Cümülön-D: Data Analytics in a Dynamic Spot Market*

Botong Huang
Microsoft
bothua@microsoft.com

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

We present a system called Cümülön-D for matrix-based data analysis in a *spot market* of a public cloud. Prices in such markets fluctuate over time: while users can acquire machines usually at a very low bid price, the cloud can terminate these machines as soon as the market price exceeds their bid price. The distinguishing features of Cümülön-D include its continuous, proactive adaptation to the changing market, and its ability to quantify and control the monetary risk involved in paying for a workflow execution. We solve the dynamic optimization problem in a principled manner with a Markov decision process, and account for practical details that are often ignored previously but nonetheless important to performance. We evaluate Cümülön-D's effectiveness and advantages over previous approaches with experiments on Amazon EC2.

1 Introduction

There has been growing interest in the so-called *spot markets* in public clouds. Contrary to the standard way of paying for the use of computing resources at a fixed price, users bid in a spot market by the specifying the type and number of machines as well as the maximum unit price (per machine-hour) they are willing to pay. Driven by supply and demand, the cloud provider adjusts the market prices in real time. Machines in use are charged at their market price, which usually remains low; however, if the market price exceeds their bid price (specified when they were acquired), they are terminated immediately, and any ongoing work will be lost.

This paper studies how best to use a spot market for data analytics. We consider this problem from the perspective of an individual use: the user specifies a workflow and a deadline, and we want to find the "best" strategy for provisioning the cluster and executing the given workflow by the deadline. Because future market prices are uncertain, we do not know how much the execution will cost exactly. Furthermore, many users, especially those who do not use the spot market on a regular basis, tend to be very risk-averse on individual executions—what if they miss the deadline or end up spending far more than using non-spot machines with fixed prices? Therefore, the "best" strategy should minimize some combination of the expected cost and risk, where the trade-off between the two can be set according to user preference.

We postulate that the best way to cope with a dynamic, uncertain spot market is with continuous, *proactive* optimization and adaptation during execution. A simpler *reactive* approach could be to leave the execution alone and not act until we lose the machines in our cluster because the market price has risen higher than the bid price. But there are missed opportunities. If the current market price is very low, shouldn't we bid for more machines and add them to our cluster? Wouldn't it also make sense to release some machines in our cluster that were acquired at higher bid prices? On the other hand, if the price is tending up, shouldn't we proactively shut down our cluster and wait for the price to come down?

While one can act on intuition, the consequence of an action often involves intricate trade-offs. First, the best action depends on the execution progress. For example, if we are close to the deadline and there is much work left, we should act conservatively—it will be risky to shut down the cluster and wait for the price to drop, because we might be forced to pay high for a large cluster when the deadline is imminent. Second, the best action may depend on the time of the day. Prices tend to fall during off-peak hours; if our deadline is after peak hours, the option of shutting down and waiting becomes more appealing. Third, the best action also depends on the workflow. Different types of computation exhibit varying degrees of parallelism; even within a single workflow, a big cluster may be cost-effective for some of the steps but not the others. The decision to change the cluster size must balance the benefit and cost of parallelism not only at the current point of execution, but also those in the future. Overall, making good adaptation decisions can be overwhelming and tricky without proper system support.

A system that automates or recommends adaptation actions must also deal with a host of messy practical issues. Each cloud provider has its own particularities. For example, Amazon EC2 charges machines by hour, at the market price when each hour of usage begins; if a machine is terminated by Amazon, its last partial hour will be free. If we have added machines to a cluster over time, these machines can be charged at different rates—even though there is only one underlying time series of market prices—because machines acquired at different times may not have their usage hours aligned. Furthermore, adaptation actions are not instantaneous. Ideally, one could look at the market price at beginning of a usage hour and release the machine if that price is too high. In practice, however, because of delay in actuating the release, one has to examine the price and make a decision *before* the hour begins. Then, it is possible for the machine to be charged at a new, different price—hence, uncertainty cannot be eliminated. In comparison, Google charges by minutes instead of hours. Google also sets fixed low prices for "preemptible" instances, but can terminate them at any time. Effec-

*Work on this paper is supported by NSF grant IIS-13-20357. Most of the work was conducted while the first author was at Duke University.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 8
Copyright 2017 VLDB Endowment 2150-8097/17/04.

tively, prices in this market transition between a known low state and some (hidden) high state. In today's fast-moving and competitive cloud computing landscape, we expect that these particularities will likely continue to exist and evolve.

Therefore, instead of relying on rules of thumb, or strategies specific to a spot market's current particularities, we strive for a principled, general approach towards dynamic adaptation, where we can quantify the benefit and risk of our actions, and account for various practical details that matter to performance. To meet this goal, we present a solution based on Markov decision processes (MDPs). For our solution to be practical, we make judicious choices on the level of details in modeling and optimization; we devise methods to speed up optimization; and we further decouple the step of *solving* the optimization problem from that of *applying* the solution at run time, so the former, more expensive step does not add overhead or complexity to execution.

This paper continues the line of work on supporting scalable, matrix-based data analysis in the cloud—specifically Cumulon [4] and Cümülön [5], the latter of which also uses the spot market. Matrix-based data analysis has seen increasing applications in recent years because of the growing popularity of statistical and machine learning methods. Two nice characteristics of such workloads make their automatic optimization more effective than for black-box workloads: declarative specification (in the language of linear algebra) and performance predictability (relatively speaking). Interestingly, declarative specification and cost-based optimization are also pillars of modern databases. Therefore, even though this work focuses on matrix-based workloads, we believe some of our ideas and techniques will also be relevant to database workloads or other declaratively specified data-parallel workloads.

This paper borrows many cost estimation and optimization methods from [4, 5], as well as the stochastic price model for spot markets from [5]. However, as motivated earlier, Cümülön-D takes the new approach of continuous, proactive adaptation, dynamically adjusting its cluster and bidding strategy according to current market condition and execution progress. This new approach is a fundamental departure from our previous work on Cümülön [5], which was reactive—e.g., it would not opportunistically acquire more spot machines when the spot price drops, or release them in anticipation of price hikes. The new proactive adaptation approach of Cümülön-D introduces new opportunities as well as challenges. Thanks to the power of proactive adaptation, Cümülön-D is competitive even without making use of any regular, non-spot machines. On the other hand, highly dynamic adaptation not only requires solving a new, harder optimization problem, but also necessitates a simpler system to tame model complexity and reduce adaptation overhead.

We evaluate Cümülön-D and compare it with other approaches using machines from the Amazon EC2 spot market. We show that Cümülön-D adapts intelligently, not only to the dynamic spot market, but also to performance variations during execution. Our results illustrate the price of uncertainty as well as the advantage of dynamic adaptation. Given the same market price model, Cümülön-D produces more economical results than previous approaches, because of better adaptivity, more realistic workload modeling, and ability to account for various particularities that impact cost.

2 System Overview

We now briefly describe the system design of Cümülön-D. As mentioned in Section 1, Cümülön-D builds on previous work [4, 5], so we shall focus on aspects that are new or different, while referring readers to [4, 5] for other details. We close this section with a discussion highlighting the difference in design philosophy between Cümülön-D and Cümülön.

Cluster and Storage A Cümülön-D (compute) cluster consists of machines from the spot market that can be terminated by the cloud provider.[1] This cluster does not need to include non-spot machines (called *on-demand instances* on Amazon EC2) that have fixed prices and will not be terminated by the provider. For simplicity (and to answer the interesting question of how far we can go with spot machines alone), we shall assume that we do not use non-spot machines for computation for now; our technical report [6] shows how to extend our system and optimization framework in a relatively straightforward way to include this option.

A separate, reliable *master* schedules computation on the compute cluster, monitors the execution progress and the spot market price, and dynamically adjusts the cluster and execution as needed. During execution, the master itself does not perform computation in the workload or solve any hard optimization problem on the fly—it simply looks up appropriate actions from a precomputed *cookbook* (details to be given later). It can run inside the cloud on a non-spot machine or outside the cloud on the user's client machine. We do not consider the cost of the master.

For *stable storage*, Cümülön-D uses a reliable distributed file system shared by but separate from the machines in the cluster. The contents of file system will not be lost if the machines are terminated. For Amazon EC2, EFS (Elastic File System) is a natural choice. Another option would be to run a separate storage cluster such as GlusterFS or HDFS using non-spot machines. Because Cümülön-D's focus on matrix workloads, its storage layer supports accesses to matrix data in the unit of *tiles*, or submatrices of fixed but configurable sizes. By design, tiles are never updated in place. Data integrity is maintained at the tile-level: missing or partially written tiles are detected and do not affect other tiles' integrity.

Besides the stable storage, Cümülön-D deploys a *distributed cache* on all machines in the compute cluster. This cache employs write-through and cache-on-read policies: every write goes to both the cache and the stable storage; if a tile to be read is not in the cache, the request goes to the stable storage and a copy of the tile fetched is cached. Cached tiles may be lost if some (or all) machines are terminated, but because of the stable storage and the write-through policy, there is no need to recover lost tiles in the cache.

Compilation and Execution Cümülön-D compiles an input program into a *workflow*, or a sequence of *jobs* for execution. Each job is executed in a data-parallel fashion using multiple independent *tasks* that do not communicate with each other. Each machine in the cluster is configured into several *slots*, each of which executes one task at a time. Cümülön-D's *scheduler*, which runs on the master, assigns tasks to slots. Each task reads its designated *input split* (portions of the job input, possibly overlapping with the other tasks) and writes its designated *output split* (a portion of the job output, disjoint from the other tasks). Passing of data between jobs happens through the shared stable storage.

At compilation time, Cümülön-D's optimizer produces a triple $(\mathcal{Q}, \mathcal{A}, \mathcal{S})$ given an input program and user-specified deadline and risk tolerance (more on these in Section 5).

- $\mathcal{Q}$, the optimized *workflow template*, contains a *template* for each job in the workflow, which encodes the execution strategy for all tasks in the job (using a DAG of standard operators; see [4] for details). However, templates leave open how input and output are split among tasks; that decision depends on the cluster configuration and will be made at run time (see below).

[1] We also note that Cümülön-D provisions a cluster from a public cloud solely for the purpose of executing a given workflow; we are not targeting scenarios where a cluster is maintained on an ongoing basis to handle a workload mix over time.

- $\mathcal{A}$, the *(action) cookbook*, instructs Cümülön-D how to adapt at run time, and will be detailed in Section 5. For now, think of it as a look-up table that maps a current price and execution state to an adaptation "action." The action may change the cluster and in turn affect how work is split among tasks in a job.
- $\mathcal{S}$ is the *split guide*. For each job, given the current cluster configuration, $\mathcal{S}$ returns the optimal split dimensions for a task in this job. When Cümülön-D's scheduler assigns a task, it consults $\mathcal{S}$ to determine what portion of the work should be given to the task. Since this decision is based on the current cluster configuration, task scheduling automatically adapts according to the actions given by $\mathcal{A}$.

