an EC2 node. This utilization plan, in contrast to using only S3 or EC2 for storage, makes the best use of the EC2 nodes and the S3 service to achieve a lower monetary cost. This non-obvious resource utilization plan, which would be difficult to determine manually, is found and deployed automatically by Conductor, thanks to both the modelling and optimization phase and the use of resource abstraction layers that allow for seamlessly using the two types of storage in combination.

While Figure 8 only shows modest cost savings when compared to one of the simpler options (storing all data on S3), we point out that the cost savings for a combined solution can be much higher, since (1) the percentage gains we illustrate in the experiment increase with the input data size, and (2) these effects are also sensitive to variations in the pricing structure. Since we did not consider larger data sizes in our experiments due to financial and experiment duration constraints, we determine the potential cost savings analytically, by assuming a different input size and pricing structure. The analytic results in Figure 9 show what happens when we assume S3 storage costs are ten times higher and scale up the input size to 64, 128, and 256 GB. These results show that hitting the sweet spot for utilizing different cloud services has an increasing impact on monetary cost as data size increases, reaching savings of about $1/3$ of the cost for an input of 256 GB.

Note that the optimal fraction of data to store on EC2 when considering 32GB of input data is higher than the optimal fraction when considering larger input data sizes. This effect results from the accounting granularity of EC2 instances: since allocated node-hours are rounded up for billing, Conductor does not immediately shut down allocated instances after the computation is finished, since they are billed until the next full hour anyway. Instead, these instances are used for storage. For larger input data sizes, these rounding effects have much less impact and instances are mostly used for computation and storage simultaneously.

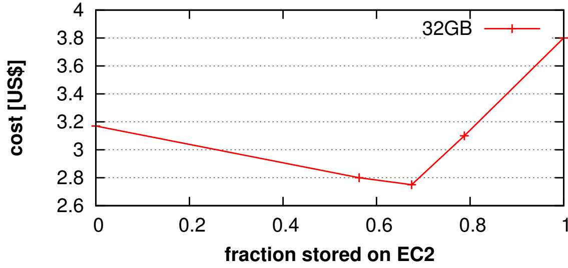


Figure 8: Total job cost depending on where the 32GB of input data is stored. A fraction of 1 (0) stored on EC2 denotes that all input data is stored on the virtual hard disks of EC2 nodes (on the S3 storage service). Conductor determined that in this scenario costs are minimized when roughly two thirds of the data is stored on EC2.

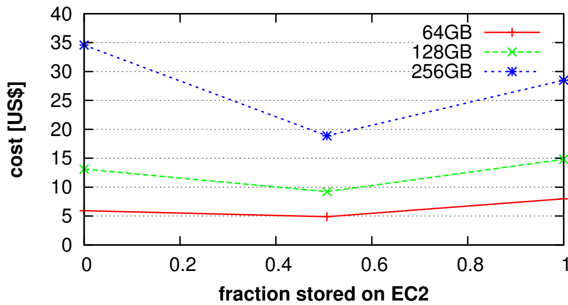


Figure 9: Total job cost depending on where the input data is stored. A fraction of 1 (0) stored on EC2 denotes that all input data is stored on the virtual hard disks of EC2 nodes (on the S3 storage service). Conductor determined that in this scenario costs are minimized when roughly 50% of the data is stored on EC2.

6.3 Savings in Hybrid Clouds

The next experiment determines whether Conductor can realize potential cost and time savings in a scenario where the cloud customer can make use of a local cluster for some of the processing, but this capacity is not enough to meet the prescribed deadline. This local cluster is modeled as just another provider (which is the user himself) that offers a single instance type (which is the machine type in the local cluster). To account for the limited size of the local cluster, we enforce a constraint in our model that limits the number of instances that can be rented.

In this scenario, Conductor determined that data should be stored on EC2 instances, and decided that the right number of EC2 instances to allocate was 16 to meet the deadline of 4 hours. In Figure 10 we show a cost and runtime comparison with an HDFS-based deployment that also allocated 16 instances. The results show that, even if the user managed to guess the right number of instances to allocate, the results that are obtained are very similar to the ones achieved by Conductor. Furthermore, Figure 11 shows what happens if the user under-

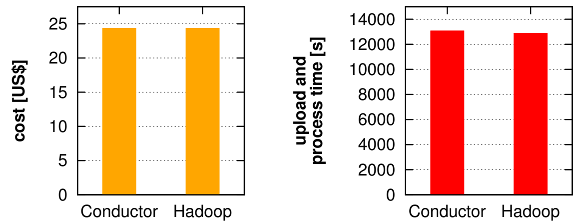


Figure 10: Monetary cost and runtime for running the job with Conductor and Hadoop when leveraging local resources.

