Proteus: agile ML elasticity through tiered reliability in dynamic resource markets

Aaron Harlap[§] Alexey Tumanov[†] Andrew Chung[§]
Gregory R. Ganger[§] Phillip B. Gibbons[§]

[§]**Carnegie Mellon University** [†]**UC Berkeley**

Abstract

Many shared computing clusters allow users to utilize excess idle resources at lower cost or priority, with the proviso that some or all may be taken away at any time. But, exploiting such dynamic resource availability and the often fluctuating markets for them requires agile elasticity and effective acquisition strategies. Proteus aggressively exploits such transient revocable resources to do machine learning (ML) cheaper and/or faster. Its parameter server framework, AgileML, efficiently adapts to bulk additions and revocations of transient machines, through a novel 3-stage active-backup approach, with minimal use of more costly non-transient resources. Its BidBrain component adaptively allocates resources from multiple EC2 spot markets to minimize average cost per work as transient resource availability and cost change over time. Our evaluations show that Proteus reduces cost by 85% relative to non-transient pricing, and by 43% relative to previous approaches, while simultaneously reducing runtimes by up to 37%.

1. Introduction

Statistical machine learning (ML) has become a primary data processing activity for business, science, and online services that attempt to extract insight from observation (training) data. Generally speaking, ML algorithms iteratively process training data to determine model parameter values that make an expert-chosen statistical model best fit it. Once fit (trained), such models can predict outcomes for new data items based on selected characteristics (e.g., for recommendation systems), correlate effects with causes (e.g., for genomic analyses of diseases), label new media files (e.g., which ones are funny cat videos?), and so on.

ML model training is often quite resource intensive, requiring hours on 10s or 100s of cores to converge on a solution. As such, it should exploit any available extra resources or cost savings. Many modern compute infrastructures offer a great opportunity: transient availability of cheap but revocable resources. For example, Amazon EC2's spot market and Google Compute Engine's preemptible instances often allow customers to use machines at a 70–80% discount [2] off the regular price, but with the risk that they can be taken away at any time. Many cluster schedulers similarly allow lower-priority jobs to use resources provisioned but not currently needed to support business-critical activities, taking the resources away when those activities need them. ML model training could often be faster and/or cheaper by aggressively exploiting such revocable resources.

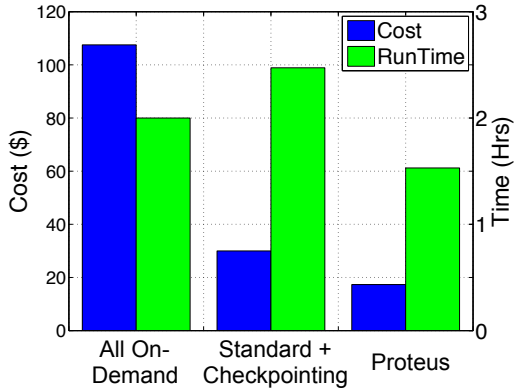


Figure 1: Cost and time benefits of Proteus. This graph shows average cost (left axis) and runtime (right axis) for running the *MLR* application (see Section 6.2) on the AWS EC2 US-EAST-1 Region. The three configurations shown are: 128 on-demand machines, using 128 spot market machines with checkpoint/restart for dealing with evictions and a standard strategy of bidding the on-demand price, and Proteus using 3 on-demand and up to 189 spot market machines. Proteus reduces cost by 85% relative to using all on-demand machines and by ≈50% relative to the checkpointing-based scheme. Full experimental details can be found in Section 6.

Unfortunately, efficient modern frameworks for parallel ML, such as TensorFlow [6], MxNet [9], and Petuum [37], are not designed to exploit transient resources. Most use a *parameter server* architecture, in which parallel workers process training data independently and use a specialized key-value store for shared state, offloading communication and synchronization challenges from ML app writers [10, 22, 25]. Like MPI-based HPC applications, these frameworks generally assume that the set of machines is fixed, optimizing aggressively for the no machine failure and

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…5.00

DOI: http://dx.doi.org/10.1145/3064176.3064182

no machine change case (and restarting the entire computation from the last checkpoint on any failure). So, using revocable machines risks significant rollback overhead, and adding newly available machines to a running computation is often not supported.

This paper describes Proteus—a parameter server system that combines agile elasticity with aggressive acquisition strategies to exploit transient revocable resources. Figure 1 illustrates the benefits for one ML example on Amazon EC2. Using three on-demand instances and up to 189 spot market instances, Proteus reduces cost by 85% when compared to using only on-demand instances, even when accounting for spot market variation and revocations, while running 24% faster. Compared to using a standard bidding strategy with a checkpointing-based approach (i.e., run on spot market machines and checkpoint regularly to retain progress if evicted [19, 30, 31]), Proteus reduces cost by ≈50% and runtimes by 32–43%, winning by avoiding checkpoint overheads, reducing restart delays, and exploiting spot market properties.

Proteus consists of two principal components: AgileML and BidBrain. The AgileML parameter-server system achieves agile elasticity by explicitly combining multiple reliability tiers, with core functionality residing on more reliable resources (non-transient resources, like on-demand instances on EC2) and most work performed on transient resources. This allows quick and efficient scaling, including expansion when resources become available and bulk extraction of revoked transient resources without big delays for rolling back state or recovering lost work. AgileML transitions among different modes/stages as transient resources come and go. When the ratio of transient to non-transient is small (e.g., 2-to-1), it simply distributes the parameter server functionality across only the non-transient machines, instead of across all machines, as is the usual approach. For much larger ratios (e.g., 63-to-1), the one non-transient machine would be a bottleneck in that configuration. In that case, AgileML uses non-transient machine(s) as on-line backup parameter servers (BackupPSs) to active primary parameter servers (ActivePSs) that run on transient machines. Updates are coalesced and streamed from actives to backups in the background at a rate that the network bandwidth accommodates.

BidBrain is Proteus' resource allocation component that decides when to acquire and yield transient resources. BidBrain is specialized for EC2, exploiting spot market characteristics in its policies, but its general approach would apply to other environments with transient resources (e.g., private clusters or other cloud costing models). It monitors current market prices for multiple instance types, which move relatively independently, and bids on new resources when their addition to the current footprint would increase work per dollar. Similarly, resources near the end of an hour may be released if they have become less cost-effective relative to others. As part of its considerations, BidBrain estimates the probability of getting free compute due to instance revocation within the billing hour (with later in the hour being better than earlier) for different bids and spot market conditions. Simultaneously considering costs (e.g., revocation and scaling inefficiencies) and benefits (e.g., cheaper new resources), BidBrain finds a happy medium between aggressive bidding on transient resources and more conservative choices.

Experiments with three real ML applications confirm that Proteus achieves significant cost and runtime reductions. AgileML's elasticity support introduces negligible performance overhead, scales well, and suffers minimal disruption during bulk addition or removal of transient machines. In breaking down the sources of benefit, we find that both the agile elasticity of AgileML and the aggressive policies of BidBrain are needed—using either one alone (e.g., BidBrain with checkpointing instead of AgileML) achieves half or less of the cost and runtime savings.

This paper makes four primary contributions. First, it describes the first parameter server ML framework (Proteus) designed to elastically scale with bulk additions and revocations of transient machines. Second, it describes an adaptive architecture+algorithm (AgileML) for exploiting multiple tiers of machine reliability (i) to more agilely resize in the face of such changes and (ii) to balance work given different ratios of non-transient to transient resources. Third, it describes a new resource manager (BidBrain) that aggressively exploits EC2 spot market properties to achieve major cost savings. Fourth, it presents results from experiments and analyses showing that aggressive multi-tier exploitation of transient machines is both possible and beneficial, reducing costs and runtimes significantly.

2. Motivation and Related Work

This section overviews efficient frameworks for ML model training, transient availability of revocable cluster/cloud resources, and what is needed for the former to exploit the latter.

2.1 ML Model Training Frameworks

Statistical machine learning algorithms determine parameter values that make a chosen statistical model fit a set of training data. Most modern ML approaches rely on *iterative convergent algorithms*, such as stochastic gradient descent (SGD), to determine these model parameter values. Such algorithms start with an initial solution guess and refine it over a number of iterations on the training data, improving an explicit goodness-of-solution objective function until sufficient convergence or goodness has been reached.

ML model training is resource-intensive, especially as model precision grows, and commonly requires parallel execution to complete in reasonable time. For example, the ML applications used for experiments reported in Section 6 scale well with machine count yet still require multiple hours even when using 10s of multicore machines.

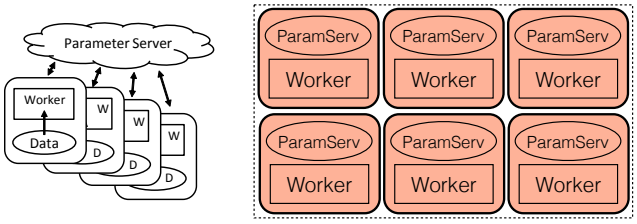


Figure 2: Traditional Parameter Server Architecture. The left figure illustrates the logical architecture, and the right figure illustrates that the parameter server is usually sharded across the same machines as the workers.

