TierML: Using tiers of reliability for agile elasticity in machine learning

Aaron Harlap[⋆], Gregory R. Ganger[⋆], Phillip B. Gibbons[⋆]
[⋆] *Carnegie Mellon University*

CMU-PDL-16-102
May 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The TierMLparameter server system for machine learning (ML) enables aggressive exploitation of transient revocable resources to complete model training cheaper and/or faster. 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 (e.g., the Amazon EC2 spot market often provides such resources at a 90% discount). Unlike other parameter server systems, TierMLexploits such transient resources, using minimal non-transient resources to efficiently adapt to bulk additions and revocations of transient machines. Our evaluations show that TierMLreduces cost by ≈75% relative to non-transient pricing and by 46%-50% relative to using transient resources with checkpointing to address bulk changes, while nearly matching or decreasing running times.

Acknowledgements: We thank the members and companies of the PDL Consortium: Broadcom, Citadel, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Two Sigma, Uber 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 Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537, CCF-1533858 and CNS-1042543 (PRObE [14]), and DARPA Grant FA87501220324.

Keywords: Big Data infrastructure, Big Learning systems

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 70–90% discount 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 [27].

Unfortunately, efficient modern frameworks for parallel ML, such as TensorFlow [5], MxNet [8], and Petuum [34], 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 [22, 19, 9]. Like MPI-based HPC applications, these frameworks generally assume that the set of machines is fixed, optimizing aggressively for the no failure and no change case (rolling back to 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 TierML, a parameter server system designed to exploit transient revocable resources by elastically resizing as groups of machines become available or are revoked. To make such elasticity more efficient, TierMLexplicitly keeps core functionality on relatively reliable resources (i.e., non-transient resources, like on-demand instances in EC2), such that the computation may be slowed but never has to stop due to bulk revocations. Whenever possible, the vast majority of TierML's ML training work is performed on transient resources, exploiting the scaling opportunity while minimizing cost. But, using a small amount of more reliable resources enables quick adjustments as resources come and go (in bulk), such as quick re-expansion when new transient resources become available after a bulk revocation. It also enables proactive transitioning to new transient resources (such as to a different class of machines) when revocation of current resources appears likely and/or the price of the new resources drops below that of the current resources.

Figure 1 illustrates the benefits for an ML model training job on Amazon EC2. Using a single on-demand instance and 63 spot market machines, when possible, TierMLprovides an average cost savings of $\approx$75% when compared to using on-demand instances, even when accounting for spot market variation and revocations, while completing the job nearly as fast (or faster). Compared to using a checkpointing-based approach (i.e., run on spot market machines and checkpoint regularly so progress is retained if evicted [17, 27, 28]), TierMLsaves 46–50% the cost and also completes $\approx$20% faster, winning by avoiding checkpoint overheads, reducing restart delays, and proactively switching among machine types as their spot market prices vary. Experiments with three real ML applications confirm that TierML's elasticity support introduces negligible performance overhead in the absence of revocations, as well as that TierMLscales well, suffers minimal disruption during bulk addition or removal of transient machines, and provides significant benefits in both highly transient