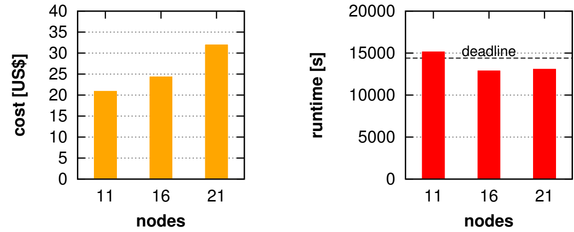


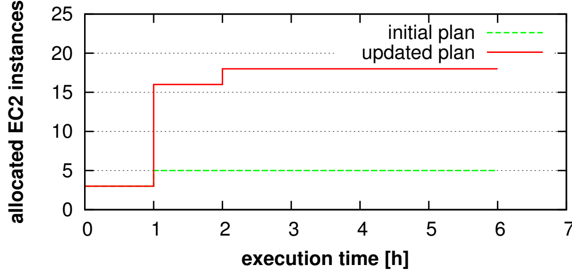
Figure 11: Monetary cost and runtime impact of deviating from the optimal deployment scheme in a hybrid scenario.

estimates or overestimates the number of EC2 instances to allocate. Again, this could lead to either an increased cost, or to missing the deadline.

6.4 Adapting to Performance Variations

In this section we present our experimental results to demonstrate how Conductor can adapt to dynamics during application runs.

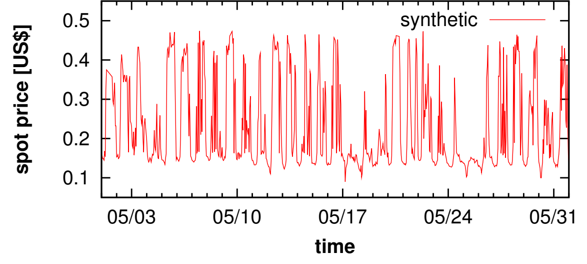
In this experiment we wrongly assume a processing speed of 1.44GB/h per node when the actual speed is 0.44GB/h. Such a difference between predicted and actual processing rates may result from wrong estimates by the user, but also due to the heterogeneity in cloud node performance [20]. Figure 12 plots the number of allocated EC2 instances and total completed tasks throughout the job execution. In the initial deployment plan, Conductor used 3 EC2 nodes in the first hour of execution and 5 EC2 nodes from the second hour on. This number of nodes would be sufficient to finish the job if the processing speed per node would indeed be 1.44GB/h. After one hour, the job progress monitoring revealed the misprediction, which caused Conductor to update the model and recompute the deployment plan. The new plan is unchanged for the first hour (corresponding to the past execution) but uses 16 EC2 nodes in the second hour and then 18 EC2 nodes from the third hour on. With this updated deployment plan, the job can be finished before the deadline even though the initial plan would have led to missing that deadline. Similarly to performance over-estimation, Conductor can react to underestimation by adapting the deployment and reducing the number of EC2 instances to use.



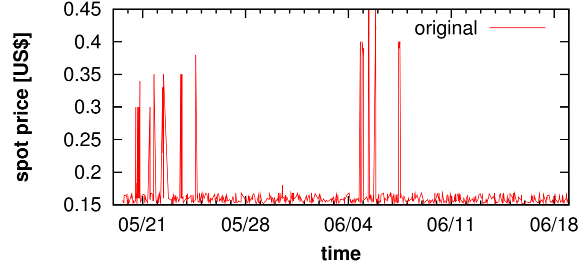
(a) Instance allocation in initial and updated deployment plan.

(b) Job progress in initial and updated deployment.

Figure 12: Job progress and Node allocation with initial and updated deployment plan.



(a) Synthetic history generated from electricity spot market.



(b) Original history for AWS EC2 instances.

Figure 13: Spot price histories for AWS EC2 instances of type m1.large

6.5 Adapting to Dynamic Pricing

Next, we evaluate through simulations the monetary savings from integrating Conductor with spot markets.

To drive spot market prices in these experiments we initially intended to use only the EC2 spot price history from AWS. However, we realized that the history of these spot prices did not exhibit any diurnal patterns, as can be seen in Figure 13. As more providers enter the market and spot markets attract more customers, we expect the price to more closely reflect data-center utilization and resource demand, and hence become more predictable. Therefore, we decided to include a second data set in our evaluation for which history can be used as a reasonable predictor. For that purpose, we use the spot price history from an electricity market. This data had to be slightly adapted, namely to make values non-negative (electricity spot prices can be negative), and also to keep the values below the normal price of EC2 instances.