Although iterative convergent ML algorithms can be built as BSP-style sequences of bag-of-task computations, such as map-reduce jobs, on systems like Hadoop [4] or Spark [38], such implementations are inefficient. They preclude several specializations, including flexible consistency models [11, 25, 27], cross-iteration optimization [12], and early exchange of updates [36]. Collectively, these specializations can provide an order of magnitude or more increase in training efficiency.

Parameter Server Architectures. The most efficient modern frameworks for parallel ML use a *parameter server* architecture,[1] which allows programmers to easily build scalable ML algorithms while benefiting from such specializations [10, 20, 22, 25]. As a result, open source ML model training frameworks like TensorFlow [6], MxNet [9], Petuum [37] and many proprietary systems use variants of this architecture.

Fig. 2 illustrates a simple parameter server system. Commonly, training data is partitioned among the worker threads that execute the ML application code for adjusting model parameter values. The only state shared among worker threads is the current parameter values, and they are kept in a specialized key-value store called the "parameter server." Worker threads process their assigned training data and use simple `read-param` and `update-param` methods to check and apply deltas to parameter values. The value type is usually application-defined, but must be serializable and have a commutative and associative aggregation function so that updates from different worker threads can be applied in any order. For the ML applications used in this paper, the values are vectors and the aggregation function is component-wise add (+).

To reduce cross-machine traffic, parameter server implementations include a worker-side library that caches parameter values and buffers updates. While logically a single separate server (left side of Fig. 2), the parameter server is usually sharded across the same machines as worker threads (right side of Fig. 2), enabling it to scale with the computation power and aggregate memory and bandwidth used for training. Threads associated with the worker-side cache communicate with the appropriate server shards for each given value. Updates are write-back cached and sent (asynchronously) to the appropriate parameter server shards each iteration.

Given their resource-intensive nature, ML model training frameworks should be able to take advantage of any extra machines or potential cost savings available. Existing solutions that address ML framework elasticity include checkpointing [19, 30, 31] and Spark RDDs. Spark RDDs in particular allow fine-granularity Spark application checkpointing and rollback to handle resource revocations. Elasticity to transient resources can, thus, be achieved by relying on the fault tolerance mechanisms of RDDs. There are two problems with this, first, RDDs work well only for deterministic computation. Second, for highly correlated bulk revocations, the amount of recovery work approaches that of the checkpointing mechanisms (see Sec. 8). Our system, Proteus, significantly improves on the performance of checkpointing both in terms of cost and application runtime.

2.2 Dynamic Availability of Revocable Resources

Today's cluster infrastructures are increasingly dynamic, and working with transient resources on a regular basis is common. Resources may be available as temporarily-unused nodes on a revocable basis at a discount (in public pay-as-you-go clouds) or for lower-priority workloads (in shared corporate clusters). For both public clouds and mixed-purpose corporate clusters, lower intensity periods for business critical workloads create an opportunity for extra machines to be made available to other workloads. But, those machines may need to be taken back if business-critical workload intensity increases. This section describes how such machines are made available in several modern infrastructures.

Amazon AWS EC2 Spot Market. Amazon AWS EC2 [1] is a public cloud that allows customers to purchase time on virtualized machine resources. The traditional EC2 model, referred to as "on demand" because machines can be requested and released by customers at any time (though billing is based on an hourly granularity), involves paying a pre-determined fixed hourly rate to have guaranteed access to rented machine resources. Amazon also has a so-called "spot market" for machines, where machines are often available at a steep discount (e.g., 70–80% lower price) with the proviso that they can be taken back at any time. So, a customer who can exploit transient machines for their work can potentially save money and/or time.

The EC2 spot market design has interesting properties that affect customer savings and the likelihood of eviction. First, it is not a free market [7]. Customers specify their *bid prices* for a given machine class, but generally do not pay that amount. Instead, a customer is billed according to the

[1] A recent study [12] showed two parameter server systems (IterStore [12] and LazyTable [11]) outperform PowerGraph [16] significantly (factors of 10X and 2X, respectively) for collaborative filtering via sparse matrix factorization. The performance advantage of parameter server systems over bag-of-task systems, for such ML applications, is clarified by combining those results with a recent study showing that a highly-tuned Spark-based system called GraphX [17] approximately matches PowerGraph.

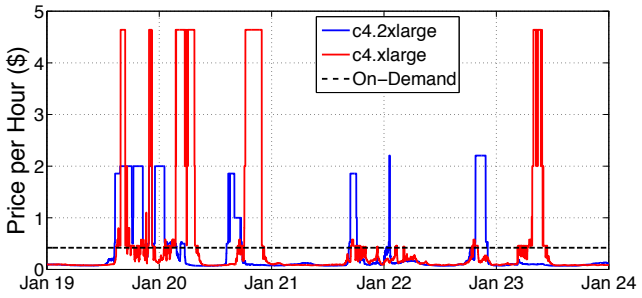


Figure 3: AWS spot prices over time. Spot prices for two classes of machines are shown for 6 days in January 2016. The unchanging on-demand price for c4.2xlarge is shown, and the values shown for c4.xlarge are doubled so that all three lines show the price for the same number of cores; c4.2xlarge machines have 8 cores and c4.xlarge machines have 4 cores.

current EC2-determined *spot price* for that machine class. Fig. 3 shows one example of spot price variability over a week, for two machine classes in an EC2 zone. Second, if a customer receives machine resources in response to their bid price, they will retain those resources until either they release them or the spot price rises above the customer's bid price. If the latter occurs, the customer is not billed for the current hour, but the resources are taken back from the customer. Third, EC2 does not guarantee any warning when resources are going to be revoked, but since 2015 EC2 has provided a two-minute warning prior to eviction in most cases. Fourth, once a customer submits a bid and receives a resource, the bid price cannot be changed. The bid can be canceled, if not yet granted, and a new bid price submitted. But, once the resource is granted, the bid price cannot be changed until the resource is terminated.

Google Preemptible Instances. Google Compute Engine (GCE) [3] offers revocable machine resources, called "preemptible instances", akin to those provided by the EC2 spot market. Google preemptible instances can be revoked at any time, as the name suggests, but differ from EC2's spot market in several ways. First, Google charges a fixed price of 70% less than the "on-demand" (non-revocable instance) price for the requested machine type. There is no price variability. Second, GCE offers a 30-second warning, rather than a 2-minute warning. Third, GCE limits preemptible instances to 24 hours.

Dynamic Resource Offers in Mixed-Function Corporate Clusters. Many corporate clusters serve a mix of online services, business critical batch analytics jobs (often with deadlines), and ad hoc jobs (often called "best effort") for application development, exploratory data analyses, etc. Business critical activities are usually given priority, but extra resources are often available for ad hoc jobs. Moreover, modern schedulers for such clusters, such as YARN [33] and Mesos [21], have mechanisms to offer recently-freed resources to already running jobs' "application master" components, allowing some of them (e.g., large map-reduce jobs) to elastically grow to higher performance levels by spreading work over more machines. But, these resources may subsequently be revoked if higher priority workloads intensify or additional jobs arrive [13, 33, 34].

2.3 Exploiting Transient Resources for ML

To exploit transient resources, ML frameworks need to couple agile elasticity with good resource allocation strategies. The elasticity must accommodate efficient bulk extraction of revoked transient resources with little-to-no warning, which is akin to correlated failures. Proteus addresses these needs. Section 3 describes AgileML, which is a parameter server system that exploits resource reliability tiers (e.g., stable EC2 on-demand instances and transient spot market instances) to achieve agile elasticity efficiently. Section 4 describes BidBrain, which uses an aggressive strategy of bidding on multiple spot markets to minimize the cost of training ML models. Section 5 describes how these two components are combined in Proteus.

3. AgileML Design

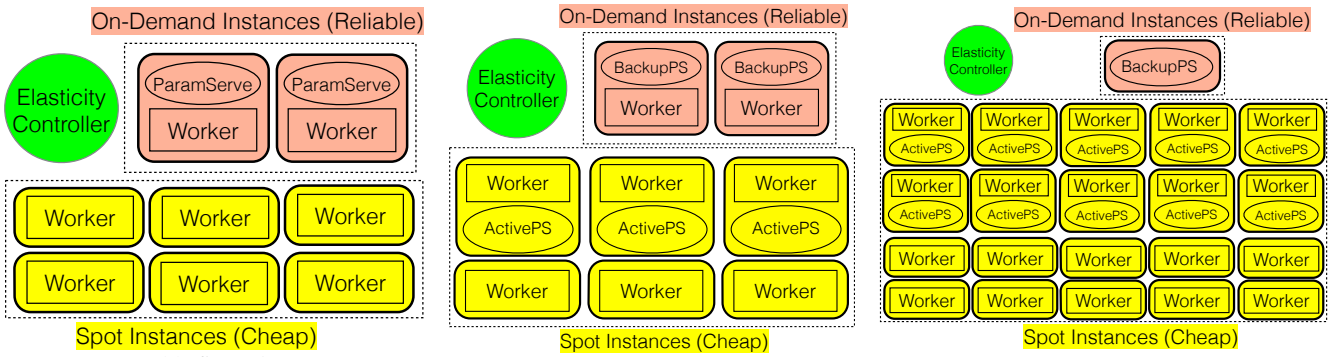
This section describes AgileML, Proteus' elastic ML training component. AgileML introduces the concept of "tiers of reliability" to organize resources into tiers based on their expected reliability (and associated cost) and deploy different functional components of the ML framework to different tiers. AgileML, which is implemented as a C++ library linked by an ML application using it, is built upon the parameter server architecture described in Sec. 2.1. Functional components include workers, parameter servers (ParamServs), and newly introduced Active and Backup parameter servers (Tab. 1). These solution state servers hold ML model parameter state and have different fault tolerance expectations. AgileML uses different combinations of these components to allow safe and agile exploitation of different quantities of transient resources.