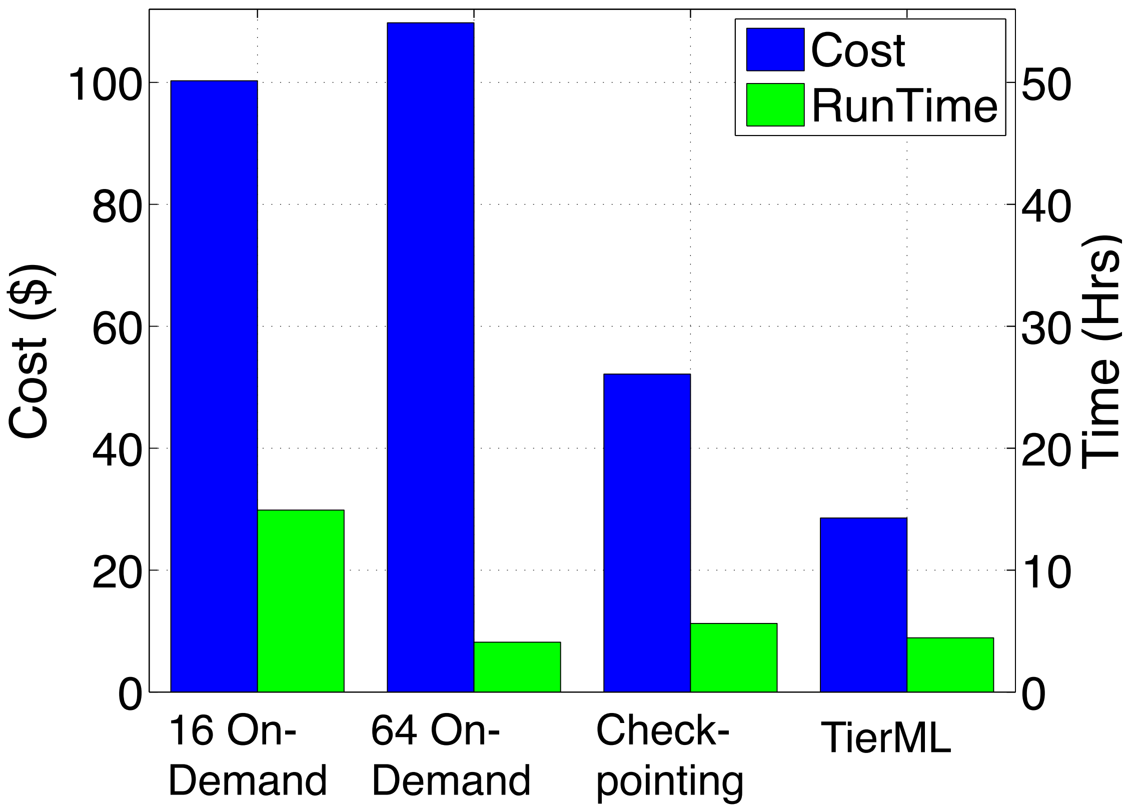

Figure 1: Benefits of exploiting EC2 spot market. This graph shows average cost (left axis) and runtime (right axis) for running the *MLR* application (see Section 4.2) on the AWS US-EAST-1 Region. The four configurations shown are: 16 and 64 on-demand machines, using 64 spot market machines with checkpoint/restart for dealing with evictions, and TierML using 1 on-demand machine and 63 spot market machines. TierMLreduces cost by 76% relative to using 64 on-demand machines and by 48% relative to the checkpointing-based scheme. Full experimental details can be found in Section 4.

and less transient situations.

Minimizing reliance on non-transient resources, while always having them be sufficient for continued operation, requires changes to the parameter server architecture and control algorithms. When the ratio of non-transient to transient is small (e.g., 2-to-1), one can simply distribute the parameter server functionality only across the non-transient machines, instead of across all machines as is the usual approach. For much larger ratios (e.g., the 63-to-1 ratio of Figure 1), the one non-transient machine would be a bottleneck in that configuration. In that case, TierMLuses the non-transient machine(s) as on-line backup parameter servers (BackupPSs) to active primary parameter servers (ActivePSs) running on transient machines. Updates are coalesced and streamed from actives to backups in the background at a rate that the network bandwidth accommodates. TierMLtransitions between these modes (and other functionality shifting) as transient resources come and go. Also, when different classes of transient machines vary in price and/or predicted MTTF, TierMLwill proactively transition to better options while continuing to execute, which our analysis of EC2 spot prices indicates provides over 30% cost savings on average.

This paper makes three primary contributions: First, it describes the first parameter server ML framework designed to elastically scale with bulk additions and revocations of transient machines. Second, it describes an architecture+algorithms for exploiting multiple tiers of machine reliability to more agilely resize in the face of such changes as well as to proactively transition among transient machine classes when beneficial. Third, it presents results from experiments and analyses showing that aggressive multi-tier exploitation of transient machines is both possible and beneficial, reducing costs by ≈75% relative to using only full-price and by 46–50% relative to checkpointing-based execution only on transient resources.