In Figure 14 we present our simulation results for cost savings with Conductor when using spot instances in different settings. In *regular*, only regular instances (without dynamic pricing) are used. In *aws* we use the original spot price history for Amazon's EC2 instances, while in *el* we use the synthetic history generated from electricity prices. *-opt* denotes the cost in an optimal deployment case where Conductor can exactly predict spot prices. In the *-pX* settings Conductor uses a simple predictor that uses the past X days of spot pricing history to derive a price prediction. With *-p0*, the predictor assumes the current spot price will not change.

A first observation from these results is that allocating EC2 instances on the spot market can reduce the total job cost compared to allocating regular instances. The average cost savings in both settings range between 50% to 60% – a significant reduction.

A second observation is that the use of a trivial predictor (*p0*) is highly effective in both spot markets, achieving close to optimal cost savings. As the predictor becomes more sophisticated by incorporating more information from the recent past, there are slight improvements when using the electricity prices, namely in reducing the standard deviation of the final cost. Thus in this case the planner can efficiently leverage historic spot data. However, when considering the less regular AWS trace of spot prices, the use of more information from the recent past causes the total price and the standard deviation to go up. This is because there end up existing more situations where the plan decides to wait for a better spot price and at deployment time ends up waiting in vain.

6.6 Overheads

Storage layer performance. We compare the performance of the storage layer offered by Conductor to other storage options in Hadoop, namely HDFS and Amazon's S3. We measure throughput in our experiments, since this is the most important performance metric for MapReduce workloads. We do so by copying 32GB of data (consisting of 64MB files) to the corresponding storage service. To allow for a fair comparison to S3, we ran the measurements on large EC2 instances, where the network bandwidth to S3 is higher, instead of on our cluster nodes.

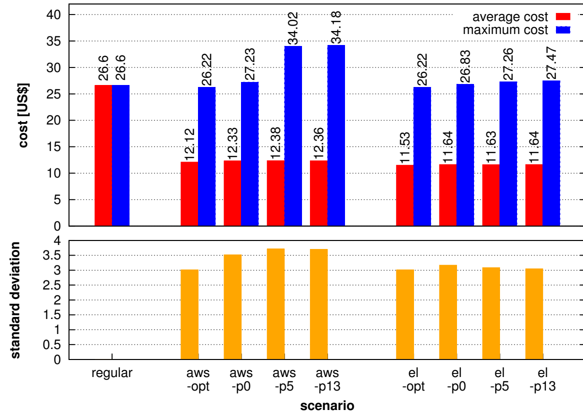


Figure 14: Average total job cost in different simulations.

Conductor's storage service and HDFS were configured with a replication factor of three, and all four nodes were either used as datanodes for HDFS or ran a storage slave for Conductor's storage. In all cases, the data was read from an Elastic Block Store volume. We considered two options for copying the data onto S3: using the integrated S3 support of Hadoop and using `s3cmd`, and a dedicated S3 command line client. The reason for also considering a separate S3 client is that we found a significant performance gap between these two options that cannot be attributed to the S3 storage service, but rather to client implementation specifics: the S3 client integrated into our version of Hadoop used SSL data transfer to S3 by default, which significantly decreased performance.

The results presented in Figure 15 show that the highest throughput was achieved with Hadoop's HDFS. Conductor's storage service exhibits roughly 25% lower throughput than HDFS in our experiments and performs comparably to S3 when using `s3cmd`. The measured throughput of S3 when using Hadoop directly is significantly lower than using `s3cmd`.

The high performance of HDFS was not particularly surprising to us, as HDFS has been actively developed for several years and significant effort has been put into performance optimization. In contrast, in our prototype the main focus was an abstraction layer that could utilize several other storage services. Therefore we believe we introduce an acceptable throughput overhead.

Modelling and Solving. Creating a linear program-based model consisting of all possible actions that could be taken for a MapReduce program and determining an optimal plan are presumably expensive operations. However, in our prototype it turned out that the model creation is very cheap. For all scenarios we considered in our evaluation, the model creation took less than 1 second on a desktop machine with an Intel Core 2 Duo CPU at 3GHz and 4GB RAM. Computing an optimal execution plan from such a model was significantly more expensive. In our experiments, we used the CPLEX 11.2.1 optimizer for that purpose. We ran CPLEX on a node equipped with 8GB of RAM and an AMD Opteron dual core CPU running at 2.6GHz. We configured CPLEX to terminate when a solution is found that is at most 1% off the optimal solution.