3.1 Workers and Execution Management

During initialization, an ML application provides AgileML with an initial list of reliable and transient nodes to be used, the input data file path, several functions called by AgileML, and a stopping criterion.[2] During execution, AgileML consists of one process executing on each node. Each process then starts one worker thread per core. The worker threads execute the ML application code for model training—adjusting model parameters as a function of input (training) data and current solution state. Each worker thread operates on a disjoint subset of input data items. By default, input data is partitioned evenly amongst workers. Workers iterate on the data until reaching the stopping criteria.[3]

[2] The stopping criterion may be a number of iterations, an amount of time, or a determination of convergence.

[3] This is an over-simplification. For greater flexibility, AgileML actually provides a notion of a *clock of work* that gets executed on each iteration. It may be some number of data items (a "mini-batch" of an iteration) or some number of iterations.



(a) Stage 1 (b) Stage 2 (c) Stage 3

Figure 4: Three stages of AgileML architecture. Stage 1: ParamServs only on reliable machine. Stage 2: ActivePSs on transient and BackupPSs on reliable. Stage 3: No Workers on Reliable Machines.

3.2 Architecture

This section describes how AgileML uses reliability tiers and the mechanism of moving between its different stages of execution. At a high level, AgileML enables ML applications to run on a dynamic mix of reliable and transient machines, maintaining the state required for continued operation on reliable machines, while taking advantage of transient machine availability. AgileML uses three stages of system functionality partitioning in order to avoid bottlenecking the reliable nodes, as the ratio of transient to reliable resources grows.

Stage 1: Parameter Servers Only on Reliable Machines. For most ML applications including K-means, DNN, Logistic Regression, Sparse Coding, as well as MF, MLR, and LDA (Sec. 6.2), the workers are stateless, while the ParamServs contain the current solution state. AgileML's first stage spreads the parameter server across reliable machines only, using transient nodes only for workers, thereby taking advantage of these two primary levels of machine reliability. This has the effect of removing all solution state from transient machines. Fig. 4a illustrates a running example of eight AWS EC2 machines with six spot instances (transient) and two on-demand instances (reliable).

Pros: By removing parameter state from transient resources, AgileML is able to utilize them without losing progress when transient resources are revoked (or fail). Unlike a traditional parameter-server architecture, no checkpointing is required to assist with using transient resources.[4]

Cons: While stage 1 successfully removes state from transient resources, it causes a network bottleneck (Sec. 6.4) when the ratio of transient to reliable resources grows too large. With 60 transient and 4 reliable machines, for example, the network bottleneck to the ParamServs slows the *MF* application by over 85%. Limiting this ratio is undesirable, as it caps achievable savings/benefits from transient resources.

Stage 2: ActivePSs on Transient Machines and BackupPSs on Reliable Machines. For higher transient to reliable node ratio, AgileML switches to stage 2 (Fig. 4b). Stage 2 uses a primary-backup model for parameter servers, using transient nodes for an active server (ActivePS) and reliable nodes for the hot standby (BackupPS). This shifts the heavy network load from the few reliable resources to the many transient resources. Solution state is sharded across the set of ActivePS instances. Workers send all updates and read requests to the ActivePSs, which update their state and push updates in bulk to the BackupPSs. Solution state affected by transient node failures or evictions is recovered from BackupPSs. Stage 2 improves on stage 1 for higher transient-to-reliable ratios (Sec. 6.4) but loses to an all-reliable baseline by 2x for ratios exceeding 63:1.

Stage 3: No Workers on Reliable Machines. Workers colocated with BackupPSs on reliable machines were found to cause straggler effects at transient-to-reliable ratios beyond 15:1, causing Proteus' performance drop relative to the PS baseline. Stage 3 simply removes these workers (Fig. 4c), allowing AgileML to match all-reliable performance levels (Sec. 6.4). The optimal ratio threshold to switch to stage 3 depends on the network bandwidth, transient-to-reliable ratio and the size of the model.

Transitioning Between Stages. AgileML dynamically transitions between these stages to match the number of transient nodes available. Transitioning between stages 1 and 2 involves switching between a set of ParamServs and the active/backup PS pair. This process is described in Sec. 3.3.

ParamServs	Serve solution state for workers and always run on reliable resources
BackupPSs	Serve as a hot backup for solution state served by ActivePSs and always runs on reliable resources
ActivePSs	Serve solution state for workers, periodically pushing aggregated updates to BackupPSs, and run on transient resources

Table 1: Types of solution state servers used by AgileML

[4] In AgileML, there is benefit in checkpointing the reliable resources in case they fail, however as we show later in this section, this checkpointing has no overhead on system performance in stages 2 and 3.

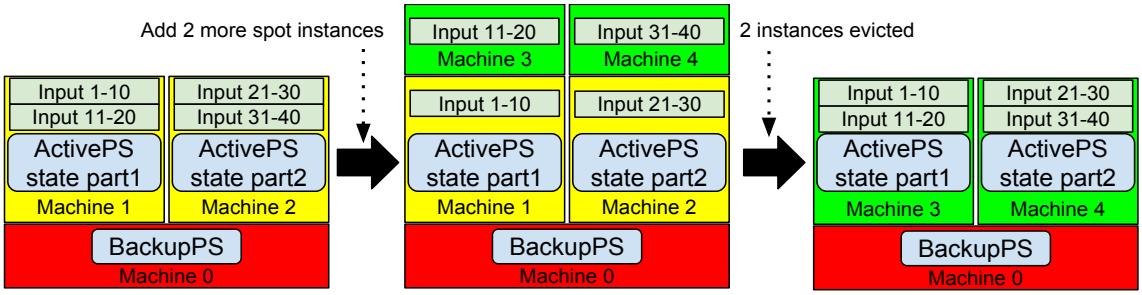


Figure 5: AgileML component and data transitions as resources are added and evicted. In this toy example, there are 40 pieces of input data. Initially, one on-demand Machine 0 runs BackupPS, and 2 spot instances (Machine 1,2) are processing $\frac{1}{2}$ of the input data each. 2 new spot instances (Machine 3,4) are added, at the same time, price, of the same type, and shown in the same color (we refer to these atomic sets as *allocations*, described in Sec. 4). Each new instance $\in \{3, 4\}$ loads $\frac{1}{2}$ of the input data, but works only on $\frac{1}{4}$ of it. An eviction of the 2 yellow spot instances triggers the second transition. The remaining spot instances assume ownership of the evicted input data with minimal delay.

When scaling up, workers are directed to send their requests to ActivePSs started in the background. When scaling down, ActivePSs push their updates to BackupPSs, which become ParamServs. The worker requests are then redirected to the ParamServs. This transition is done with minimal overhead in the background. Transitioning between stages 2 and 3 boils down to re-assigning work between reliable and transient resources. Scaling up, work is offloaded from workers on reliable nodes to workers on transient nodes, followed by shutting down the former workers. Transitioning back to stage 2 requires reassigning input data to reliable workers. This change of assignment incurs zero run-time overhead, as it involves just a single worker notification message.[5]

Elasticity Controller: This component of AgileML makes decisions to switch between stages based on the transient-to-reliable ratio and the network bandwidth. It is responsible for (a) tracking which resources are participating in ongoing computations, (b) assigning a subset of input data to each worker, and (c) starting new ActivePSs. On eviction, it re-shards the solution state and shuts down ActivePSs using policies discussed next.

3.3 Handling Elasticity: Policy and Mechanism

The toy example in Fig. 5 illustrates how AgileML handles adding and removing resources from an ongoing computation. We evaluate AgileML's effectiveness at handling such elasticity in Section 6.6.