How to come up with the workflow template $\mathcal{Q}$ and split guide $\mathcal{S}$ is not the focus of this paper—we refer readers to [4] for details. On a high level, to find $\mathcal{Q}$, we start with an algebraic representation of the input program, and apply rule-based rewriting to transform the program and partition it into jobs. To find $\mathcal{S}$, we employ a suite of cost models, including an I/O performance model, I/O volume and execution time models for each operator (constructed by benchmarking across a wide range of input shapes and sizes for each machine type), as well as models for predicting job progress from task execution times. Specific to Cümülön-D, when modeling I/O performance, we ignore performance variations due to the contents of the distributed cache. This simplification saves us from having to model the cache contents in the event that subset of machines are terminated. In practice, this assumption is reasonable: as distributed file systems continue to improve, variations in I/O performance will account for a smaller fraction of the total time.

A machine can be terminated by either the cloud provider or Cümülön-D itself. In either case, the master simply returns the input splits assigned to any failed tasks to the pool of remaining work, and then continues scheduling remaining work on remaining machines. By consulting $\mathcal{S}$, it ensures that new assignments are optimal for the updated cluster configuration. Thanks to the stable storage, termination of machines does not affect output of tasks that have already completed, so none of their work needs to be redone.

Discussion and Comparison with Cümülön [5] As discussed in Section 1, the key feature of Cümülön-D that distinguishes it from Cümülön is its continuous, proactive adaptation. This fundamental difference in approaches manifests itself not only in terms of the optimization problem being solved, but also in system design. One notable difference is Cümülön-D's assumption of stable storage independent of its compute cluster. In contrast, Cümülön assumes none and uses one cluster for both computation and storage of intermediate results. Consequently, Cümülön needs to include some non-spot machines in its cluster to provide stable storage. To avoid overwhelming these machines, however, Cümülön does not send every write to its stable storage, but instead adopts a dual-store architecture where useful intermediate results become persisted in the stable storage either through reads naturally occurring during execution or by explicit checkpoint operations strategically added by the system. Nonetheless, data written by completed tasks can be lost under Cümülön, which is not possible under Cümülön-D. This difference in turn leads to other features required by Cümülön but not Cümülön-D, e.g., lineage-based recovery of lost intermediate results, as well as modeling of data access and caching patterns so the Cümülön optimizer can estimate recovery cost. Overall, Cümülön-D has a vastly simpler system than Cümülön.

On the other hand, Cümülön-D's nimbler system enables more sophisticated optimization and adaptation—it can proactively grow and shrink the cluster with relatively low overhead, as there is no complex recovery process after termination of machines. In contrast, Cümülön only makes a one-time decision on how to bid for spot machines, and assumes a single batch of spot machines at any time during execution—not because of any lack of system support but because of the limit of optimization: recovery costs would be difficult to predict otherwise. One could re-invoke Cümülön's optimizer to get another batch of spot machines after losing the original batch, but this adaptation is reactive, not proactive.

In sum, Cümülön-D and Cümülön represent rather different design trade-offs between the system and its optimization. Cümülön's system is more complex, self-sufficient, and has more "smarts" built-in, but predicting its behavior (especially when machines are terminated) becomes so complicated that the scope of optimization needs to be limited. Cümülön-D opts for a simpler, nimbler, and more predictable system with a more sophisticated optimizer. We made this design trade-off for Cümülön-D not only because it presents an interesting opportunity to compare the two approaches, but also because of cloud technology and market trends: cloud providers have been steadily improving their distributed storage offerings in terms of both cost and performance, and Amazon EFS has just become available in mid-2016.

3 Preliminaries for Optimization

As mentioned in Section 2, Cümülön-D compiles a "cookbook" that it consults at run time to generate adaptation actions. Before describing how this process works, we need to first introduce notations and assumptions, clarify what we mean precisely by an adaptation action, and show how we estimate the progress and cost of an execution resulted from applying a sequence of such actions.

Charging Scheme We assume that the cloud provider can announce at any time a new *market price* for each machine type in a spot market. The market price remains the same until further notice, but there is no guarantee of when and how much the price will change again. Users can acquire machines any time by placing a *bid*, which consists of a bid price, as well as the number and type of machines requested. For the bid to succeed, the bid price must be no less than the current market price for the machine type requested. Once a bid is placed, it cannot be modified.

The cloud provider charges each machine in use in time increments of τ_{charge}. Specifically, usage time is rounded up to a multiple of τ_{charge}, and each period of τ_{charge} is charged by the market price at the beginning of that period. For Amazon, $\tau_{\text{charge}} = 3600$s (one hour) although the last partial hour is free if the machine is terminated by Amazon. For Google, $\tau_{\text{charge}} = 60$s (one minute).

Cluster Configuration For simplicity, we assume a *homogeneous* cluster, i.e., all machines come from one spot market, have the same type, and are configured into the same number of slots.[2] These machines can, of course, be acquired and terminated at different times. At compile time, given the workflow, Cümülön-D optimizes for each machine type and each spot market, and picks the best. The optimal number of slots per machine can be determined given the machine type and workflow template [4]. Therefore, in the remainder of this paper, we focus on the problem of optimally *adapting the cluster size*, given the machine type and the number of slots per machine. For brevity, we shall refer to "machines of given type and configuration" simply as "machines" when the context is clear.

Jobs and Speedup Functions Let J denote the number of jobs in the workflow template of interest. Let $T^{(j)}(n)$ denote the execution time of job j ($j = 1, ..., J$) using n machines, assuming no interruption. Let $w^{(j)} = T^{(j)}(1)$ represent the total amount of work in job j, whose unit of measurement is the amount of work

[2]We leave as future work the more general case of a *heterogeneous* cluster where machines have different types and/or come from different markets.

performed by a single machine in unit time. Let $W = \sum_{j=1}^{J} w^{(j)}$ denote the total amount of work in the workflow.

The *speedup function* (well studied in [3]) of job j is given by $g^{(j)}(n) = \frac{T^{(j)}(1)}{T^{(j)}(n)}$. In general, parallelism produces diminishing returns because of its overhead and any inherently serial portion of the computation. In practice, we observe $g^{(j)}(n)$ to be increasing, concave, and sublinear (even for highly parallelizable jobs like matrix multiply), with $g^{(j)}(1) = 1$ and $g^{(j)}(n) \leq n$. Cümülön-D makes no assumption at all on the speedup functions, except that when analyzing our optimization algorithms later, we note $g^{(j)}(n)$'s diminishing-return property naturally caps the number of machines in use simultaneously because a larger cluster would no longer be cost-effective—we denote this cap N.

Note that $T^{(j)}(n) = \frac{w^{(j)}}{g^{(j)}(n)}$; i.e., we can interpret $g^{(j)}(n)$ as the effective "speed" of an n-machine cluster in calculating the execution time of job j. In Section 2, we briefly outlined how to estimate $T^{(j)}(n)$, from which $w^{(j)}$ and $g^{(j)}(n)$ are defined; in subsequent sections, we shall primarily work with $w^{(j)}$ and $g^{(j)}(n)$ instead.

Plan, Actions, and Overhead Given a deadline, we look for a *plan* that encodes the sequence of *actions* over time in order to finish the workload before the deadline. An *action* can be bidding for more machines at some price, or releasing a subset of machines. In formulating and solving the optimization problem, we make two simplifications, but they do not limit our solution in practice:

- First, we discretize time into steps of size τ_{opt}, and consider actions only at the beginning of each time step. Later in Section 5.4, we discuss how to apply our solution in practice such that Cümülön-D can take actions at any time. The choice of τ_{opt} reflects the trade-off between optimization time and solution quality, which we will study experimentally in Section 6.
- Second, we ignore the problem of setting bid price for now, and assume we always bid "high enough." For the purpose of optimization, we simply set bid price to ∞, which means machines will not be terminated by the cloud provider (but can still be released voluntarily by Cümülön-D).[3] Of course, in reality, Cümülön-D does not bid at ∞, and it handles termination by the provider; Section 5.3 discusses how to further derive a practical bidding strategy given a solution to the optimization problem.

Hence, with the simplifications above, assuming the current time is 0 and given deadline d (in the unit of τ_{opt}), we just need to pick the sequence $\langle n_t : 0 \leq t \leq d-1 \rangle$, where n_t denotes the chosen cluster size in time step t. When the context is clear, we will abbreviate the plan as $\langle n_t \rangle$.

As discussed in Section 2, a change in the cluster size at run time automatically affects (through the split guide $\mathcal{S}$) how remaining work is divided into tasks. This effect is also captured by the speedup functions defined earlier in this section. In reality, changing the size of a cluster also incurs various overheads. For example, when new machines are acquired, they require time to initialize, during which they perform no useful work but are charged nonetheless. When machines are terminated or released, some work may be lost in incomplete tasks, which needs to be redone by other machines, thereby delaying execution progress. Let $\mathsf{Overhead}(\varpi, n, n')$ denote the *adaptation overhead* in terms of the amount of non-useful or wasted work incurred, where ϖ is the amount of work left in the workflow at the time when the cluster size changes from n to n'. We derived an estimate for this quantity by benchmarking the initialization overhead and by using the techniques outlined in Section 2 for estimating work wasted due to termination. We can reasonably assume that overhead processing for one adaptation action does not extend beyond the duration of one time step (otherwise we should choose a longer τ_{opt} to avoid excessive adaptation).

Plan Progress We now show how to measure the progress of execution under a plan. Given a workflow with J jobs and total work $W = \sum_{j=1}^{J} w^{(j)}$, let $\mathsf{Progress}(\varpi, n, n')$ denote the progress made by a cluster in a time step. Specifically, ϖ is the amount of work left at the beginning of the time step; n and n' are the cluster sizes in the previous and current time steps, respectively; $\mathsf{Progress}(\varpi, n, n')$ returns the amount of work left in the workflow at the end of the current time step.