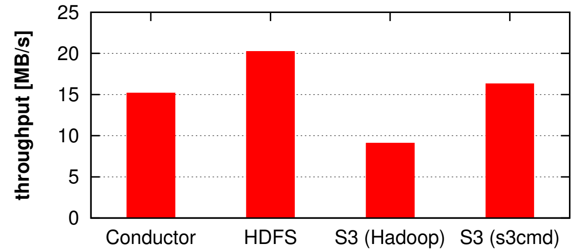


Figure 15: Performance comparison of different storage options.

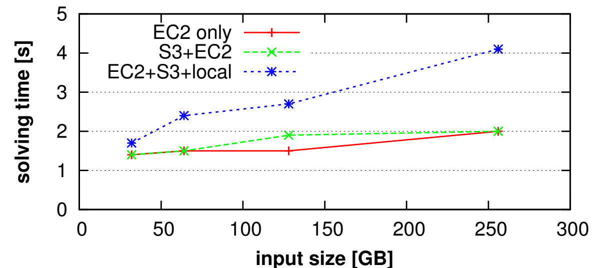


Figure 16: Model solving time for different input sizes and available resources.

Figure 16 depicts the solving time for various input sizes and different resources that are available to run the job. EC2-only means that we assume to have only EC2 available for both computation and storage. In S3+EC2 we also allow the use of the S3 storage service and in EC2+S3+local we can additionally use a local cluster for storage and computation. The results show that the solving times are acceptable, and that adding more features to the model roughly doubles the solving time. Furthermore, we can see that the input data size directly influences solving time. This is because the input size has an impact on the model size (since the input size, together with processing speed and upload speed, gives a lower bound on execution steps to include in the model), which then implies an increase on solving time.

7 Related Work

Our work builds on contributions from several different areas, which we briefly survey.

Resource Management. Automatic resource management has been studied in other contexts. For example, operating systems automatically allocate resources; cluster resource management systems have been proposed [3, 7]; and resource management in Grid computing has also been studied [10, 15]. In cloud computing,

resource management presents new challenges that are not addressed by previous work. Namely, as opposed to clusters and grids, a cloud computation has access to infinite resources but must pay an increased marginal cost for each resource it uses. Our system takes these distinguishing features into account, and modeling them is one of the primary challenges of this work. A recently proposed system called Mesos [6] allows different cluster computing frameworks to share a static commodity cluster to improve utilization and reduce data redundancy. Mesos employs a scheduling mechanism in which resources are offered to the different frameworks, and each framework can decide which resources to use and how to use them. Besides targeting a dynamic cloud setting, Conductor differs from Mesos by considering job deployment at task-level granularity, taking into account user-provided optimization goals and the costs of the on-demand cloud resources.

Scheduling. There is a significant amount of research in scheduling jobs in the context of distributed execution frameworks. Quincy [7] is a fair scheduler for scheduling concurrent distributed jobs with fine-grain resource sharing for Dryad. Late scheduler [20] overcomes the performance degradation due to straggler tasks in Hadoop for heterogeneous environments. Delay scheduling [19] strikes a balance between data locality and fairness by employing a lazy approach for scheduling jobs to maximize the locality. The most fundamental comparison point is that Conductor must consider the dynamic allocation of cloud resources as an additional dimension in the scheduling problem, while the aforementioned work can simply assume a static set of resources.

Optimization. Many systems require decisions to be made at runtime from a large set of possible alternatives. Therefore, optimization techniques have been employed to select the best choice dynamically. For example, Rhizoma [18] proposed automating resource allocation for generic applications. Rhizoma uses a specification of resources and maximizes the utility for an application. Although Rhizoma uses cloud computing as motivation, the application they describe is a publish-subscribe system deployed on PlanetLab, where the challenge is to find well-connected, lightly loaded nodes. Unlike Rhizoma, our proposal is geared towards deploying computations on the cloud, without requiring the specification of application resource requirements. We model the problem of cloud resource allocation as a linear program. Modeling other problems in such a way has been done in a variety of fields including systems research [8]. Similarly, a recent proposal [14] seeks to shift computations among multiple data centers based on changing electricity prices in the spot market. Conductor's approach is related, but addresses a different problem of making the best possible use of a diverse set of cloud resources for a specific type of computation, for which a runtime model can be derived automatically. Our own short position paper [16] described high level ideas, which are incorporated in this work, but did not present a complete system.