Scaling Up. *Workers.* Once a node becomes available, and the appropriate software has been initialized, it contacts the elasticity controller responsible for the job and receives its input data assignment (see transition to phase 2 in Fig. 5). It loads the data (from S3 storage for AWS EC2) and signals the elasticity controller that it's ready. The elasticity controller then signals corresponding workers to update their working sets. *ActivePS.* AgileML achieves best performance when running ActivePSs on half of the resources (Sec. 6.4). This ratio is thus maintained when scaling the resource footprint. When the resource footprint increases, AgileML starts new ActivePSs on the longest running transient resources that do not yet have an ActivePS. It notifies the resource to host the ActivePS and serve a given partition assignment. A partition is a unique subset of the parameter state. During start-up, AgileML divides the parameter state into N partitions, where N is the maximum number of ActivePSs that can exist at any one point. We found that setting N equal to half of the maximum number of resources that could be used by AgileML at any point to be effective. Each partition is assigned to a ParamServ. In stage 2 and 3 each partition is also assigned to an ActivePS, which is responsible for forwarding updates applied to the partition to the BackupPS that owns it. By using partitions in this way, AgileML avoids the need to re-shard the parameter state when adding or removing servers, instead re-assigning partitions as needed.

The resource that starts the new ActivePS contacts the previous partition owner for a copy of the partition. The original owner points all workers to the new partition owner. During ownership propagation, all partition messages are forwarded to the new ActivePS. Workers and ActivePS additions happen in the background with negligible impact on system performance (Sec. 6.6).

Scaling Down. AgileML differentiates between evictions and failures based on whether it received a warning, and it handles them differently. When resources are removed from an ongoing computation after some warning, such as the two-minute warning offered by AWS or the 30-second warning offered by GCE, we call this an *eviction*. When resources are removed without warning or with a warning detected with insufficient lead time, we call this a *failure* or an *effective failure*, respectively.

Evictions. AgileML's elasticity controller checks for eviction warnings every 5s. These warnings consist of a set of instances marked for eviction, if any. When this set includes all transient resources, the elasticity controller signals all ActivePSs to push their most recent consistent state to the BackupPSs and cease operation. A special end-of-life flag is appended to these updates to signal the last message from ActivePS to BackupPS. When the BackupPSs receive end-of-life messages from ActivePSs, they signal any workers

[5] Input data assigned to workers on reliable resources is preloaded by workers on transient resources, simplifying the transition from stage 2 to 3.

on reliable machines (including those getting turned on by the elasticity controller, as discussed at the end of this section) to address read and update requests to them. Note that the AgileML design makes this scenario simple and fast.

When an eviction is about to take back only some of the transient resources, the elasticity controller signals the ActivePSs that are being evicted to either (i) move their partitions to the ActivePSs that will survive the eviction, or (ii) move them to transient resources that are going to survive the eviction and do not yet have an ActivePS running on them (see transition to phase 3 in Fig. 5). The process for relocating partitions mirrors the process of adding ActivePSs above, which includes pointing all surviving workers to the new partition owner.

Failures. In the case of failures, which are detected via heartbeat messages, or effective failures, when the eviction warning is not early enough for all evicted ActivePSs to send their end-of-life messages to BackupPSs, AgileML performs a form of on-line roll-back recovery. This roll-back recovery depends on how many resources have failed.

When all or most of the transient resources fail (usually due to an effective failure), the BackupPSs will use the last consistent state[6] from the ActivePSs as the new solution state, and the workers will re-do the work lost in the roll-back recovery. The ActivePSs send the workers the iteration number of the last iteration included in the new solution state, and all workers will restart from what is essentially an online checkpoint. When a single or few resources running ActivePSs fail, the elasticity controller reassigns partition ownership from those ActivePSs to other transient resources. This is done by the BackupPSs sending their solution states to the new owners of the ActivePSs. The surviving ActivePSs then roll-back to a state consistent with the current state of the BackupPSs. This roll-back to a consistent state is straightforward, because the ActivePSs already store the aggregate of the delta applied to their local state since the last time they applied their state to the BackupPSs.

Reacting to the eviction and failures of workers is orchestrated by the elasticity controller. When a worker is removed from a computation, the previous owner of the worker's input data takes ownership for it. A previous owner always exists when input data is assigned to a transient node. Thus, there will be no need to stop and load the input data from storage. To account for the infrequent failure of reliable resources, checkpointing of reliable resources can be used. In stage 3 of AgileML, checkpointing of reliable resources has no overhead on ML training speed because there are no worker threads running on these resources.

[6] Recall that parameter server systems often allow flexibility in progress synchronization among workers and shared state consistency. Often, workers see parameter values that do not yet reflect recent updates from all other workers, but a bound on the staleness is often enforced [11, 25]. In such systems, the consistent state corresponds to the latest common iteration and reflects all updates up to that iteration and no updates afterwards.Achieving a consistent state requires either synchronization of worker progress or (usually) some extra buffer memory.

Stage Transitions. AgileML uses the ratio of transient to reliable resources to determine which stage to use. For ratios greater than 1:1, AgileML uses stage 2, and for ratios greater than 15:1, it uses stage 3 (Sec. 6.4). While transitioning between stages is important for AgileML performance, as the ratio of transient to reliable resources changes, we find that perfect threshold settings are not required. For our work, appropriate thresholds for different compute clusters were determined by measuring and comparing system performance for the three stages at different ratios (Sec. 6.4), resulting in the 1:1 and 15:1 thresholds as well as the observation regarding low sensitivity. We believe that future work can automate the threshold selection process for any given cluster.

4. BidBrain Design

BidBrain is Proteus' resource allocation component. It keeps track of current and historical market prices for different types of resources (e.g., Amazon EC2 instance types) and makes allocation decisions of the form $\vec{x}$, where x_i corresponds to the set of instances allocated of type i. Fig. 6 illustrates how this allocation vector changes over time as the allocation footprint managed by BidBrain changes (due to BidBrain decisions or to evictions).

BidBrain decisions consider several parameters characterizing the application (Table 2), including its ability to scale with more instances (ϕ), cost of modifying its resource footprint (σ), and the cost of evictions (λ). BidBrain's primary objective is to minimize cost per unit work. The current implementation of BidBrain focuses on Amazon EC2, but we believe that its mathematical framework and mechanisms can also be applied in other cloud provider settings.

Resource Allocation. BidBrain interfaces with AWS to acquire new resources. To do so, BidBrain supplies a (instance type, count, bid price) tuple. We call this an allocation request. An *allocation* is defined as a set of instances of the same type acquired at the same time and price. BidBrain's *total footprint* $\vec{x}$ is a set of such allocations that are the elements of $\vec{x}$. We use different colors in Figures 5 and 6 to signify different atomic allocations. Resource allocation decisions are made periodically as well as a few minutes before the termination of each billing hour.

4.1 Formulation

At each allocation decision, BidBrain calculates the total expected cost and the total expected work by considering available instance types and their current spot market prices. BidBrain works with the following free variables: (a) instance type i to choose, (b) *bid delta* to bid over the spot price.

Expected Cost. Given a set of *allocations*, the total cost for a given footprint $\vec{x}$ is calculated as the sum over individual allocation costs $C_A[x_i]$, where each allocation's cost is calculated as the product of its instance count k_i and instance

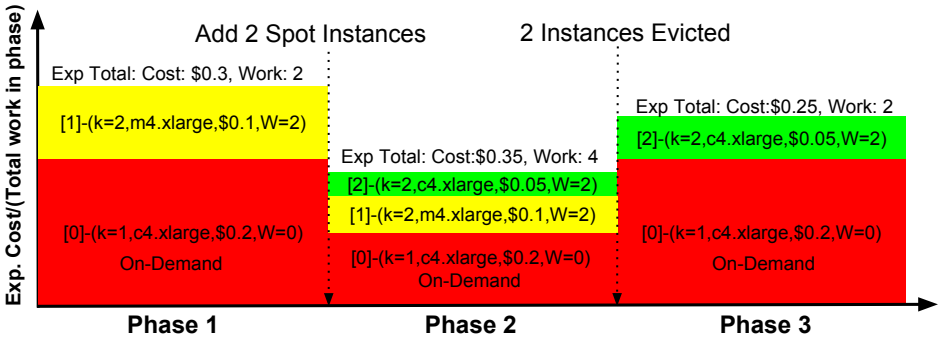


Figure 6: Expected cost per unit work for the toy example transitions in Fig. 5. Each block represents an *allocation* (Sec. 4), described by how many instances are in the allocation (k), instance type, the expected cost of the allocation, and the expected work produced by this allocation. Each block's height equates to that allocation's relative contribution to the cost of the total work done in its phase. Combining the blocks' heights in each phase equates to the total expected cost per unit work for that phase. In phase 1, BidBrain has an expensive, required on-demand allocation (red) that produces no work and a spot allocation (yellow). The on-demand instance type is pre-determined to be c4.xlarge and is never terminated by BidBrain, even if it negatively affects cost-per-work. In phase 2, BidBrain further amortizes the cost of the red allocation by adding a second spot allocation [2] (green), which lowers the total expected cost-per-work. This transition increases its actual cost at that moment, but reduces the final cost by decreasing the amount of time for which the on-demand allocation is needed.

price P_i:

$$C_A = \sum_{i=0}^{n} C_A[x_i] = \sum_{i=0}^{n} k_i * P_i$$

One of BidBrain's features is to reason about the probability of free compute it can get if its instances are evicted before the billing hour expires. If the allocation is evicted, AWS refunds the amount charged at the beginning of the current billing hour. To capture this, BidBrain calculates the *expected* cost of an allocation by considering the probability of eviction β_i for a given instance type i at a given *bid delta*. There are only two possibilities: an allocation can either be evicted (with probability β_i) or it will reach the end of its billing hour in the remaining t_r minutes (with probability $1 - \beta_i$). The expected cost can, therefore, be written down by the definition of expectation (Eq. 1).

$$C_A[x_i] = (1 - \beta_i) * P_i * k_i * t_r + \beta_i * 0 * k_i * t_r \quad (1)$$

Estimating Evictions. To estimate β_i in Eq. 1, BidBrain uses historical AWS spot market price data. This historical data is gathered individually for each instance type in each availability zone and indicates the price at each instant in time. Combining such data with knowledge that spot instances are evicted when the price rises above the bid, BidBrain computes the historical probability of being evicted within the hour and the median time to eviction for a given bid delta. The *bid delta* is the difference between the bid price and the market price. Using the AWS spot market trace from March to June of 2016, we ran simulations with a wide range of *bid deltas* [$0.00001,$0.4] and recorded the probability of getting evicted within the hour, β, and the median time to eviction. Using this information, BidBrain estimates the probability of eviction for any allocation.