2 Motivation and Related Work

This section first presents background on ML model training, focusing on the *parameter server architectures*, the most efficient modern frameworks for parallel ML training. Then, it motivates the desire to have such frameworks be highly *elastic*, i.e., able to efficiently adapt to the dynamic availability and revocation of cluster resources. Finally, it discusses existing approaches to elasticity and how they fall short of desired.

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. These algorithms start with some guess at a solution (a set of parameter values) and refine this guess over a number of iterations over 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 4 scale well with machine count yet still require multiple hours even when using 10s of multicore machines.

Although iterative convergent ML algorithms can be built as sequences of bag-of-task computations, such as map-reduce jobs, on systems like Hadoop [1] or Spark [35], such implementations are inefficient. In particular, they preclude several specializations, including flexible consistency models [24, 10, 22], cross-iteration optimization [11], and early exchange of updates [33]. 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 [22, 19, 9]. As a result, open source ML model training frameworks like TensorFlow [5], MxNet [8], and Petuum [34] use this architecture, as do many proprietary systems.

Figure 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

[1]For example, one recent study showed two parameter server systems (IterStore [11] and LazyTable [10]) outperform PowerGraph [14] significantly (factors of 10X and 2X, respectively) for collaborative filtering via sparse matrix factorization [11]. 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 [15] approximately matches PowerGraph.

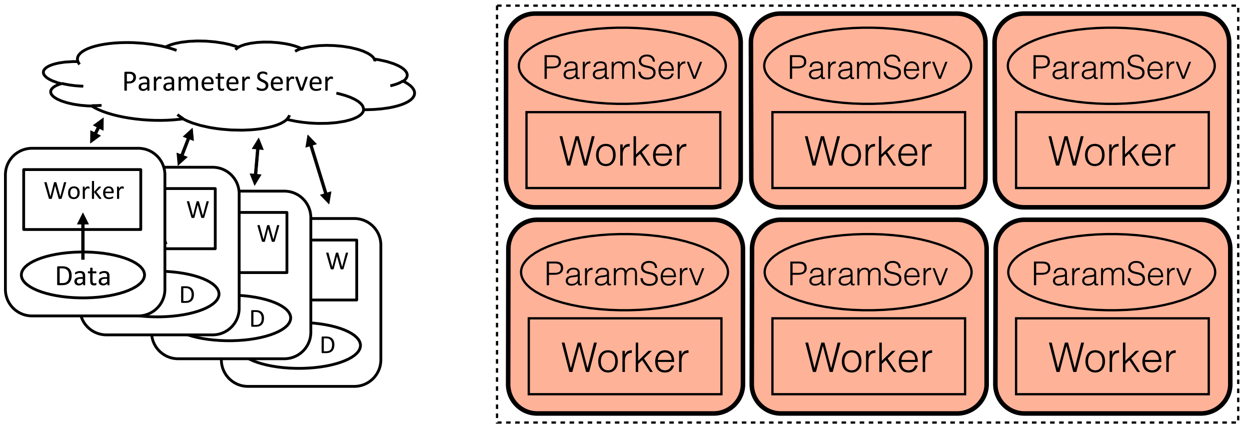

Figure 2: Traditional Parameter Server Architecture. The left figure illustrates the logical architecture, and the right figure illustrates that the parameter server is usually sharded across the same machines as the workers.

of Figure 2), the parameter server is usually sharded across the same machines as worker threads (right side of Figure 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 take advantage of any extra machines or potential cost savings available. Today's cluster infrastructures provide such opportunities, as discussed next.