Definition 1. *If $n' = 0$, $\mathsf{Progress}(\varpi, n, n') = \varpi$. Otherwise,*

- *let job $j_{\vdash} = \max\{j : \sum_{j=1}^{j} w^{(j)} \leq W - \varpi\} + 1$ be the one that the current time step starts in;*
- *let $\varpi_{\vdash} = \left(\sum_{j=1}^{j_{\vdash}} w^{(j)}\right) - (W - \varpi)$ denote the amount of work in job $j_{\vdash}$ remaining at the beginning of the current time step;*
- *let job $j_{\dashv} = \max\{j : \left(\frac{\varpi_{\vdash} + \mathsf{Overhead}(\varpi,n,n')}{g^{j_{\vdash}}(n')}\right) + \left(\sum_{j=j_{\dashv}+1}^{j} \frac{w^{(j)}}{g^{(j)}(n')}\right) < \tau_{\mathsf{opt}}\} + 1$ be the one that the current time step stops in;*
- *let $\varpi_{\dashv} = g^{(j_{\dashv})}(n') \cdot \left(\left(\frac{\varpi_{\vdash} + \mathsf{Overhead}(\varpi,n,n')}{g^{j_{\vdash}}(n')}\right) + \left(\sum_{j=j_{\dashv}+1}^{j_{\dashv}} \frac{w^{(j)}}{g^{(j)}(n')}\right) - \tau_{\mathsf{opt}}\right)$ (or 0 if $j_{\dashv} = J + 1$) denote the amount of work in job $j_{\dashv}$ remaining at the end of the current time step.*

Then, $\mathsf{Progress}(\varpi, n, n') = \varpi_{\dashv} + \sum_{j=j_{\dashv}+1}^{J} w^{(j)}$.

With Progress, we can define $\mathsf{WorkLeft}(\langle n_i : i \leq t \rangle)$, the amount of work left in the workflow after time step t by following the plan.

Definition 2. *The following recurrence defines WorkLeft:*

- *$\mathsf{WorkLeft}(\langle\rangle) = W$.*
- *$\mathsf{WorkLeft}(\langle n_i : i \leq t \rangle) = \mathsf{Progress}(\mathsf{WorkLeft}(\langle n_i : i \leq t-1 \rangle), n_{t-1}, n_t)$, with $n_{-1} = 0$ for notational convenience.*

Plan Cost Let $p(\tau)$ denote the market price at time τ; recall that it may change at any time. $\mathsf{Cost}(\langle n_t \rangle)$, the (monetary) cost of executing a plan, can be hard to derive analytically in general, because of the discrepancy between τ_{opt} and τ_{charge}. τ_{charge} is defined by the cloud provider, but we can choose τ_{opt} ourselves. We limit the choice of τ_{opt} to two cases: τ_{charge} divides τ_{opt}, or τ_{opt} divides τ_{charge}.

We start with the easier case when τ_{charge} divides τ_{opt}. The n_t machines during time step t will be charged $\tau_{\mathsf{opt}}/\tau_{\mathsf{charge}}$ times. Thus we have $\mathsf{Cost}(\langle n_t \rangle) = \sum_{t=0}^{d-1} \left(n_t \sum_{i=0}^{\tau_{\mathsf{opt}}/\tau_{\mathsf{charge}}-1} p(t\tau_{\mathsf{opt}} + i\tau_{\mathsf{charge}})\right)$, where d is the deadline. When $\tau_{\mathsf{opt}} = \tau_{\mathsf{charge}}$, $\mathsf{Cost}(\langle n_t \rangle)$ simplifies to $\sum_{t=0}^{d-1} n_t p(t\tau_{\mathsf{opt}})$.

Now consider the case when τ_{opt} divides τ_{charge}. The situation is more complicated because at the beginning of time step t, not all n_t machines will be charged as some of them were already charged at earlier time steps. To help us define Cost, we derive from $\langle n_t \rangle$ another sequence $\langle m_t \rangle$, where m_t denotes the number of machines that actually get charged at time step t. Let $\kappa = \tau_{\mathsf{charge}}/\tau_{\mathsf{opt}}$. Note that once a machine has been charged (for another τ_{charge}), there is no incentive for us to terminate it before κ time steps as we have already paid for its usage. Hence, for time step t, we should retain all $\sum_{i=1}^{\kappa-1} m_{t-i}$ machines charged in the previous $\kappa - 1$ time steps. In addition, we pick $m_t = n_t - \sum_{i=1}^{\kappa-1} m_{t-i}$ machines to be charged in time step t. These m_t machine should come as much as possible from the $m_{t-\kappa}$ machines that we paid for in time step $t - \kappa$; we can actively terminate some if $m_t < m_{t-\kappa}$, or bid for more if $m_t > m_{t-\kappa}$. Hence, $\langle m_t \rangle$ can be derived from $\langle n_t \rangle$ using the recurrence $m_t = n_t - \sum_{1 \leq i < \kappa,\ t-i \geq 0} m_{t-i}$. (Alternatively, we can

[3] If bidding at ∞ sounds crazy, recall that machines are charged not at their bid price, but at their market price. Again, we stress that bidding at ∞ is a only a simplification to the optimization problem. The effect of bidding at ∞ on the result of optimization is limited, because Cümülön-D's optimizer considers releasing machines proactively when the price becomes too high.

derive $\langle n_t \rangle$ from $\langle m_t \rangle$ using $n_t = \sum_{0 \le i < \kappa,\ t-i \ge 0} m_{t-i}$.) With $\langle m_t \rangle$ defined, we can now derive the cost under the case where τ_{opt} divides τ_{charge} as $\text{Cost}(\langle n_t \rangle) = \sum_{t=0}^{d-1} m_t p(t\tau_{\text{opt}})$.

Finally, we unify the definitions for the above cases into one.

Definition 3. *$\text{Cost}(\langle n_t \rangle) = \sum_{t=0}^{d-1} m_t \text{Charge}(t)$, where*

- $\kappa = \lceil \tau_{\text{charge}} / \tau_{\text{opt}} \rceil$,
- $m_t = n_t - \sum_{1 \le i < \kappa,\ t-i \ge 0} m_{t-i}$ *for* $t = 0, \ldots, d-1$, *and*
- $\text{Charge}(t) = \sum_{i=0}^{\lceil \tau_{\text{opt}} / \tau_{\text{charge}} \rceil - 1} p(t\tau_{\text{opt}} + i\tau_{\text{charge}})$.

Note that if τ_{charge} divides τ_{opt}, we have $\kappa = 1$ and $\langle m_t \rangle$ is the same as $\langle n_t \rangle$ per above definition.

We have so far omitted the monetary cost of I/Os themselves (although as discussed in Section 2, we already account for their contribution to execution time, and in turn, cost of machines). Charging schemes for I/Os vary greatly. For example, Amazon EFS charges by the amount of data stored over time, but Amazon S3 additionally charges by the number of requests. Running a separate storage cluster, on the other hand, incurs only machine cost for that cluster. It is straightforward to incorporate various I/O charging schemes into our cost model, but for simplicity, we will ignore the monetary cost of I/Os for this paper, and revisit this issue in the context of experiments in our technical report [6].

4 Optimization with Deterministic Price

Before considering market uncertainty in Section 5, let us study the simpler (but unrealistic) problem where the future market prices are known exactly. The optimization problem can be formalized as follows: given a deadline d, pick $\langle n_t : 0 \le t \le d-1 \rangle$ to minimize $\text{Cost}(\langle n_t \rangle)$ subject to $\text{WorkLeft}(\langle n_t \rangle) = 0$, i.e., the workload completes before the deadline. We give two algorithms.

A Simple Greedy Algorithm Suppose τ_{charge} divides τ_{opt}, and assume that all per-job speedup functions are the same, i.e., $g^{(j)}(n) = g(n)$ for all j. In this case we can adopt a simple approach that iteratively and greedily allocates one machine to one selected time step at a time. Starting with $n_t = 0$ for all t, we pick the time step t that maximizes $(g(n_t + 1) - g(n_t))\,/\text{Charge}(t)$ and increment n_t by one. Since the future market prices are known, $\text{Charge}(t)$ (Definition 3) are known as well. Intuitively, we always choose the allocation with the highest work/cost ratio. We repeat this process until $\text{WorkLeft}(\langle n_t \rangle) = 0$.

We can show that this simple greedy algorithm is optimal assuming a concave $g(n)$ and no adaptation overhead, i.e., $\text{Overhead}(\varpi, n, n')$ is always 0. For a formal proof, see our technical report [6].

A Dynamic Programming Algorithm In general, the choices of cluster size across time steps are interrelated and the greedy strategy is suboptimal. Instead, we solve the problem with dynamic programming (DP). Let $C(t, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1} \rangle)$ denote the minimum cost of finishing ϖ amount of work left starting from time step t, when the numbers of machines last charged at the beginning of previous κ time steps are given by the sequence $\langle m_{t-\kappa}, \ldots, m_{t-1} \rangle$. Note that for the $m_{t-\kappa}$ machines, a new charging period is starting in time step t; there is need to model m_i for $i < t - \kappa$, because machines that were charged before time step $t - \kappa$ must have been either released or recharged during $[t-\kappa, t-1]$.

We have the following DP state transition function (with $m_i = 0$ when $i < 0$ for notational convenience):

$$\begin{aligned} &C(t, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1} \rangle) \\ &= \min_{m_t \ge 0} \left\{ m_t \text{Charge}(t) + C(t+1, \text{Progress}(\varpi, n_{t-1}, n_t), \langle m_{t-\kappa+1}, \ldots, m_{t-1}, m_t \rangle) \right\}, \end{aligned}$$

where $n_{t-1} = \sum_{i=t-\kappa}^{t-1} m_i$ and $n_t = \sum_{i=t-\kappa+1}^{t} m_i$.

Recall from Section 3 that the time dimension t is discretized into d steps, and N denotes the maximum number of machines in use simultaneously (practically upper-bounded because of the diminishing return of parallelism). We also discretize the remaining work dimension ϖ; finer-grained discretization makes the DP more expensive to solve but may improve solution quality. In this paper, we set the unit amount of work to be that completed by one machine in τ_{opt}, which we found to offer acceptable quality in our experiments. The DP algorithm works backwards in time, starting with $C(d, 0, \langle m_{d-\kappa}, \ldots, m_{d-1} \rangle) = 0$ for all choices of $m_{d-\kappa}, \ldots, m_{d-1}$ such that $0 \le m_{d-\kappa} + \cdots + m_{d-1} \le N$. The time complexity of the algorithm is $O(dWN\binom{N+\kappa}{\kappa})$, or simply $O(dWN^{\kappa+1})$, where W is discretized as an integer as discussed above.

This DP algorithm is impractical as it assumes perfect knowledge of future market prices. Nonetheless, given each price trace, we can apply this algorithm to obtain the lowest achievable cost; comparing that with what is achievable by an algorithm with only uncertain knowledge, we get a measure of "price of uncertainty," which we shall explore in Section 6.

5 Optimization with Uncertain Future Price

Given a stochastic market price model, we want to pick, at the beginning of each time step, the "best" action based on the current market price, time remaining before the deadline, as well as cluster and execution states. To handle uncertainty, intuitively, we define the "best" action to one that minimizes a linear combination of expected remaining cost and its standard deviation, whose coefficient captures the user's risk tolerance. We first describe how to approach this problem as an MDP. Then, we discuss how to apply the solution to adapt dynamically in practice.

5.1 Formulation as an MDP

We assume the market prices are discrete and that the effective size of the domain is P. For notational convenience (as we take actions only for each time step), we define $p_t = p(t\tau_{\text{opt}})$; i.e., subscripts of p refer to time steps while arguments of p refer to time.