β	Probability that allocation is evicted (0-1)
ϕ	How efficiently application scales (0-1)
σ	Overhead of adding/removing resources (min)
λ	Overhead of evicting resource (min)
ν	Work produced by instance type
ω_i	Max compute time remaining in allocation i
C_A	Expected cost of a set of allocations ($)
W_A	Expected work of a set of allocations
E_A	Expected cost per work of a set of allocations

Table 2: Summary of parameters used by BidBrain

Expected Work. BidBrain explicitly reasons about the expected amount of work each allocation is expected to produce. To capture this, BidBrain computes the *expected* useful compute time Δt_i for each allocation by considering factors such as eviction overhead, overhead for resource addition, and scalability of the application.

The maximum useful compute time of any allocation is the time remaining in the current billing hour ω_i. If BidBrain expects the allocation to be evicted prior to the end of the billing hour, it reduces ω_i accordingly. The eviction of any allocation reduces the useful compute of each allocation x_i by the eviction overhead λ of the application. The probability of an eviction for a set of allocations is computed as: $1 - \prod_{j=0}^{n}(1 - \beta_j)$, where β_j is the probability of eviction for allocation j. When considering removing or adding resources, BidBrain subtracts this overhead σ from the expected compute time for each allocation (Eq. 2).

The expected work for an allocation is the product of its resources k_i, expected useful compute time Δt_i and the work produced per time by that instance type ν.[7] BidBrain expresses the expected work for a set of allocations as the sum of each allocation's expected work reduced by the scalability overhead ϕ of the application (Eq. 3).

$$\Delta t_i = \omega_i - (1 - \prod_{j=0}^{n}(1 - \beta_j)) * \lambda - \sigma \quad (2)$$

$$W_A = (\sum_{i=0}^{n} k_i * \Delta t_i * \nu) * \phi \quad (3)$$

[7] For ML workloads performed by AgileML, work produced is proportional to the number of cores on an instance. For example, ν of a c4.2xlarge instance (8 cores) is equivalent to 2 * ν of a c4.xlarge instance (4 cores).

Table 2 summarizes the parameters used by BidBrain. In future work, we plan to automate the process of determining ϕ, σ, λ and ν. Currently, we set ϕ, σ, λ empirically (see experiment description in Sections 6.5 and 6.6). ν is set to equal the number of virtual cores in the instance and is a proxy for how much work an application is expected to achieve on that instance per unit time. ϕ measures the first order Taylor series expansion coefficient of the application's scalability curve as a function of instance count of a given type. σ and λ measure for how long the application does not make progress after a change in the resource footprint.

4.2 Resource Acquisition

BidBrain acquires resources x_i only if they lower the expected cost per work of its footprint $\vec{x}$. Expected cost per work for a set of allocations is approximated as the expected cost divided by the expected work produced (Eq. 4).

$$E_A = C_A / W_A \tag{4}$$

During every "decision point", BidBrain builds a list of possible allocations that it can make. This set of possible allocations is constructed by pairing different bid prices with different instance types. The range of possible bid prices includes $[\$.0001, \$.4]$ over the current spot market price. Once BidBrain constructs the set of possible allocations, it computes the cost per work for the current allocations and the cost per work for current allocations plus each of the possible allocations. If the cost per work for the best possible allocation is smaller than for the current allocations, BidBrain will send this allocation request to AWS. As described earlier, each allocation is made for the duration of the billing hour. This means that briefly before the end of an allocation's billing hour, BidBrain compares the cost per work if the allocation is renewed or terminated. If the cost per work is lower when the allocation is not renewed, BidBrain will terminate all the instances in the allocation prior to them reaching the next billing hour. In addition to spot resources, BidBrain acquires the required amount of on-demand resources (reliable instances in Fig 4). It does not consider terminating these resources even if they negatively affect cost-per-work.

4.3 Application Compatibility

BidBrain's design is compatible with applications beyond AgileML. It should work well for batch computations, where optimizing cost per unit work makes sense, that are able to efficiently add and remove large portions of their resource footprint quickly and efficiently. In future work, we plan to explore other optimization metrics to fit other elastic application types.

5. Proteus Implementation

This section describes how Proteus incorporates BidBrain and AgileML and how it connects to AWS. Figure 7 shows a high level overview. As described in Section 3, the user links an ML application to Proteus and specifies the location of the training data-set. The user is also responsible for providing the security credentials necessary to connect to AWS.

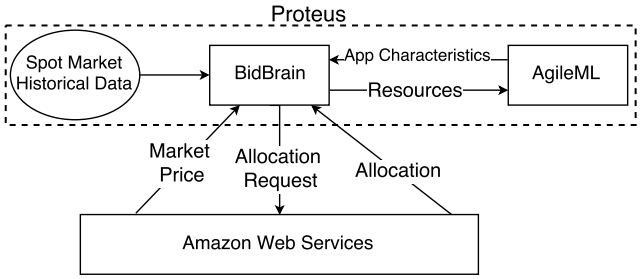


Figure 7: The Proteus architecture consists of the resource allocation component, BidBrain, and the elastic ML framework, AgileML.

Upon start-up, Proteus connects AgileML to BidBrain via a ZMQ message that specifies the application characteristics (Sec. 4). Proteus then connects to AWS, gathers the current spot market price via *boto.ec2* API calls and loads the historic spot market data, both of which are directed into BidBrain. Using the information about the AWS spot market in combination with information about the characteristics of the ML application, BidBrain builds allocation requests (Sec. 4), which Proteus sends to AWS via the *boto.ec2* API. Upon receiving the allocation requests, AWS returns a set of spot request ids, which are translated by BidBrain to the assigned AWS EC2 instances. Once these instances become reachable via SSH, BidBrain sends a ZMQ message to AgileML's elasticity controller containing the list of IP addresses and sizes of the instances in the new allocation.

BidBrain considers making new allocation requests to AWS every two minutes, briefly before the end of a billing hour of any current allocations, and immediately following an eviction. BidBrain monitors AWS for eviction notifications. Upon receiving an eviction notification, BidBrain translates it to the ids of the resources that are affected and notifies AgileML's elasticity controller. Proteus assumes that multiple ML applications are executed in sequence. Upon completing the final job in the queue, Proteus immediately terminates the on-demand resources. It then waits until the end of current billing hours to terminate the spot allocations, in hope that they are evicted by AWS prior to the end of the billing hour, lowering the overall cost.

In the current design, there is no redundancy for BidBrain or the elasticity controller. If either components fails, Proteus is still able to continue making progress. Either component can be restarted, if it fails, and synchronized with the ongoing computation.

6. Evaluation

This section evaluates Proteus' effectiveness. The results support a number of findings, including: 1) In the context of AWS, Proteus' exploitation of spot market resources significantly reduces cost (e.g., by ≈85% compared to on-demand only) and outperforms standard bidding policy combined with a checkpointing-based elasticity in terms of both cost (by 42%–47%) and runtime (by 32%–43%); 2) Proteus' elasticity support introduces minimal overhead to a traditional non-elastic parameter-server configuration; 3) Proteus

enacts bulk machine additions and revocations with minimal disruption, performing most setup actions in the background.

6.1 Experimental Setup

Experimental Platforms. We report results for experiments on two virtual cluster configurations on AWS. **Cluster-A** is a cluster of 64 Amazon EC2 c4.2xlarge instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). **Cluster-B** is a cluster of 128 Amazon EC2 c4.xlarge instances. Each instance has 4 vCPUs and 7.5 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). From our testing using `iperf`, we observe a bandwidth of 1 Gbps between each pair of EC2 instances.

6.2 Application Benchmarks

We use three popular iterative ML applications.