2.2 Dynamic Availability of Revocable Resources

Cluster workloads are highly dynamic, and modern infrastructures increasingly provide temporarily-unused machines on a revocable basis at a discount (in public for-pay clouds) or for lower-priority users (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 [2] 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., 80–90% lower price) with the proviso that they can be taken back at any time. So, a customer that 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 [6]. 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 current EC2-determined *spot price* for that machine class.[2] Figure 3 shows one example of spot price variability over a week, for two machine classes in an EC2 zone. Second, if a customer

[2]This surprising practice of charging less than a customer explicitly states a willingness to pay is discussed further in Section 4.7.

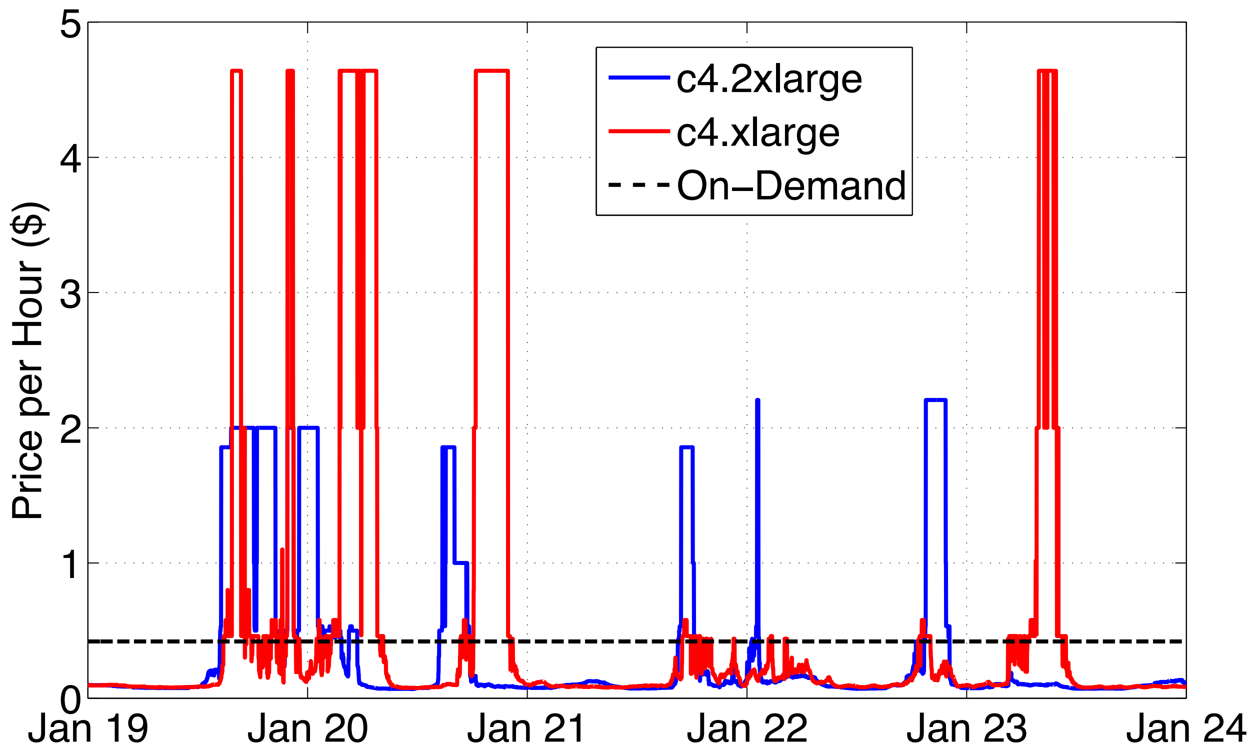

Figure 3: AWS spot prices over time. Spot prices for two classes of machines are shown for 6 days in January 2016. The unchanging on-demand price for c4.2xlarge is shown, and the values shown for c4.xlarge are doubled so that all three lines show the price for the same number of cores; c4.2xlarge machines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

receives machine resources in response to their bid price, they will retain them until 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—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 [30] and Mesos [18], 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 [12, 30, 31].

2.3 Prior Work

This section discusses prior work on exploiting transient resources, based on checkpointing, DHTs, RDDs, and bidding strategies.