We use a stochastic market price model [5] trained using historical price data from Amazon EC2. We assume the model is Markovian, i.e., the probability of local upward or downward movement is arbitrarily dependent on the current price and time. From this model, we derive:[4]

- *Price transition function*: Let $\mathbb{p}_{t,\Delta}^{u,v}$ denote the probability that the price becomes v at the beginning of time step $t + \Delta$ given that it was u at the beginning of time step t; i.e.: $\mathbb{p}_{t,\Delta}^{u,v} = \mathbb{P}[p_{t+\Delta} = v \mid p_t = u]$.
- *Expected charge function*: Let $\mathbb{c}_t^u = \mathbb{E}[\text{Charge}(t) \mid p_t = u]$ denote the expect cost charged for a machine during time step t given that the price was u at the beginning of the time step. Recall the definition of $\text{Charge}(t)$ in Definition 3 as a summation of prices; $\mathbb{c}_t^u$ can be easily derived by linearity of expectation.

Our market price model captures diurnal as well as weekly behaviors (see [5] for details). Section 6 will give an example where the periodic price behavior influences the optimization outcome.

We can now formulate the optimization problem as an MDP. A state in our MDP has the form $(t, u, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1} \rangle)$, representing the following situation at the beginning of time step t: the market price is u; we still have ϖ amount of work left in the workflow to finish before deadline d; and $\langle m_{t-\kappa}, \ldots, m_{t-1} \rangle$ machines were charged during the previous κ time steps, respectively.

[4] A technicality: recall that we define time (and time steps) relative to the starting time; therefore, $\mathbb{p}_{t,\Delta}^{u,v}$ and $\mathbb{c}_t^u$ are relative to and hence implicitly dependent on the starting time.

This representation of state is similar to that of the DP state in Section 4, except the additional price component. In this state, we need to choose the cluster size n_t (equivalently, m_t) for time step t. Choosing m_t, or equivalently, $n_t = \sum_{i=t-\kappa+1}^{t} m_i$, will incur an expected cost of $m_t \mathbb{c}_t^u$ for time step t, and we then transition to one of the states for time step $t+1$ of the form $(t+1, v, \varpi', \langle m_{t-\kappa+1}, \ldots, m_t\rangle)$ with probability $\mathbb{p}_{t,1}^{u,v}$, where ϖ' is the amount work left at the end of time step t.

Formally, we define policy $\mathfrak{L} : (t, u, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1}\rangle) \to m_t$, which maps any given state to an action m_t. Given current state s, let $C(\mathfrak{L}, s)$ denote the additional cost to finish the remaining work under $\mathfrak{L}$ starting from s, and let $C(\mathfrak{L}, s \mid m)$ denote the additional cost to finish the remaining work if we choose to charge m machines in the current time step and then follow $\mathfrak{L}$ afterwards. Clearly, $C(\mathfrak{L}, s) = C(\mathfrak{L}, s \mid \mathfrak{L}(s))$. These quantities are random variables. Let $\overline{C}$ and $\widetilde{C}$ denote their expectation and variance, respectively, which can be derived as follows (see [6] for details):

Lemma 1. *Let $s_t = (t, u, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1}\rangle)$ denote the current state at time step t. We have:*

$$\overline{C}(\mathfrak{L}, s_t \mid m_t) = m_t \mathbb{c}_t^u + \sum_v \mathbb{p}_{t,1}^{u,v} \overline{C}(\mathfrak{L}, s_{t+1}), \quad \textit{and}$$

$$\widetilde{C}(\mathfrak{L}, s_t \mid m_t) = \sum_v \mathbb{p}_{t,1}^{u,v} \widetilde{C}(\mathfrak{L}, s_{t+1}) + \sum_v \mathbb{p}_{t,1}^{u,v} (\overline{C}(\mathfrak{L}, s_{t+1}))^2 - \left(\sum_v \mathbb{p}_{t,1}^{u,v} \overline{C}(\mathfrak{L}, s_{t+1})\right)^2,$$

where $n_{t-1} = \sum_{i=t-\kappa}^{t-1} m_i$, $n_t = \sum_{i=t-\kappa+1}^{t} m_i$, *and* $s_{t+1} = (t+1, v, \textit{Progress}(\varpi, n_{t-1}, n_t), \langle m_{t-\kappa+1}, \ldots m_{t-1}, m_t\rangle)$.

Since we are given a deadline d, this MDP has a finite time horizon, with the constraint at time step d, we must be in a state with $\varpi = 0$. Note that the state space covers all possible futures, and $\mathfrak{L}$ encodes the best action m_t to take for any future state.

We now define policy $\mathfrak{L}_\gamma$, which in every time step minimizes a linear combination of expected remaining cost and standard deviation. The user-specifiable parameter $\gamma \geq 0$ controls how much risk the user is willing to take: $\gamma = 0$ means minimizing expected cost along while ignoring risk; a larger γ penalizes policies that lead to executions with highly variable costs. Then, the optimal action at time step t is defined as:

$$\mathfrak{L}_\gamma(s_t) = \arg\min_{m_t \geq 0} \left(\overline{C}(\mathfrak{L}_\gamma, s_t \mid m_t) + \gamma \sqrt{\widetilde{C}(\mathfrak{L}_\gamma, s_t \mid m_t)} \right).$$

5.2 Solving the MDP

We solve the finite-horizon MDP with DP, starting from time step d in states with $\varpi = 0$, and computing the remaining cost expectation $\overline{C}$, variance $\widetilde{C}$, and $\mathfrak{L}_\gamma$ backwards in time for every state. Eventually, this backward induction process ends with states for time step 0. These states cover all possible prices at time step 0, allowing Cümülön-D to adapt to any potential price change between the time when optimization starts and the time when execution starts (time step 0).

The total number of states is $O(dPW\binom{N+\kappa}{\kappa}) = O(dPWN^\kappa)$. For each of the $O(PWN^\kappa)$ states associated with time step t, we need to consider up to N possible m_t values. Costing out each m_t option requires evaluating the summations in Lemma 1, which naively takes $O(P)$ time. Hence, the overall time complexity of a naive DP implementation is $O(dP^2WN^{\kappa+1})$.

Using some preprocessing, we can reduce the complexity by a factor of $\frac{PN}{P+N}$. In practice, this reduction is easily more than an order of magnitude because N and P are on the order of a hundred. Specifically, for each combination of $(t, u, \langle m_{t-\kappa+1}, \ldots, m_t\rangle)$ and each possible ϖ' value, we precompute the summations:

$$\sum_v \mathbb{p}_{t,1}^{u,v} \overline{C}(\mathfrak{L}_\gamma, (t+1, v, \varpi', \langle m_{t-\kappa+1}, \ldots, m_t\rangle)),$$
$$\sum_v \mathbb{p}_{t,1}^{u,v} \widetilde{C}(\mathfrak{L}_\gamma, (t+1, v, \varpi', \langle m_{t-\kappa+1}, \ldots, m_t\rangle)), \text{ and}$$
$$\sum_v \mathbb{p}_{t,1}^{u,v} (\overline{C}(\mathfrak{L}_\gamma, (t+1, v, \varpi', \langle m_{t-\kappa+1}, \ldots, m_t\rangle)))^2.$$

Precomputation takes $O(dP^2WN^\kappa)$ time over all time steps. For each state $(t, u, \varpi, \langle m_{t-\kappa}, \ldots, m_{t-1}\rangle)$, costing out each m_t option takes only constant time using the precomputed summations. Hence, backward induction takes $O(dPWN^{\kappa+1})$ time overall, bringing the total (including precomputation) to $O(dPWN^\kappa(P+N))$.

5.3 Actuation Delays and Skyline Bidding

We now turn to the issue of how to set bid prices, which we have ignored thus far. One main reason for setting a bid price is to protect against the unfortunate case when the market price surges and our machines are charged too high (we shall define what we mean by "too high" shortly). If adaptation actions are instantaneous, then there will be no penalty for bidding at ∞, because at run time, Cümülön-D can simply check each machine at the beginning of each of its usage periods of length $\tau_{\textsf{charge}}$, and release it if the market price is too high. In reality, however, adaptation actions may have delays. Let $(\delta_\triangleright, \delta_\square)$ denote the *actuation delays* for acquiring and releasing machines, respectively: if we issue a bid request at time τ, machines we get will be charged from time $\tau + \delta_\triangleright$; if we release a machine at time τ, it will be considered still in use until $\tau + \delta_\square$ and charged accordingly. On Amazon EC2, $\delta_\triangleright$ is around five minutes while $\delta_\square$ is around several seconds. As discussed in Section 1, with actuation delays, we can no longer eliminate the possibility of being charged too high under ∞-bidding. Hence, we need the extra protection offered by bidding at a maximum price.

With or without actuation delays, we need to establish what price is "too high." It turns out that an answer already lies in our policy $\mathfrak{L}_\gamma$ obtained by solving the MDP. Let $\rho(t, \varpi)$ denote the upper bound on the price we are willing to pay for a machine when there is ϖ amount of work left at the beginning of time step t. Consider a "worst-case" state of the form $s_u = (t, u, \varpi, \langle 0, \ldots, 0\rangle)$; i.e., the current price is u and there are no machines with remaining usage time that have been paid for. If $\mathfrak{L}_\gamma(s_u) = 0$, i.e., $\mathfrak{L}_\gamma$ tells us not to get any machine, then the price of u is too high. Therefore, we define $\rho(t, \varpi) = \min\{u \mid \mathfrak{L}_\gamma(s_u) = 0\}$.

Generally speaking, $\rho(t, \varpi)$ is higher when t is closer to deadline d and when ϖ is far from 0. In practice, "penny-pinching" by setting the bid price exactly at $\rho(t, \varpi)$ may lead to acquiring machines at low bid prices early during execution, only to lose and reacquire them at higher bid prices later, incurring more overhead. Hence, we must weigh the benefit of capping the price against a higher chance of losing machines and incurring extra overhead. To this end, Cümülön-D "looks ahead" in time in $\mathfrak{L}_\gamma$ for a potentially higher bid price. Specifically, at the beginning of time step t, with ϖ amount of work left, Cümülön-D sets the bid price at $\max\{\rho(i, \varpi) \mid t \leq i \leq t + \Delta_{\textsf{ahead}}\}$, where the parameter $\Delta_{\textsf{ahead}}$ specifies the number of time steps to "look ahead." We call our strategy *skyline bidding* because its bid price, usually a non-decreasing function of time, resembles a rising staircase or skyline.