Matrix Factorization (MF) is a technique (a.k.a. collaborative filtering) commonly used in recommendation systems, such as recommending movies to users on Netflix. The goal is to discover latent interactions between the two entities (e.g., users and movies). Given a partially filled matrix X (e.g., a matrix where entry (i, j) is user i's rating of movie j), MF factorizes X into factor matrices L and R such that their product approximates X (i.e., $X \approx LR$). Like others [11, 14, 24], our *MF* implementation uses the stochastic gradient descent (SGD) algorithm. Each worker is assigned a subset of the observed entries in X; in every iteration, each worker processes every element of its assigned subset and updates the corresponding row of L and column of R based on the gradient. L and R are stored in the parameter server.

Our *MF* experiments use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements, and factor it into two matrices with rank 1000. We also use a synthetically enlarged version of the *Netflix* dataset that is 256 times the original. It is a 7683k-by-284k sparse matrix with 4.24 billion known elements with rank 100.

Multinomial Logistic Regression (MLR) is a popular model for multi-way classification, often used in the last layer of deep learning models for image classification [23] or text classification [26]. In *MLR*, the likelihood that each (d-dimensional) observation $\boldsymbol{x} \in \mathbb{R}^d$ belongs to each of the K classes is modeled by *softmax* transformation $p(\text{class}{=}k|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^T \boldsymbol{x})}{\sum_j \exp(\boldsymbol{w}_j^T \boldsymbol{x})}$, where $\{w_j\}_{j=1}^K$ are the linear (d-dimensional) weights associated with each class and are considered the model parameters. The weight vectors are stored in the parameter server, and we train the *MLR* model using SGD, where each gradient updates the full model [8].

Our *MLR* experiments use the *ImageNet* dataset [29] with LLC features [35], containing 64k observations with a feature dimension of 21,504 and 1000 classes.

Latent Dirichlet Allocation (LDA) is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words*. *LDA* discovers the topics via word co-occurrence. For example, "Sanders" is more likely to co-occur with "Senate" than "super-nova", and thus "Sanders" and "Senate" are categorized to the same topic associated with political terms, and "super-nova" to another topic associated with scientific terms. Further, a document with many instances of "Sanders" would be assigned a topic distribution that peaks for the politics topics. *LDA* learns the hidden topics and the documents' associations with those topics jointly. It is used for news categorization, visual pattern discovery in images, ancestral grouping from genetics data, community detection in social networks, and other such applications.

Our *LDA* solver implements collapsed Gibbs sampling [18]. In every iteration, each worker goes through its assigned documents and makes adjustments to the topic assignment of the documents and the words. The *LDA* experiments use the *Nytimes* dataset [5], containing 100m words in 300k documents with a vocabulary size of 100k. They are configured to classify words and documents into 1000 topics.

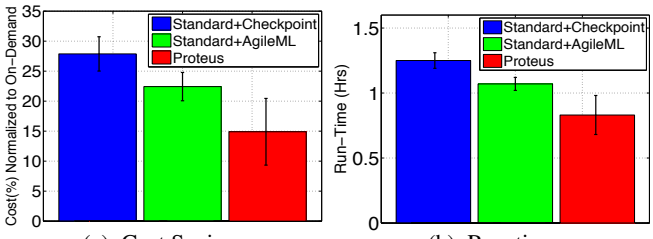
6.3 Cost Savings with Proteus

Proteus enables significant cost reductions on infrastructures that offer inexpensive transient machines. Fig. 1 summarizes the cost and time savings using BidBrain and AgileML for the *MLR* application. This section drills down further by evaluating Proteus' ability to reduce cost on EC2, relative to using only reliable on-demand machines, by analyzing the AWS Spot Market Traces from June 11, 2016 to August 14, 2016 for the US-EAST-1 region (all 4 zones).[8] We also compare the cost savings achieved by Proteus to those from existing approaches (see Section 8), which combine a checkpointing-based scheme for exploiting spot market machines with a standard spot market bidding strategy.

We perform cost savings analysis with long-term AWS traces, rather than experiments on EC2 for several reasons. Simulations on long-term AWS traces let us experiment with different approaches applied to the same market data, allowing for fair comparisons. Using AWS traces also allows us to gather data points across many months to get a fuller picture of expected behavior than our budget-limited experiments could otherwise provide. For each scheme and bidding model considered, we present the average cost (relative to full on-demand price) across 1000 randomly chosen day/time starting points in each zone. We perform experiments on durations with length of 2 and 20 hours, which is representative of long-running ML experiments (e.g., 4 hours for MLR) as well as the common practice of performing sequences of ML jobs for hyperparameter explorations.

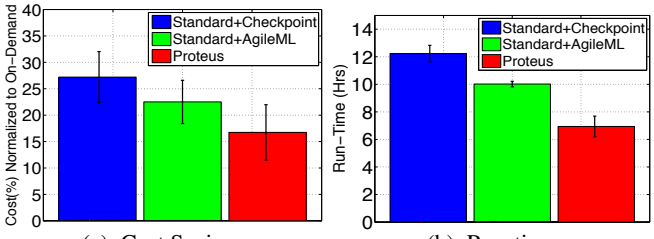
We present cost results as average cost per job, so for accounting purposes we do not charge a given job for any minutes that remained in a job's final billing hours (e.g., if 20 minutes left, the job is charged for only 2/3 of the cost of

[8] We used AWS Spot Market Traces from March 14, 2016 to Jun 10, 2016 to train the β parameter used in BidBrain.



(a) Cost Savings (b) Run-time

Figure 8: 2hr Job Duration



(a) Cost Savings (b) Run-time

Figure 9: 20hr Job Duration

the hour). We did this because the left over time is used by the following job in a sequence. This accounting was done the same way for all experiments, providing no benefit to Proteus.

Checkpointing-Based Scheme. As a comparison point for AgileML, we consider a scheme that tries to run entirely on spot market machines, using checkpointing to recover from evictions. We assume an MTTF-based checkpointing frequency, like that used in Flint [30]. We observe a resulting average checkpointing overhead of 17% for *MF* on both Cluster-A and Cluster-B (Sec. 6.3) when bidding the on-demand price, from the combined overheads of producing a consistent checkpoint state (recall that bounded staleness is allowed during ML application execution) and storing it. These overhead values are consistent with those reported by others [30].

Standard Bidding Strategy. As a comparison point for BidBrain, we consider an oft-used bidding strategy that selects the resource type with the lowest current market price and bids the on-demand price. It uses these resources until they are evicted, at which point it again selects the resources with the lowest current market price and bids the on-demand price. This is the default bidding policy used by Spot Fleet EC2, a service provided by AWS for users to acquire allocations of spot resources.

Cost Savings Results. Figure 8 and 9 show the cost savings and run-time for three different configurations for jobs of 2 hours and 20 hours, respectively, relative to running on 64 on-demand machines from Cluster-A: (1) the standard bidding strategy combined with the checkpointing-based scheme (blue). (2) the standard bidding strategy combined with AgileML, allowing evaluation of the incremental benefit of AgileML over the checkpointing-based scheme (green). (3) Proteus which combines BidBrain and AgileML (red). Comparing Proteus to the second configuration allows evaluation of the additional benefit of BidBrain over the standard bidding strategy.

The results demonstrate that Proteus significantly reduces both cost and run-times. On average, Proteus reduces cost by 83%–85% compared to traditional execution on on-demand machines and by 42%–47% compared to the state-of-the-art approach (Standard+Checkpoint). In addition to significantly lowering costs, Proteus also reduces run-times by 32%–43%. The results also show that each of BidBrain and AgileML contribute significantly to Proteus' overall cost and runtime improvements.

Attribution of Benefits. Proteus' superior performance arises from several factors. AgileML's ability to perform agile elasticity, ability to efficiently handle evictions, and lack of run-time overhead reduces the cost by 18%–20% compared to the checkpointing-based scheme (see blue and green bars in Figure 8 and 9). Similar benefits are seen when evaluating AgileML vs. the checkpointing-based scheme combined with BidBrain. The remaining improvements come from BidBrain's ability to effectively exploit the spot market. BidBrain reduces cost and run-time by providing opportunities for *free computing* and projecting how resource allocations impact work throughput.

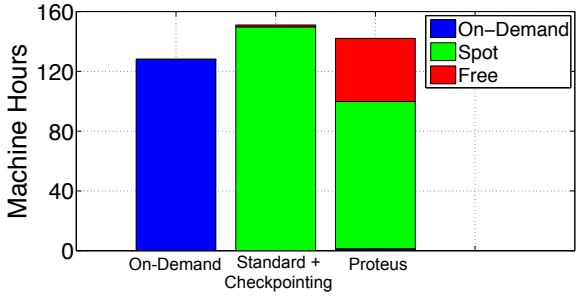


Figure 10: Breakdown of machine hours (for 2-hour jobs) among on-demand resources, spot resources (not evicted), and free resources (spot resources evicted prior to end of billing hour).