Checkpointing. One way to exploit transient resources is by relying on checkpointing when needing to make a transition [17, 27, 28]. For example, a non-elastic computation can be started on EC2 spot market machines and checkpointed regularly. If the machines are revoked, the computation can be restarted on another set of machines from the last completed checkpoint. Gupta et al. [17] propose this approach for scientific computations. Parameter server architectures such as TensorFlow [5], MxNet [8], Petuum [34], LazyTables [10], and IterStore [11] 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. Although this may be acceptable in small-to-medium clusters under traditional failure modes, 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 TierML's alternative, efficient approach.) In our experiment study, we compare TierML's explicit elasticity support to this checkpointing-based approach.

Distributed Hash Tables (DHTs). The parameter server system described by Li et al. [22] 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 TierML's 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 via Resilient Distributed Datasets (RDDs), logging the history of deterministic transformations so failed tasks can be recomputed by reapplying the logged transformations from a checkpoint. Concurrent with our work, Sharma et al. [27] proposed Flint, a system for running Spark applications on transient machines. Unlike our tiered approach that uses both transient and non-transient machines at the same time, Flint runs solely on transient machines,[3] like the checkpointing approach described above. It reduces the cost of checkpointing/recovery for Spark applications by selectively checkpointing any RDD needed for fast recovery. Whereas Flint relies heavily on Spark's computing model in exploiting transient machines, TierMLenables parameter server systems to exploit such resources, which as noted in Section 2.1 are different and much more efficient for iterative convergent ML. In addition, TierML's use of some non-transient resources enables efficient, proactive switching among transient machine classes, increasing savings by an additional 30% and reducing machine re-acquisition delays after bulk evictions when spot prices are volatile. In contrast, because of the high overhead of switching between machine classes for checkpoint-based approaches, Flint only considers switching when current resources are revoked (i.e., the overhead must be paid).

Bidding strategies. A number of papers have studied possible bidding strategies for EC2 spot instances [6, 36, 29], noting the challenges in finding an effective strategy. Agmon et al. [6], for example, observed that there is little correlation in AWS between the near term spot price and the current availability of spot instances. Marathe et al. [25] proposed using redundancy *across* AWS zones to do HPC computations on EC2, and, for interactive workloads, Flint [27] seeks to diversify across zones and machine classes to lessen the likelihood of large revocations. Like Flint, TierMLbids the on-demand price. TierMLstrategically switches between classes of machines, as discussed in Section 3.4.

3 TierMLDesign and Implementation

TierML is designed to reduce cost and decrease run-time when utilizing compute clusters that offer resources of different tiers of reliability and cost, such as those described in Section 2.2. TierML, which is implemented as a C++ library linked by an ML application using it, is built upon the parameter server architecture described in Section 2.1. In the traditional parameter server model, it is common to have every machine run multiple workers (one per core) and an instance of the ParamServ. Together, these ParamServsjointly provide the functionality to read and to update parameter values. This section describes TierML's architecture, how it handles elasticity, and how it exploits transient resources to reduce cost.

3.1 Workers and Execution Management

During initialization, an ML application provides TierML with an initial list of reliable and transient nodes to be used, the input data file path, several functions called by TierML, and a stopping criterion. The stopping criterion may be a number of iterations, an amount of time, or a determination of convergence. The input file contains a sequence of data items in an understood format (e.g., rows

[3]or solely on non-transient machines in the rare case that such machines are cheaper. For non-ML interactive workloads like TPC-H, Flint seeks out a heterogeneous mix of machine classes.