Finally, we slightly modify the MDP formulation in Section 5.1 in order to account for actuation delays. In practice, to actuate a cluster change by time τ, Cümülön-D must decide to act at time $\tau - \delta$, where $\delta = \max\{\delta_\triangleright, \delta_\square\}$. Cümülön-D only knows the market price at that time. Therefore, we interpret the price component u in the MDP state generally as the market price at the time of decision, which is δ before the beginning of each time step; we also incorporate this time offset into the price transition and expected charge functions: $\mathbb{p}_{t,\Delta}^{u,v} = \mathbb{P}[p((t+\Delta)\tau_{\textsf{opt}} - \delta) = v \mid p(t\tau_{\textsf{opt}} - \delta) = u]$ and $\mathbb{c}_t^u = \mathbb{E}[\textsf{Charge}(t) \mid p(t\tau_{\textsf{opt}} - \delta) = u]$. Lastly, the remaining work component ϖ, instead of representing the amount of work currently left, is replaced by an estimate of what will be left later δ amount of time (i.e., at the beginning of time step t).

5.4 Applying Cookbook to Execution

The policy $\mathfrak{L}_\gamma$ obtained by solving the MDP becomes the adaptation "cookbook" used by Cümülön-D at run time, as discussed in Section 2. We now detail how to apply this cookbook in practice.

Roughly speaking, given a state (time, market price, execution progress, and cluster configuration), Cümülön-D simply looks up the action returned by $\mathfrak{L}_\gamma$ and applies it. One technicality is that we have discretized the dimensions of an otherwise continuous state space, so $\mathfrak{L}_\gamma$ only suggests actions for a set of discrete states. Cümülön-D obtains actions for other states by interpolating suggested actions for nearby discrete states.

Strict Mode The most straightforward way of applying $\mathfrak{L}_\gamma$ is to follow it strictly. In this *strict mode* of application, Cümülön-D consults $\mathfrak{L}_\gamma$ once every τ_{opt} amount of time—always δ ahead of the beginning of each time step.

Dynamic Mode and Delayed Release The strict mode may be too restrictive in practice—for example, when we observe a drastic change in the market price, it may be better to act immediately instead of waiting until the next prescribed adaptation time. Therefore, Cümülön-D by default applies $\mathfrak{L}_\gamma$ in what we call *dynamic mode*. In this mode, Cümülön-D monitors the market continuously and consults $\mathfrak{L}_\gamma$ whenever one of the following conditions becomes true: 1) the market price changes (including when any machine is terminated by the provider); 2) at least one machine is about to be charged for another τ_{charge} amount of time; and 3) time since the last consultation of $\mathfrak{L}_\gamma$ has exceeded a threshold τ_{check} (typically a constant fraction of τ_{opt}; we use $\tau_{\text{check}} = \frac{1}{2}\tau_{\text{opt}}$ by default).

In the dynamic mode, since machines can start in between time step boundaries, their charging periods may not be aligned. Therefore, to derive the input m_i's when consulting $\mathfrak{L}_\gamma$, Cümülön-D needs to "snap" each machine's charging period to those whose boundaries are multiples of τ_{opt} away from the planned time of action. More precisely, we let m_{t-i} be the number of machines who was last charged no later than $\tau - i\tau_{\text{opt}}$ and before $\tau - (i-1)\tau_{\text{opt}}$, where τ is the planned time of action (δ after the time of planning).

Cümülön-D then follows the suggestion of $\mathfrak{L}_\gamma$ and changes the cluster size if needed, but with an important exception which we call the rule of *delayed release*. This rule says that we never release a machine if it still has remaining usage time that we have already paid for (recall that machines are charged in time increments of τ_{charge}). This rule is clearly harmless because we do not incur any immediate charges for the extra machines. Later, before the paid period expires, Condition 2 above would automatically trigger a reassessment of the situation, and at that time we may still decide to release the machine.[5]

6 Experiments

Platform and Implementation Our experiments are conducted in the context of Amazon EC2, although Cümülön-D do not limit its support of charging schemes to Amazon's. In fact, to make their results easier to interpret, our experiments by default use a market with $\tau_{\text{charge}} = 1$sec and zero actuation delay ($\delta_{\triangleright} = \delta_{\square} = 0$sec). In addition, we experiment with Amazon's charging scheme as well as varying actuation delays.

We have built Cümülön-D's execution engine on top of Hadoop, although for efficient parallelization of matrix operations, we do not follow the MapReduce model; see [4] for details. Besides its spot-only compute cluster, Cümülön-D's master runs on a separate `c1.medium` non-spot machine. The distributed cache is currently implemented using an HDFS over the compute cluster. We set the HDFS replication factor to one, because it is okay to lose cached tiles when machines are released or terminated. For stable storage, we use another HDFS running on a separate storage cluster consisting of 10 `c1.medium` non-spot machines, with replication factor of one. This configuration was adequate for all workloads we experimented with. Amazon EFS would have been a natural choice for stable storage, but at the time of writing this paper, EFS was not yet available publicly. Finally, we note that Cümülön-D does not rely on any special features of Hadoop and HDFS; they can be replaced with alternatives that offer different price-performance points.

Modeling and Optimization Setup For experiments, we choose to focus on the machine type `c1.medium` on Amazon EC2. We build cost models by benchmarking as discussed in Section 2. We use the stochastic market price model from [5], which was trained with historical spot price data in the first six months of 2014 for `c1.medium` in zone `us-east-1a`, one of the more challenging markets with plenty of spikes and high-price regions.

Unless otherwise noted, we set $\gamma = 0.1$ in the objective function of Cümülön-D's optimizer, and use $\tau_{\text{opt}} = 1$hr (see [6] for experiments that vary τ_{opt} to study its influence on the trade-off between optimization time and quality). We set the maximum cluster size $N = 100$: in our experiments, even for highly parallelizable jobs like matrix multiply, the monetary cost of a super-sized cluster quickly outweighs its benefit. We discretize the market price with the granularity of one cent, and the amount of work with the granularity of one machine-hour.

Under the default setting of $\delta_{\triangleright} = \delta_{\square} = 0$, we always set bid price to ∞, which is safe as discussed in Section 5.3. In other experiments with non-default settings, Cümülön-D uses skyline bidding with look-ahead $\Delta_{\text{ahead}} = 4$ in Section 5.3.

Workloads We use two workloads inspired by real-world applications, also used in [5]. The first one concerns singular value decomposition (SVD), key to many data analysis methods. We consider the first and most expensive step of a randomized SVD algorithm [11]: $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$, where $\mathbf{A}$ is a $102{,}400 \times 102{,}400$ matrix, $\mathbf{G}$ is $2048 \times 102{,}400$, and $k = 5$. We call this workload *RSVD-1*. The second one is Gaussian non-negative matrix factorization (GNMF) [10], with applications such as document clustering and recommendation systems. Our workload involves a $7{,}510{,}016 \times 640{,}137$ word-document matrix derived from a 2.5GB wiki text corpus; the number of topics sought is 100. We call this workload *GNMF*. Besides *RSVD-1* and *GNMF*, we also use synthetic workloads whose results allow more intuitive interpretation.

Evaluation Methods Evaluation with an uncertain spot market is challenging. For a given spot market and machine type, there is only one price trace in reality. Each execution gives one data point, insufficient for quantifying uncertainty. We will also likely miss price traces in the "tail" where the user's risk tolerance is tested and the actions of Cümülön-D are more interesting and important. Hence, for experiments, we simulate future prices and "play" a simulated price trace for each execution. We let Cümülön-D adapt the execution, and charge and terminate machines according to the trace. The collection of resulting costs, one for each execution, approximates the distribution of execution cost under Cümülön-D.

In more detail, given starting time and price, we simulate our stochastic market price model multiple times to obtain a training set and a test set of price traces. Using the training set, we estimate the price transition and expected charge function (Section 5.1). Using the test set of 100,000 traces, we obtain 100,000 resulting costs by following a Cümülön-D policy, from which we then obtain mean and standard deviation. By default we do not include the costs of

[5] It is worth noting that, when operating in the strict mode, we do not need an explicit rule for delayed release, because it is implied (and hence automatically enforced) by the MDP solution; see [5] for a detailed explanation.

the master and storage cluster because they depend on the setup and are the same across policies. A closer examination of storage cost and how it affects comparison with Cümülön is presented in [6].

Our experiments focus on two market scenarios, $S_{\$0.02}$ and $S_{\$0.2}$. The starting time for both is 0am on Monday. The starting prices are $\$0.02$ (the most common price observed) and $\$0.20$ (higher than the fixed, non-spot price of $\$0.145$), respectively.

The second challenge is the cost of evaluation. Because of dynamic adaptation, different price traces lead to different execution traces. For a large test set, multiple real executions—with each typically costing several dollars—become prohibitively expensive. Therefore, we have built an execution simulator for Cümülön-D. The simulation is performed at the level of tasks and slots (Section 2). Once a task is scheduled, the simulator randomly draws its execution time from a normal distribution derived from cost models discussed in Section 2. When a machine is released or terminated, the failure and rescheduling of its ongoing tasks are simulated. For simplicity and speed, the simulator does not track the contents of the distributed cache; its impact is limited because, thanks to the stable storage, data written by completed tasks never need to be recomputed. Finally, the simulator also accounts for adaptation overheads and actuation delays. Section 6.2 presents some results comparing simulated and real execution costs.

Alternatives Compared We experimentally compare Cümülön-D with *Fixed*, which always bids for a fixed number of machines at a fixed price (when the workflow starts or when the machines are terminated); and *DBA* [18], which dynamically adjusts the bid price (but not the bid size) as the deadline approaches, and is proven to be optimal under Amazon's charging scheme assuming perfect speedup. We also compare with a practically infeasible approach *Oracle*, which has perfect knowledge of future prices, and uses the DP algorithm in Section 4 to obtain the optimal plan for each trace in the test set. While clearly impractical, *Oracle* offers lower bounds on costs, which we use to gauge the "price of uncertainty."

We also have extended Cümülön-D to Cümülön-D+, which considers the possibility of using non-spot machines in addition to spot machines (see [6] for details). Besides comparing with Cümülön-D (spot-only), we also compare Cümülön-D+ with *Fixed+*, which can be seen as an application of the approach by *Dyna* [21] to our setting. Finally, in [6], we compare Cümülön-D+ with Cümülön [5], accounting for the costs of storage options.

A Note on Optimization Time Because of space limit, we refer interested readers to [6] for optimization time measurements. In all our experiments, Cümülön-D spends no more than a few minutes to compute its cookbook on a standard desktop computer. This computation occurs only at compile time, and is negligible compared with the workflow execution times (on the order of hours).