Free computing occurs when an allocation produces useful work but is evicted by AWS before the end of a billing hour. The user receives a refund for the last partial hour, which means that any work produced by the allocation during the current billing hour has no cost to the user. Users increase their likelihood of getting *free computing* by bidding closer to the current spot market price, which increases the likelihood of evictions. When executing applications with significant eviction overheads, regularly bidding just above the current market price hoping to gain *free computing* is not an effective strategy. BidBrain accounts for eviction overheads in making decisions about how much above the market price to bid. We experimented with always bidding just above the market price to acquire *free computing* as often as possible, but it increased the run-time of applications (3-4x) and resulted higher cost due to suffering too many evictions after too short a period of time. BidBrain's predictions of eviction likelihood and times are effective enough to find a happy medium. On average, 32% of Proteus' computing is *free computing*, as shown in Fig. 10.

Experiments in the Wild. In addition to simulations, we ran a number of Proteus jobs on AWS. Although the results

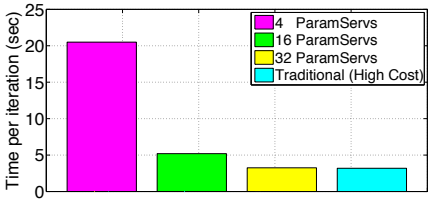


Figure 11: AgileML stage 1 with 4–32 reliable machines out of 64 total compared to traditional (all 64 reliable; cyan), for MF.

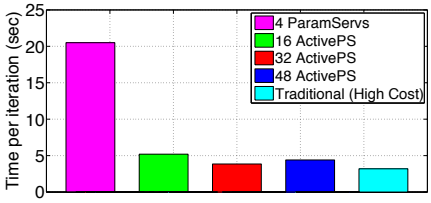


Figure 12: AgileML stage 2 with 4 reliable and 60 transient compared to stage 1 (same ratio; magenta) and traditional (64 reliable).

Figure 13: AgileML stage 3 (red) with 1 reliable and 63 transient compared to stage 2 (same ratio; blue) and traditional.

cover a much smaller sample size than our simulations, the observed behavior is consistent with the simulation results.

6.4 Efficiency with AgileML Tiering

AgileML enables execution on a mix of reliable and transient machines, and efficient scale-up and scale-down, while always maintaining state required for continued operation on reliable machines. To avoid the reliable machines becoming a bottleneck, AgileML uses three stages of functionality partitioning (see Section 3.2), decreasing reliance on reliable machines as the ratio of transient to reliable increases. (Of course, higher ratios are better from a cost standpoint, because transient machines are often 70–80% cheaper.) This section evaluates AgileML's performance relative to the traditional parameter-server architecture run entirely on high-cost reliable machines, in which all functionality (worker and parameter server) is partitioned among all machines, showing that AgileML avoids performance loss at least to a ratio of 63 transient machines to 1 reliable machine. All results in this section are for the *MF* application with the *Netflix* data set on Cluster-A, but results for the other applications and Cluster-B are consistent and omitted only due to space constraints.

Stage 1: Parameter Servers only on Reliable Machines. The first stage spreads the parameter server across the reliable machines, rather than all machines, using transient machines only for worker processes.

Figure 11 shows the time-per-iteration for different numbers of machines running parameter server shards (ParamServs) in a 64-machine Cluster-A, representing different ratios of transient to reliable machines under the stage 1 configuration. All 64 machines run workers. The 64 ParamServ case, which is labeled "Traditional" in the graph, represents the traditional parameter server architecture in which all machines are reliable and run both worker and parameter server processes. The results show that stage 1 has negligible slowdown for a small ratio (e.g., 1:1, represented by "32 ParamServ") of transient to reliable machines, but introduces significant slowdown as the ratio increases. The slowdown is caused by network bottlenecks caused by many workers communicating with a relatively smaller number of ParamServs.

Stage 2: ActivePSs on Transient Machines and BackupPSs on Reliable Machines. To avoid the network bottleneck for higher ratios, stage 2 switches to a tiered primary-backup model, using reliable machines for continuity but not requiring them to serve as active parameter servers for a much larger number of workers.

Figure 12 shows the time-per-iteration for different configurations in a 64-machine Cluster-A that consists of 4 reliable machines and 60 transient machines. The "4 ParamServs" and "Traditional" bars described above for Figure 11 are included as well, for comparison. The other three bars represent running ActivePSs on different numbers of transient machines, together with BackupPSs on the 4 reliable machines. All 64 machines run worker processes, in each case. The results show that the ActivePS-based architecture with 32 ActivePSs introduces ≈18% slowdown compared to the traditional parameter-server architecture, when using a 15:1 ratio of transient to reliable machines. This slowdown does not occur at 7:1 and represents the beginning of the straggler problem addressed by stage 3.

Stage 3: No Workers on Reliable Machines. When the ratio of transient to reliable machines increases beyond 15:1, we observe even larger slowdowns for AgileML stage 2 relative to the traditional parameter-server architecture. This slowdown is caused by the workers running on reliable machines becoming stragglers; the network load of running BackupPSs for a much larger number of ActivePSs interferes with worker communication. To solve this problem, stage 3 does not run workers on the reliable machines when the ratio is very high. While this reduces aggregate worker computation power, stage 3 is only used when the reduction is small because the fraction of reliable machines is low.

Fig. 13 shows time-per-iteration with and without workers on the one reliable machine in a 64-machine Cluster-A that consists of 1 reliable machine and 63 transient machines. The one reliable machine runs only a BackupPS. The "Traditional" bar is again shown for comparison. The results show that, by shutting down reliable machine workers once they become stragglers, AgileML is able to match the performance of the traditional parameter-server architecture at a 63:1 ratio of transient to reliable machines.

Stage 3 provides the best performance only as the ratio of transient vs. reliable machines increases. Thus, all three stages are needed for AgileML to achieve high performance across a range of possible ratios. To illustrate, Fig. 14 compares time-per-iteration attained with the same footprint (8 reliable + 8 transient machines), but in two different modalities: stage 2 and stage 3. Stage 2 is clearly best for this 1:1 ratio, unlike Fig. 13, where the ratio was 63:1.

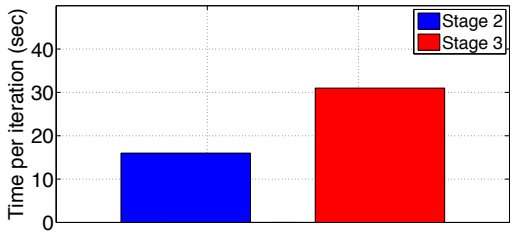


Figure 14: AgileML running on 8 reliable and 8 transient machines in stage 2 and stage 3 mode. Stage 2 is better for lower transient-to-reliable ratios.

6.5 AgileML Scalability

This section confirms that AgileML scales well as machines are added, like the traditional parameter-server architecture has been shown to do. Figure 15 shows time-per-iteration for the *LDA* application as a function of the number of Cluster-A machines used. (We observe the same scaling behavior for the other ML applications tested.) So, strong scaling is evaluated, and the curve labeled "Ideal" corresponds to perfect scaling of the 4-machine case. The 4-machine case uses the traditional parameter-server architecture to provide a baseline. The 8-machine case uses the stage 1 configuration for 4 reliable and 4 transient machines. The 16-, 32-, and 64-machine cases use the stage 3 configuration for 1 reliable machine and the remainder transient. These results show that AgileML scales effectively, exploiting available transient machines to speed up applications.

6.6 Efficiency of AgileML Elasticity

This section confirms that AgileML's mechanisms for bulk incorporation and eviction of machines induce minimal disruption of the ongoing ML application. Figure 16 shows time-per-iteration for each of 45 *MF* iterations on Cluster-A machines. The first 10 iterations execute on 4 reliable machines. 60 transient machines are incorporated during iteration 11, resulting in immediate speedup consistent with Figure 15. Adding the 60 machines causes no disruption because they are started, initialized, and prepared in the background, signaling the elasticity controller for final incorporation when ready. The opposite change is made in iteration 35, evicting the 60 transient machines from the computation, as though in reaction to an eviction notice. A 13% blip in performance is seen during the iteration in which the eviction is done, after which the time-per-iteration stabilizes, returning to its full 4-machine value. The blip occurs because of network overhead in aggressively bringing up-to-date the BackupPSs and transitioning them to being active ParamServs.

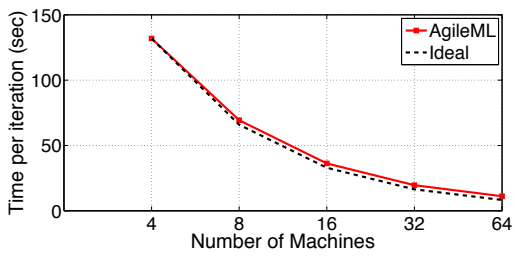


Figure 15: AgileML scalability for LDA. Showing time-per-iteration when using 4 to 64 machines.