Hybrid deployment and spot markets. With the advent of public and private cloud infrastructures, there is a demand to utilize both of them. CloudWard Bound [5] makes a case for the hybrid deployment of multi-tier enterprise applications where the infrastructure is partly hosted on-premise, and partly in the cloud. For a given application, CloudWard Bound can suggest deployment plans that leverage cloud resources for some application components, obey user-provided privacy policies and satisfy application latency requirements. In contrast, Conductor focuses on a distributed bulk data-processing framework where it can manage the deployment of processing tasks for individual jobs. Conductor's deployment plans need to consider the varying resource requirements jobs can have throughout their execution, which is less relevant in the context of long-term deployments of enterprise applications that CloudWard Bound targets. Furthermore, data privacy is not the focus of Conductor.

Dynamic allocation of spot instances for MapReduce computations has also been proposed recently [1, 11]. In contrast, we focus on the broader problem of trying to incorporate multiple providers of potentially diverse resources (both from regular and spot markets) to determine a globally optimal resource allocation plan.

Resource exploration. The availability of multiple machine types raises the question of how the different machine characteristics will impact application performance. Accurately predicting application performance when low-level characteristics are known is a challenging problem that has been studied in the past [4]. We consider the problem of resource exploration to be complementary to our work; our approach could directly benefit from resource exploration techniques since we can leverage them to automatically predict the performance characteristics of different resource types.

8 Conclusion

In this paper we motivated and presented the design of Conductor, a system that assists cloud customers in choosing the right set of resources to use when running cloud computations. Conductor takes the burden of manual choice and optimization away from the customer, by automating the selection process and providing mechanisms to utilize different services seamlessly. This automation and flexibility allows the customer to state high level goals about price or performance, rather than having to make low level resource selection decisions.

Conductor requires users to specify simple high level goals, and a small amount of information about the MapReduce computation, and then uses optimization

tools to determine an execution plan. This execution plan is deployed, and then adapted, if any of the information used in computing the plan changes at runtime. Our evaluation shows that Conductor is able to find and deploy non-obvious execution plans, while incurring only a modest overhead.

Conductor is an important first step in automating cloud resource selection, but much work remains in generalizing the set of supported applications, increasing the adaptivity to changing conditions, and providing powerful abstractions for computation that cover a wide range of services.

References

[1] Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., and Krintz, C. See Spot Run: Using spot instances for MapReduce workflows. In *2nd USENIX Workshop on Hot Topics in Cloud Computing* (2010).

[2] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association.

[3] Dusseau, A. C., Arpaci, R. H., and Culler, D. E. Effective distributed scheduling of parallel workloads. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (1996).

[4] Ganapathi, A. S. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[5] Hajjat, M., Sun, X., Sung, Y.-W. E., Maltz, D., Rao, S., Sripanidkulchai, K., and Tawarmalani, M. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM.

[6] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association.

[7] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009).

[8] Keeton, K., Kelly, T., Merchant, A., Santos, C., Wiener, J., Zhu, X., and Beyer, D. Don't settle for less than the best: Use optimization to make decisions. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (2007).

[9] Ko, S. Y., Hoque, I., Cho, B., and Gupta, I. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM.

[10] Krawczyk, S., and Bubendorfer, K. Grid resource allocation: allocation mechanisms and utilisation patterns. In *AusGrid '08: Proceedings of the sixth Australasian workshop on Grid computing and e-research* (Darlinghurst, Australia, Australia, 2008), Australian Computer Society, Inc.

[11] Liu, H. Cutting mapreduce cost with spot market. In *3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011).

[12] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. Ivy: a read/write peer-to-peer file system. *SIGOPS Operating Systems Review 36*, SI (2002).

[13] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008).

[14] Qureshi, A., Weber, R., Balakrishnan, H., Guttag, J., and Maggs, B. Cutting the electric bill for internet-scale systems. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (2009).

[15] Raman, R., Livny, M., and Solomon, M. Matchmaking: Distributed resource management for high throughput computing. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1998), IEEE Computer Society, p. 140.

[16] Wieder, A., Bhatotia, P., Post, A., and Rodrigues, R. Conductor: orchestrating the clouds. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS 2010)*.

[17] Wieder, A., Bhatotia, P., Post, A., and Rodrigues, R. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *PODC '10: Proceedings of the 29th ACM symposium on Principles of distributed computing* (2010).

[18] Yin, Q., Schüpbach, A., Cappos, J., Baumann, A., and Roscoe, T. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (2009).

[19] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM.

[20] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association.