6.1 A Closer Look at a Cookbook

We begin by developing some intuitive understanding of what a Cümülön-D cookbook does. Recall that the cookbook is based on the MDP policy $\mathfrak{L}_\gamma$, which maps a state to an action. To make $\mathfrak{L}_\gamma$ easier to understand, we consider a synthetic workflow of three matrix multiply jobs, with $w^{(1)} = 100$, $w^{(2)} = 10$, and $w^{(3)} = 200$ machine-hours worth of work respectively. The deadline is in 40 hours. Even under the default setting of $\kappa = 1$, the simplified MDP state $(t, u, \varpi, \langle m_{t-1} = n_{t-1} \rangle)$ is four-dimensional. Hence, we choose three two-dimensional slices of the state space to visualize in Figure 1. For all three slices, we fix $n_{t-1} = 0$; i.e., we have an empty cluster at hand—either we have not yet acquired any machines, or they have all been released (by Cümülön-D) or terminated (by the provider) earlier. Then, in each slice, we fix the value for one dimension and let the other two vary.

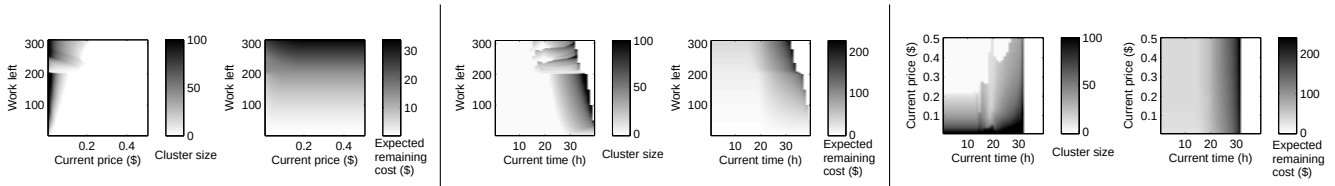
Figure 1a shows the slice with $t = 10$, which is 30 hours before the deadline. As we vary the values of the two free dimensions (current price u and work left ϖ), the density plot on the left shows the action (new cluster size) suggested by the cookbook, while the one on the right shows the expected remaining cost by following the cookbook (as estimated by the optimizer). On the left, we see an upper-left triangular region with $n_t > 0$, where n_t tends to decrease with higher u and increase with higher ϖ. Intuitively, if the current price u is high, Cümülön-D bids for fewer machines because there is still plenty of time left for price to drop, allowing it to finish the work for cheaper. On the other hand, if the amount ϖ of work left is high, Cümülön-D tends to bid for more machines to ensure progress and avoid the situation where, as the deadline looms, so much work is left that we have to resort to a large, less cost-effective cluster. One exception to this trend happens around $\varpi \in (200, 250)$. The reason is that the second job with $w_2 = 10$ benefits much less from parallelism than the other two larger jobs; a smaller cluster would be more cost-effective for this job. Another interesting point can be seen in the plot on the right—the expected remaining cost depends mostly on the work left, and less on the current market price. This feature is nice, meaning that Cümülön-D is effective in limiting the impact of transient market fluctuations.

Figure 1b shows the slice with current price $u = \$0.25$. This price is high, so Cümülön-D does not bid unless it is close to the deadline or too much work is left, as illustrated in the left plot by a lower-left region with $n_t = 0$. The blank upper-right region contains "infeasible" states where we will miss the deadline even if we use the maximum $N = 100$ machines. For the same reason as in Figure 1a, Cümülön bids for fewer machines when ϖ is around 200 to 210. Above this region there are several other "ripples" due to our setting of $\tau_{\mathsf{opt}} = 1\text{hr}$ (because of space limit, see [6] for a detailed explanation). Note that this rippling effect also exists in Figure 1a, albeit in a less noticeable way.

Figure 1c shows the slice with $\varpi = 310$, which means that we have not yet made any progress. The left plot shows that, intuitively, Cumulon tends to bid for more machines when the current price is lower and when the deadline is closer. The blank vertical region on the right ($t > 32$) shows infeasible states, where we can no longer meet the deadline. Another interesting point can be seen from the spike around $t = 20$ along the boundary between where Cümülön-D decides to acquire some machines ($n_t > 0$) and where it decides to "wait and see" ($n_t = 0$). This spike can be explained by the market: $t = 20$ corresponds to Monday 8pm ($t = 0$ is midnight), the beginning of off-peak hours when the price tends to be low. Even if the actual price at that time is high, it is likely temporary. Hence, we may not want to miss the chance to acquire some machines for the hour (recall $\tau_{\mathsf{opt}} = 1\text{hr}$), because their expected cost over the hour may still be low (recall that we pay the price at the beginning of each $\tau_{\mathsf{charge}} = 1\text{sec}$ period). Cümülön-D is able to consider this nuance, because our market price model accounts for periodicity and we condition the price transition and expected charge functions on starting time and price.

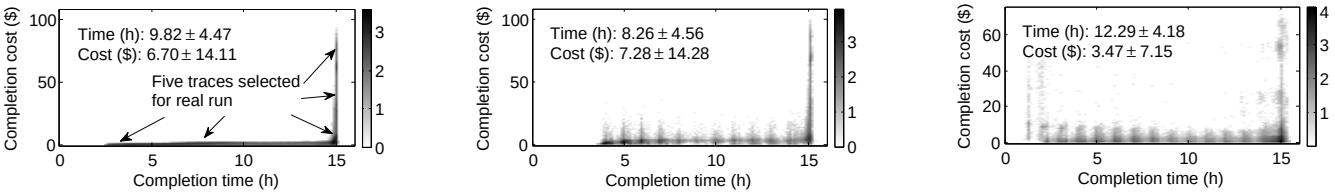
6.2 Evaluating a Cookbook

We now turn to workflow *RSVD-1* and evaluate Cümülön-D using simulation over the large test set of price traces for $S_{\$0.20}$. The deadline is 15 hours, and we choose $\gamma = 0$ (i.e., minimizing expected cost alone) in order to compare with *Oracle* (for which cost variance does not apply). The default settings of $\tau_{\mathsf{charge}} = 1\text{sec}$ and $\tau_{\mathsf{opt}} = 1\text{hr}$ are used. We apply Cümülön-D's policy $\mathfrak{L}_0$ both in dynamic and strict modes (recall Section 5.4). Figure 2 shows the scatter plots of the completion cost vs. time over the 100,000 test traces for the three approaches.



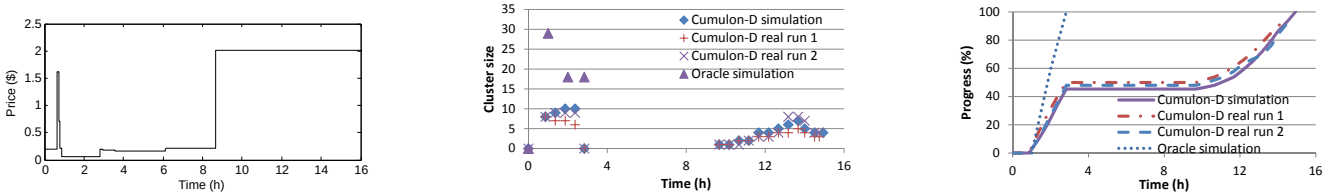
(a) $t = 10$ (b) $u = \$0.25$ (c) $\varpi = 310$

Figure 1: Visualization of Cümülön-D's cookbook, for a synthetic workflow with three matrix multiplies, over three 2-d slices of the 4-d state space. In all three slices, $n_{t-1} = 0$. Each slice is visualized as two density plots, one for the optimal action (new cluster size), and one for the expected remaining cost.



(a) Cümülön-D in dynamic mode (b) Cümülön-D in strict mode (c) *Oracle*

Figure 2: Completion time and cost distributions for simulated executions of *RSVD-1* under $S_{\$0.20}$, with $d = 15$hrs. The densities are counts in the test set in $\log_{10}$-scale. The numbers in the legends are formatted as "*average* $\pm$ *standard deviation*."



(a) Price trace (b) Cluster size over time (c) Execution progress

Figure 3: Timeline for Trace 2, Table 1, showing simulations of Cümülön-D, *Oracle*, and two real runs of Cümülön-D. Cümülön-D runs in dynamic mode.

Table 1: Comparison of simulated vs. actual completion times and costs of *RSVD-1*, using Cümülön-D's dynamic mode, for the five price traces marked in Figure 2a. Settings are same as Figure 2a.

	Trace 1		Trace 2		Trace 3		Trace 4		Trace 5	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
Simulation	14.87	75.77	14.93	39.49	14.89	3.44	7.84	2.89	3.14	0.77
Real run 1	14.83	66.56	14.74	29.83	14.40	2.67	7.23	2.03	2.52	0.86
Real run 2	15.09	79.32	14.74	38.56	14.50	3.31	8.11	3.48	2.25	0.84

For Cümülön-D under dynamic mode (Figure 2a), most density concentrates in a low-cost region with completion time ranging from 3 to 15 hours. These correspond to the "typical" price traces where there are enough times with low market prices before the deadline for Cümülön-D to finish with cheap machines. However, there is also a high-cost "tail" with completion time near the deadline. These are the "unlucky" cases where Cümülön-D chose to wait for a later price drop, but the market price stayed high, leaving Cümülön-D no choice but to go for a larger cluster of expensive machines in order to finish on time. While the worst case is very costly (around \$100), we note that such cases are rare, as evidenced by the reasonable standard deviation numbers reported in the legend. We also note that for many of such cases, there is little we can do under a deadline—the worst cost under *Oracle* is also more than \$75, even though it has perfect knowledge of the future.

To validate our execution simulator, we select five representative regions shown in Figure 2a, and randomly pick a test price trace for each region. Next, we run the Cümülön-D on Amazon EC2 against the test trace. Table 1 summarizes the result. Although there are some variations in the detailed timings during the course of execution (more on these in Section 6.3), the overall completion times and costs are quite consistent between simulated and real runs.

Figure 2b shows the simulated results when Cümülön-D applies its cookbook in strict mode, where it adjusts the cluster size only at hourly boundaries. This hourly behavior creates the artifact of completion times clustering around hourly boundaries. Compared with Figure 2a, density around the deadline is lower, because under strict mode, whenever Cümülön-D gets some machines, they will be kept for an hour (τ_{opt}) in this case, which tends to leave less work and hence lower risk in the last hour. However, since strict mode misses many fine-grained adaptation opportunities, the overall mean completion cost rises from \$6.70 to \$7.28.

Finally, Figure 2c shows the performance of *Oracle*. It achieves an overall mean completion cost of \$3.47, about a factor of two better than Cümülön-D. This difference represents a price we pay for uncertainty, as *Oracle* gives a practically unattainable lower bound by assuming perfect knowledge of the future. We make two more observations. First, the mean completion time for *Oracle* is longer than Cümülön-D, which is not a disadvantage because finishing early has no reward as long as the deadline is met. However, it does reflect the more aggressive behavior by *Oracle*—it often waits for the lowest price to act, without guessing or balancing the risk of uncertainty like Cümülön-D does. Second, as mentioned earlier, there are "unlucky" price traces for which even *Oracle* incurs very high costs. These cases tend to be more spread over different completion times, because, unlike Cümülön-D, *Oracle* never waits in hope for the situation to improve.