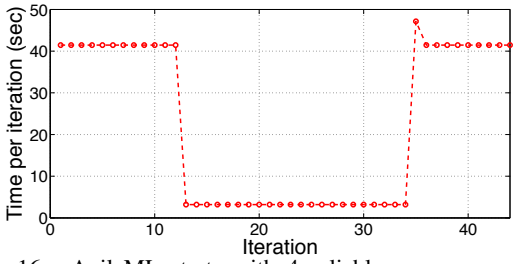


Figure 16: AgileML starts with 4 reliable resources, adds 60 transient resources at iteration 11, evicting 35 transient resources at iteration 35.

7. Discussion and Limitations

The popularity of systems like Proteus may increase spot market prices. We believe, however, that infrastructure clouds will continue to offer cheaper transient resources in order to monetize resources that would otherwise be idle due to varying demand on resources provisioned for contract customers. We also believe that AWS will eventually transition to a free market, where users are charged the *bid price* instead of the *market price*. This transition will render the commonly used strategy of bidding far above the market price obsolete, further motivating the need for intelligent bidding strategies, such as the one we instantiate with BidBrain.

BidBrain's allocation policies use AWS market price information to estimate resource reliability. However, BidBrain's allocation policies could be retargeted to be applicable beyond the AWS spot market. While this would eliminate the benefit received from free compute, only a portion of BidBrain's wins comes from such AWS specifics (Sec. 6.3). Instead of basing resource reliability predictions on historical AWS spot market data, BidBrain may perform reliability calculations by observing available resource capacity, its dynamics over time, and the activity of higher-priority jobs sharing the cluster. For example, in a private cluster setting, purchase cost (P_i) may be the same constant value for any best-effort allocation, but the expected work (Eq. (3)) still varies based on expected time to eviction (Eq. (2)).

Proteus is designed to work with stateless AgileML workers, which reduces system complexity and obviates the need for I/O intensive state-management operations, such as checkpointing. Indeed, the elasticity controller only needs to coordinate easily reassigned state in ActivePSs and ParamServs. This design choice is further motivated by the fact that most ML training algorithms can be implemented with stateless workers.

8. Prior Work

Previous work exploits transient resources using checkpointing, DHTs, RDDs, and heuristic bidding strategies.

Checkpointing. Checkpointing can be used to preserve state when using transient resources [19, 30, 31]. For example, a non-elastic computation can be started on EC2 spot market machines and checkpointed regularly. If the ma-

chines are revoked, the computation can be restarted on another set of machines from the last completed checkpoint. Gupta et al. [19] propose this approach for scientific computations. Parameter server architectures such as TensorFlow [6], MxNet [9], Petuum [37], LazyTables [11], and IterStore [12] provide no explicit mechanism for exploiting transient resources, and hence would likewise rely on checkpointing. A single machine failure causes most of these systems to restart an ongoing computation from the most recent checkpoint.[9] Although this is reasonable in small-to-medium clusters under traditional failure models, it can incur high overheads in elastic settings due to the frequency of revocations (e.g., all the spikes in Figure 3). Moreover, dynamically adding machines to running ML applications is not supported by these frameworks. To do so would seem to require stopping the computation in a consistent state, adding the resources, adjusting the mapping of computation tasks to machines and copying any needed state accordingly, and then restarting. (Section 3.3 describes AgileML's alternative, efficient approach.) We hope this work will motivate other ML frameworks to become agile elastic, and when they do, we believe they will integrate well with BidBrain and provide a great comparison for AgileML. In our experimental study, we compare Proteus' explicit elasticity support to this checkpointing-based approach.

Distributed Hash Tables (DHTs). The parameter server system described by Li et al. [25] includes support for adding and removing machines during execution. To realize this feature, the system uses a direct-mapped DHT design based on consistent hashing, wherein each parameter server process is responsible for a particular key range, and parameter value replication. Protocols for adding and removing machines are described. While DHTs are effective for adding or removing resources one-at-a-time, we believe that Proteus' approach to elasticity is better suited to the *bulk* addition and removal of nodes that characterize the transient resource availability discussed above. Li et al. did not evaluate the speed of node set changes, but we expect that it would be insufficient to address revocation of a sizable subset of cheap machines within the limited warning period provided. The replication mechanism also would not solve this issue, because bulk revocation is akin to *correlated* failure of many nodes, while the mechanism is designed for independent failures.

Spark and RDDs. Spark achieves fault tolerance with RDDs, storing deterministic transformations for subsequent replay on recovery from checkpoint. Flint [30], concurrently with our work, proposed a system for running Spark applications on transient machines. Unlike our tiered approach leveraging a mix of transient and non-transient machines simultaneously, Flint runs ML workloads entirely on transient nodes,[10] like the checkpointing approach above. RDDs reduce the cost of checkpointing/recovery for Spark applications by selectively choosing the set of RDDs needed. Whereas Flint relies heavily on the Spark's computing model in exploiting transient machines, AgileML enables exploitation of such resources for parameter server systems, which are different and much more efficient for iterative convergent ML (Sec. 2.1). Furthermore, Proteus' aggressive allocation strategy on the spot market provides significant savings, including 32% *free computing* on average. In contrast, Flint only considers switching when current resources are revoked and only bids the on-demand price, corresponding to Standard+AgileML in our graphs.

Bidding Strategies. Bidding strategies for EC2 spot instances have been studied [7, 32, 39]. Agmon et al. [7] show minimal correlation between near term AWS spot prices and spot instance availability. Marathe et al. [28] propose using redundancy *across* AWS zones for HPC computations on EC2. For interactive workloads, Flint [30] seeks to diversify across zones and machine classes to minimize revocation probability. Spot Fleet EC2, an AWS service, allows users to specify a resource capacity target and automatically maintain that target, replacing evicted instances. It is application agnostic, however, and does not take into account any application-level concerns, such as maximizing performance per unit cost. By default, Spot Fleet follows the same configured strategy as the standard bidding policy, bidding the on-demand price on the currently cheapest available resource (Sec. 6.3). We show significant improvements over such a bidding strategy.

9. Summary

Proteus aggressively exploits transient revocable machines to complete ML model training faster and cheaper. For example, Proteus can exploit EC2's spot market to save ≈85% compared to using only on-demand machines. By combining non-transient (e.g., on-demand) and transient (spot) machines, Proteus can rapidly and efficiently incorporate transient resources and deal with revocations, which combines with its aggressive allocation strategy to save ≈50% compared to a state-of-the-art checkpointing-based approach using a standard spot market bidding strategy.

Acknowledgments

We thank Steven Hand for shepherding this paper, and we thank the members and companies of the PDL Consortium: Broadcom, Citadel, Dell EMC, Google, HP Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas and Western Digital for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS), National Science Foundation under awards CNS-1042537, CCF-1533858, CNS-1042543 (PRObE [15]) and DARPA Grant FA87501220324.

[9] Tensorflow has a mechanism for handling single machine failures via its straggler mitigation mechanism.

[10] or entirely on non-transient nodes in the rare cases when they are cheaper

References

[1] AWS EC2. http://aws.amazon.com/ec2/.

[2] Spot Bid Advisor. https://aws.amazon.com/ec2/spot/bid-advisor/.

[3] Google Compute Engine. https://cloud.google.com/compute/.

[4] Apache Hadoop. http://hadoop.apache.org/.

[5] New York Times dataset. http://www.ldc.upenn.edu/.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 16)*.

[7] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3):16, 2013.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[9] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[11] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, 2014.

[12] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ML computations. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC 14)*, pages 1–14. ACM, 2014.

[13] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC 14)*, pages 1–14. ACM, 2014.

[14] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th Conference on Knowledge Discovery and Data Mining (KDD 11)*, 2011.

[15] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX*, 38(3), 2013.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[18] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[19] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé. Towards realizing the potential of malleable jobs. In *Proceedings of the 21st International Conference on High Performance Computing (HiPC 14)*, pages 1–10. IEEE, 2014.

[20] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC 16)*, pages 98–111. ACM, 2016.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, volume 11, pages 22–22, 2011.

[22] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS 13)*, 2013.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[24] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Proceedings of the 23rd Annual Conference on Neural Information Processing Systems (NIPS 09)*, 2009.

[25] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[26] J. Liu, J. Chen, and J. Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 547–556. ACM, 2009.

[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[28] A. Marathe, R. Harris, D. Lowenthal, B. R. De Supinski, B. Rountree, and M. Schulz. Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2. In *Proceedings of the 23rd international sym-*

posium on High-performance parallel and distributed computing, pages 279–290. ACM, 2014.

[29] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, 2014.

[30] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*, page 6. ACM, 2016.

[31] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. Spoton: a batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*, pages 329–341. ACM, 2015.

[32] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 5th IEEE International Conference on Cloud Computing(CLOUD 12)*, pages 91–98. IEEE, 2012.

[33] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, , B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC 13)*, page 5. ACM, 2013.

[34] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys 15)*, page 18. ACM, 2015.

[35] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *Proceedings of the Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3360–3367. IEEE, 2010.

[36] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC 15)*, pages 381–394. ACM, 2015.

[37] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 22th Conference on Knowledge Discovery and Data Mining (KDD 15)*, pages 1335–1344. ACM, 2015.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 10:10–10, 2010.

[39] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 71–84. ACM, 2015.