6.3 Zooming in on a Trace

To better understand how Cümülön-D adapts dynamically, we zoom in on one of the five test traces in Table 1. Figure 3 shows the price trace as well as changes to the cluster size and execution progress over time. This trace represents an unlucky case that highlights the price of uncertainty. With the high starting price of \$0.20, Cümülön-D (in simulation) starts with no machines and waits for the price to drop. When the price does drop to \$0.068 around 0.9hr, Cümülön-D decides to bid for eight machines and starts to make progress. However, around 2.8hr, the price rises to \$0.190. Having finished almost half of the work, Cümülön-D decides to give up all machines and wait again. Unfortunately, the price unchar-

acteristically stays high for six hours, and at time 8.7hr, it climbs even higher to $2.010. Around 9.7hr, considering that the deadline of 15hr is approaching and there is still a fair amount of work left, Cümülön-D decides to resume execution with one machine under the high price. As the time goes by, the pressure of deadline, or the risk of having to use a expensive, suboptimally large cluster to finish on time, continues to increase. Therefore, Cümülön-D gradually increases the cluster size to offset this pressure. In the end, Cümülön-D completes at 14.9hr, incurring a total cost of $39.49.

Oracle, however, acts differently. Knowing that the price only stays low from 0.9hr to 2.8hr, it pushes all work into this duration, even though the required cluster is so large that it is not cost-effective. Using 29 machines for the first hour and 18 for the second, *Oracle* finishes execution at 2.8hr, incurring a total cost of only $3.23. Perfect knowledge of the future gives *Oracle* a huge advantage in this specific case.

Figure 3 also compares the simulated execution of Cümülön-D with two real ones. The same cookbook is applied in the same way; differences only come from natural variations in real execution times and errors in Cümülön-D's cost models. As we can see in Figures 3b and 3c, there are slight deviations in both adaptation actions and execution progress throughout the timelines. However, the final results (Table 1) and the timelines are close. This example illustrates the self-correcting nature of Cümülön-D's cookbook. When execution turns out slower (faster) than expected, it leads to a state where a bigger (smaller, respectively) cluster may be favored next, which helps to bring execution back in line with the cookbook's expectation. This feature of Cümülön-D hence offers some protection against performance variations and model inaccuracies.

6.4 Effect of Deadline and Risk Tolerance

Next, we study how the user-specified deadline (d) and risk tolerance (γ in $\mathfrak{L}_\gamma$) impact Cümülön-D. We use workflow *GNMF* with two iterations. In Figure 4, we use the default of $\gamma = 0.1$ and vary the deadline from 1 to 30 hours. The market scenario is $S_{\$0.02}$. For each deadline, we obtain Cümülön-D's cookbook and plot the mean and standard deviation of costs under this cookbook over 100,000 simulations. As we relax the deadline, both mean and standard deviation drop; intuitively, we can afford to wait for more low-price opportunities and complete the work with cheaper machines.

In Figure 5, we fix the deadline at 15hr, and vary γ from 0 (i.e., minimizing expected cost alone) to 1 (minimizing the sum of expectation and standard deviation). The market scenario is $S_{\$0.2}$ (different from the above). As γ increases, we put more weight on reducing variance, leading to a more conservative cookbook that produces lower cost variance but potentially higher mean, as shown in the cost plot on left. A conservative cookbook generally prefers doing more work in the current time step rather than leaving it to the less certain future. Hence, cookbooks with higher γ tend to complete execution earlier, as shown in the time plot on right.

Comparing the data points at $d = 15$hr in Figure 4 (with $S_{\$0.02}$) and those at $\gamma = 0.1$ in Figure 5 (with $S_{\$0.2}$), we see that when the starting market price is higher ($0.20 vs. $0.02), both the mean and standard deviation of costs are higher, which is intuitive.

6.5 Hourly Charging and Actuation Delays

We now turn to experiments with setups closer to the current practice of Amazon EC2. We change τ_{charge} from our earlier setting of 1sec to 1hr; if a machine is terminated by the provider, its last partial hour is free. The workload is *RSVD-1*, with a 15hr deadline. The market scenario is $S_{\$0.02}$. For Figure 6, we further vary the actuation delays $\delta_{\triangleright} = \delta_{\square}$ from 0 to 1000sec. As discussed in Section 5.3, with non-zero actuation delays, Cümülön-D uses skyline bidding instead of ∞-bidding to guard against possible price spikes. We compare the simulated costs of Cümülön-D under these two bidding strategies (note that ∞-bidding is also employed in [9]). Figure 6 also shows the performance of *Oracle* as a theoretical lower bound; it is a flat line because *Oracle*, with perfect knowledge of the future, is unaffected by actuation delays. As we can see, longer market delays make Cümülön-D perform worse. Intuitively, they translate to higher uncertainty between the time of a decision and the time it is carried out, and hence more to pay.

The trade-off between skyline bidding and ∞-bidding is more intricate. As Figure 6 shows, when the delays are short (under 4 minutes), ∞-bidding has slightly lower mean costs, but as delays become longer, skyline bidding becomes progressively better. There are two factors in play here. First, as discussed in Section 5.3, skyline bidding guards against sudden price changes during actuation delays, and hence can lower cost. Second, and on the other hand, a temporary spike hike occurring after the start of a charging period may cause termination of machines under skyline bidding; even though we do not pay for this period, we cannot retain these machines, potentially costing us more later. In contrast, ∞-bidding allows these machines to be retained until the end of the charging period at no extra cost. In this case, due to the long charging period ($\tau_{\text{charge}} = 1$hr), the effect of the second factor becomes more pronounced, thereby helping ∞-bidding. In general, however, skyline bidding is safer and less sensitive to actuation delays, which is why we make it the default for Cümülön-D.

6.6 Skyline vs. Fixed-Price Bidding

Continuing with the evaluation of skyline bidding started by the last experiment, we now study how it compares with other strategies for setting the bid price (besides ∞). Specifically, we consider *fixed-price* bidding: here, we follow the same actions suggested by the cookbook, but when bidding, instead of using the skyline strategy, we set the bid price to some fixed value. The only exception is when bidding within the last τ_{opt} period before the deadline, we always bid ∞ to secure machines needed to complete the work.

We use the same experiment setup as in Figure 6, but fix the actuation delays at 10 minutes. We test fixed-price bidding with different bid prices from $0.10 to $1.50, and plot the mean of simulated completion costs for each bid price in Figure 7. The results of skyline bidding and ∞-bidding are shown as horizontal lines, because they are independent of the setting for fixed-price bidding.

As we can see, executions are costly when we use a low fixed bid price, because we have trouble getting or holding on to machines. This problem is alleviated when we increase the fixed bid price. However, beyond a certain point, the cost starts to rise, and eventually converges to that of ∞-bidding. The reason is that with high bid prices, we are more susceptible to sudden price hikes.

More importantly, skyline bidding outperforms fixed-price bidding regardless of its setting. This observation illustrates the power of adaptation. Not only does the optimal bid price depend on the workflow and market condition, it is also a "moving target" as execution progresses toward the deadline. Skyline bidding captures this dynamic behavior while fixed-price bidding cannot.

6.7 Cümülön-D vs. Fixed and DBA

We now turn to comparison between Cümülön-D and previous work. Optimal bidding strategies have been studied in the past, but most of them are not dynamic (see Section 7 for discussion). A straightforward way of applying these strategies in a dynamic setting is the following, which we call *Fixed*: once we have determined a bid price and a cluster size upfront, we will always bid for the given

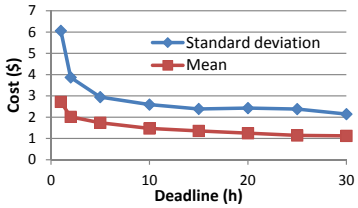


Figure 4: Mean and standard deviation of simulated costs for two iterations of *GNMF* under $S_{\$0.02}$, with $\gamma = 0.1$ and varying deadline.

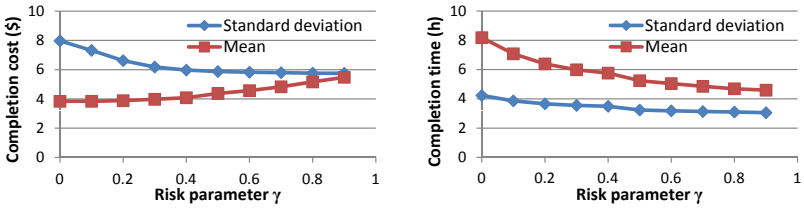


Figure 5: Mean and standard deviation of simulated costs (left) and times (right) for two iterations of *GNMF* under $S_{\$0.2}$, with a 15hr deadline and varying γ.

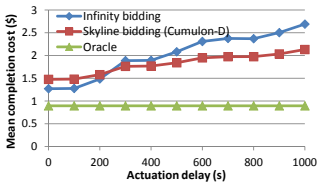


Figure 6: Comparison of ∞ and skyline bidding in Cümülön-D with *Oracle*. *RSVD-1* under 15hr deadline and market scenario $S_{\$0.02}$, with $\tau_{\text{charge}} = 1$hr and varying $\delta_{\triangleright} = \delta_{\square}$.

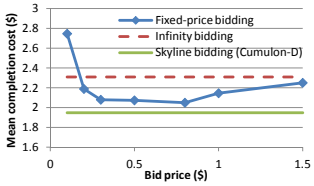


Figure 7: Comparison of skyline, ∞, and fixed-price bidding in Cümülön-D. Same workflow and settings as Figure 6, except $\delta_{\triangleright} = \delta_{\square} =$ 10min.

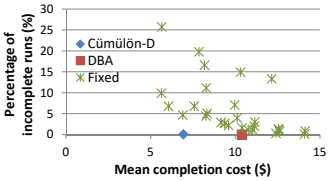


Figure 8: Comparison of Cümülön-D with *Fixed* (under various settings) and *DBA*, in terms of mean cost and incomplete rate. *RSVD-1* under 15hr deadline and market scenario $S_{\$0.2}$, with $\tau_{\text{charge}} = 1$hr and $\delta_{\triangleright} = \delta_{\square} = 10$min.

Table 2: Mean completion costs of *RSVD-1* for Cümülön-D, Cümülön-D+, and *Fixed*+, under two market scenarios. Other settings remain the same as in Figure 8. As a reference, the optimal cost of using only non-spot machines is $5.22.

	$S_{\$0.2}$	$S_{\$0.02}$
Cümülön-D (spot-only)	$6.80	$1.30
Cümülön-D+ (hybrid)	$2.53	$1.19
Fixed+ (hybrid)	$3.38	$1.96

number of machines with the given bid price, when the workflow starts and whenever the machines are terminated by the provider.

Besides *Fixed*, we also compare with *DBA* [18], which dynamically adjusts the bidding price. Taking advantage of Amazon's hourly charging scheme, *DBA* always goes for a cluster that allows the work to be completed within an hour. *DBA* starts with a low bid price; if that price is too low currently or if the cluster is terminated (in which case no charge is incurred because an hour has not passed), *DBA* tries again later with an increased bid price. *DBA* was shown to be optimal, assuming perfect (linear) speedup functions.

Again, we use the *RSVD-1* workflow with a 15hr deadline. The market scenario is $S_{\$0.2}$, with $\tau_{\text{charge}} = 1$hr and 10min actuation delays. For *Fixed*, to cover all possibilities for the optimal strategy, we experiment with all combinations of reasonable bid prices (from $0.05 to $1.00) and bid sizes (from 5 to 80 machines). For *DBA*, we used a fixed bid size of 60 machines because this number of machines can finish *RSVD-1* within an hour with high probability. For both *Fixed* and *DBA*, we make an exception in order to improve the probability of completion: if we need to bid within one hour before the deadline, we use a bid price of ∞ to secure the cluster.

Figure 8 shows the results as a scatter plot of the mean cost achieved and the percentage of the 100,000 test traces for which the workflow failed to complete by the deadline. As we can see, Cümülön-D clearly outperforms the other two approaches. The points for *Fixed* present a trade-off between the mean cost and ability to meet the deadline—lower bid prices and bid sizes tend to lead to lower costs (because of lower average price over time and higher cost-effectiveness, respectively), but they also less likely to meet the deadline. A relatively good trade-off is achieved by always bidding for 20 machines at $0.50, which results in a mean cost of $9.87 and a near-zero chance of missing the deadline. *Fixed* is soundly beaten by Cümülön-D's mean cost of $6.62, because by adapting to the evolving market and execution state, Cümülön-D does a better job at lowering cost and staying on schedule.

In comparison, *DBA* achieves a mean cost of $10.59, even worse than *Fixed* under certain settings. The key issue is *DBA*'s unrealistic assumption of perfect speedup. Even though in this case τ_{charge} is long (one hour) and the workflow (*RSVD-1*) is highly parallelizable, cramming all work into one hour with a large cluster is suboptimal. In situations where τ_{charge} is short (e.g., Google) or the work is less parallelizable, *DBA* cannot be applied at all.

6.8 Spot-Only vs. Hybrid Cluster

Cümülön-D uses a spot-only cluster. In [6], we show how to extend Cümülön-D to Cümülön-D+, which can use a hybrid cluster consisting of both spot and non-spot machines. We now experimentally study how this additional flexibility helps, and how Cümülön-D+ compares with recent work on *Dyna* [21], which also uses both spot and non-spot machines. A direct comparison with *Dyna* is difficult, because it was designed to exploit multiple markets where machines with better configurations could be potentially used at lower costs. For our setting where all spot and non-spot machines are of the same type, we have designed a surrogate strategy called *Fixed*+: whenever the market price is below the fixed, non-spot price, *Fixed*+ bids for spot machines at the non-spot price; whenever the market price rises above the non-spot price, *Fixed*+ simply falls back to using non-spot machines. We also enforce delayed release (Section 5.4) so that *Fixed*+ never releases a machine without fully using its paid hour (except for spot termination caused by market price increase). In the spirit of *Dyna*, *Fixed*+ does not consider bidding for more machines than the optimal non-spot cluster. (Please refer to [6] for more detailed discussion on why we think *Fixed*+ is a reasonable approximation to *Dyna* in our setting.)

Again, we use the *RSVD-1* workflow with a 15hr deadline. We consider market scenarios $S_{\$0.02}$ and $S_{\$0.2}$, with $\tau_{\text{charge}} = 1$hr and 10min actuation delays, and the fixed, non-spot price is $0.145. The optimal non-spot cluster size turns out to be 3, so *Fixed*+ will bid for 3 spot machines as well. We compare the mean completion costs for Cümülön-D (spot-only), Cümülön-D+, and *Fixed*+ in Table 2. As we can see, Cümülön-D+ consistently outperforms Cümülön-D because of the additional option of using non-spot machines; it also consistently outperforms *Fixed*+ thanks to its more dynamic bidding strategy. Specifically, in $S_{\$0.2}$ where spot price starts very high, rather than starting to execute on non-spot machines as *Fixed*+ does, Cümülön-D+ decides to wait, either until spot price drops, or deadline approaches (in which case it still has the option of acquiring more than the optimal number of non-spot

machines to ensure completion). In contrast, in $S_{\$0.02}$ where spot price starts low (which is the more common scenario), Cümülön-D+ is smart to go for more spot machines than *Fixed+* to reduce cost. Interestingly, even spot-only Cümülön-D is able to beat *Fixed+* in this case despite Cümülön-D's lack of the non-spot option.

7 Related Work

One approach for handling the unreliability of spot machines is to use a combination of spot and non-spot machines. Examples of this approach from previous work include *Cap3* [7], *Dyna* [21], *Qubole*, Amazon's Elastic MapReduce, and Cümülön [5] (which we have discussed extensively in Section 2). A second approach, which this paper also adopts, relies on separate stable storage. For example, [13, 15, 17] proposed various checkpointing and migration techniques for execution on spot machines, but they relied on heuristic bidding strategies with no optimality guarantees. Going a step further, [2, 12, 14, 20] built models that suggest optimal bid prices, but they are fixed for the entire workload. On the other hand, [2, 18, 7, 20] considered workloads parallelizable on a varying number of machines, but they only picked a fixed cluster size during execution. As we have shown in Section 6.7, Cümülön-D, with its dynamic adaptation, works better than these *Fixed* strategies. *DBA* [18] introduced the method of dynamic bidding during execution, but only did so for setting bid prices. As discussed in Section 6.7, their strategy was Amazon-specific and relied on the unrealistic assumption of perfect speedup. Not only is Cümülön-D more general, but it also outperforms *DBA* with more informed and flexible adaptation, and better modeling of factors influencing cost. *Dyna* [21] had a different focus on exploiting multiple markets, but was otherwise not as adaptive as Cümülön-D; it did not consider dynamically adjusting the unit of parallel execution or acting proactively based on the current market condition. As shown in Section 6.8, Cümülön-D, when extended to consider the use of non-spot machines, consistently outperforms *Fixed+*, which approximates *Dyna*'s strategy in our setting.

On the note of generality, it is worth noting that most previous work was specific to Amazon (e.g., assuming $\tau_{\text{charge}} = 1\text{hr}$); some further assumed market price only changing at hour boundaries, or ignored actuation delays (e.g., [14, 9] used this assumption to justify ∞-bidding). It is unclear how to generalize these previous approaches. Cümülön-D offers a more general solution framework that can handle alternative and more realistic market settings.

While Cümülön-D and its precursors [4, 5] aim at helping individual uses of spot markets in a public cloud, others have considered the problem from different angles. [1, 8] focused on modeling of Amazon's spot market prices. Such models can be readily incorporated into Cümülön-D (Section 5.1). [16, 19] studied how to maximize a provider's profit by supplying and pricing spot markets optimally. Our work is complementary; a provider can benefit from understanding how users bid intelligently and dynamically.

8 Conclusion and Future Work

In this paper, we have presented Cümülön-D, a system to help users run matrix-based data analytics using spot markets in a public cloud. Cümülön-D lets its user specify the input program declaratively, set a deadline, and provide the desired balance between minimizing expected cost and variance. Cümülön-D uses a simple yet flexible system architecture to support continuous, proactive adaptation of execution to a dynamic market. We model the dynamic optimization problem in a principled, general way as an MDP, accounting for various practical details that have often been ignored in previous work. We make careful choices on the level of details in modeling and optimization, and devise methods to speed up optimization and run-time application of its result. Experiments on Amazon EC2 illustrate the "price of uncertainty" and demonstrate Cümülön-D's advantages over previous approaches.

Although developed in the context of supporting matrix-based workflows, many of our models and techniques are generally applicable to other parallelizable workloads. For example, database queries are declaratively specified using a set of standard operators, and are amenable to data-parallel processing. Since our workloads have relatively predictable performance, Cümülön-D currently focuses its optimization on the variability of market price (which is the dominant source of uncertainty), and relies on applying the cookbook dynamically to cope with the variability of performance at run time. To extend Cümülön-D to less predictable or even black-box workloads, we will have to further consider performance uncertainty in optimization. For example, performance of database workloads can be highly sensitive to data characteristics. If workloads run long or repeatedly, it may be a good idea to dynamically adapt the performance models as well during executions. We leave these generalizations to future work. Another promising direction is to extend Cümülön-D to consider using a heterogeneous cluster consisting of machines of different types from different markets.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *CloudCom* 2011.

[2] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under SLA constraints. *MASCOTS* 2010.

[3] A. B. Downey. A model for speedup of parallel programs. Technical report, University of California, Berkeley, 1997.

[4] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. *SIGMOD* 2013.

[5] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumülön: Matrix-based data analytics in the cloud with spot instances. *PVLDB*, 9(3):156–167, Sept. 2015. Detailed technical report version: `http://db.cs.duke.edu/papers/pvldb15-HuangJarrettEtAl-cumulon-spot.pdf`.

[6] B. Huang and J. Yang. Cumülön-D: Data analytics in a dynamic spot market. Technical report, Duke University, 2016. `http://db.cs.duke.edu/papers/2016-HuangYang-cumulon_dynamic_spot.pdf`.

[7] H. Huang, L. Wang, B. Tak, L. Wang, and C. Tang. CAP3: a cloud auto-provisioning framework for parallel processing using on-demand and spot instances. *Cloud* 2013.

[8] B. Javadi, R. K. Thulasiram, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. *Utility and Cloud Computing* 2011.

[9] S. Khatua and N. Mukherjee. Application-centric resource provisioning for Amazon EC2 spot instances. *EuroPar* 2013.

[10] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. *NIPS* 2000.

[11] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, Aug. 2009.

[12] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. *INFOCOM* 2012.

[13] M. Taifi, J. Y. Shi, and A. Khreishah. SpotMPI: A framework for auction-based HPC computing using amazon spot instances. *ICA3PP* 2011.

[14] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. *Cloud* 2012.

[15] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. *Advanced Information Networking and Applications* 2012.

[16] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. *ICDCS* 2013.

[17] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Trans. Services Computing*, 5(4):512–524, 2012.

[18] M. Zafer, Y. Song, and K. Lee. Optimal bids for spot VMs in a cloud for deadline constrained jobs. *Cloud* 2012.

[19] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. *Utility and Cloud Computing* 2011.

[20] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang. How to bid the cloud. *SIGCOMM* 2015.

[21] A. C. Zhou, B. He, and C. Liu. Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds. *IEEE Trans. Cloud Computing*, 4(1):34–48, 2016.