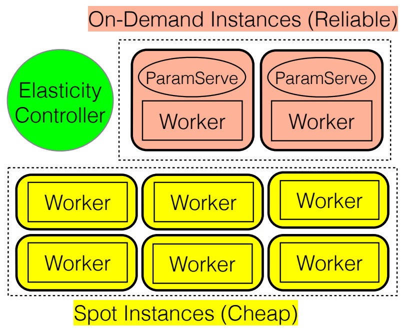

Figure 4: Stage 1: ParamServsonly on reliable machines

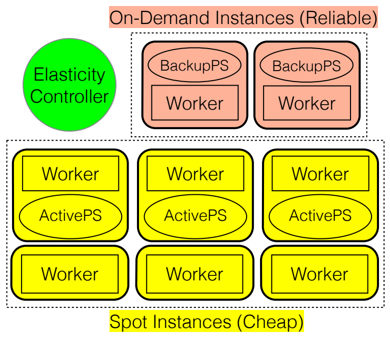

Figure 5: Stage 2: ActivePSson transient and BackupPSson reliable

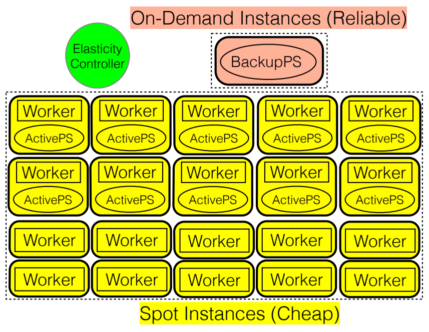

Figure 6: Stage 3: No Workers on Reliable Machines

that each contain one input data item in an easy-to-process format). During execution, TierML consists of one process executing on each node being used. Each TierML process starts a worker thread for each core on the node and a number of background threads for its internal functionality. The worker threads execute the ML application code for adjusting model parameters based on input data and the current solution state.

Each worker thread is assigned a unique ID, from zero to $N-1$, and a disjoint subset of the input data items. The default assignment is a contiguous range of the input data, determined based on the worker ID, number of workers, and number of data items. Each worker has an outer loop for iterating until the stopping criterion is reached and an inner loop for each iteration.[4]

3.2 Architecture

The objective of TierML is to enable ongoing computations to be fully elastic, in that they can take full advantage of changing resource availability and lower price machines. TierML accomplishes this by introducing the concept of tiers of reliability. TierMLenables 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 the cheaper prices of transient machines. To avoid the reliable machines becoming a bottleneck as the ratio of transient to reliable machines increases, TierMLuses three stages of functionality partitioning, as described in this section.

Stage 1: Parameter servers only on reliable machines. For a majority of ML applications including K-means, DNN, Logistic Regression, Sparse Coding, and the three described in Section 4.2, amongst many others, the workers are stateless while the ParamServscontain the current state of the solution. To take advantage of the two tiers of machine reliability (transient and reliable) in clusters, TierML's first stage spreads the parameter server across the reliable machines, rather than all the machines, running only workers on transient machines. This has the effect of removing all state from transient machines. To illustrate, consider a running example of eight AWS EC2 machines, of which six are spot instances (transient) and two are on-demand instances (reliable). As shown in Figure 4, we run a ParamServ and workers on each reliable machine and only workers on each transient machine.

By removing parameter state from transient resources, TierML is able to utilize them without losing progress when transient resources are revoked (or fail). This means that unlike a traditional

[4]We have simplified the description a bit. For greater flexibility, TierML actually provides a notion of a *clock of work* that gets executed on each inner loop, which may be some number of data items (a "mini-batch" of an iteration) or some number of iterations.

parameter-server architecture, no checkpointing is required to assist with using transient resources.[5] While this modification successfully removes state from transient resources, as shown in Section 4.3 it also causes a network bottleneck when the ratio of transient to reliable resources grows too large (e.g., with 60 transient and 4 reliable machines, the network bottleneck to the ParamServsslows the *MF* application by 7x). On the other hand, if we were to use only a *low* ratio of (cheap) transient vs. (expensive) reliable resources, then our cost savings would be modest at best.

Stage 2: ActivePSson transient machines and BackupPSson reliable machines. To avoid the network bottleneck for higher transient vs. reliable ratios, TierML's stage 2 introduces the concept of ActivePSs. Each ParamServis assigned several ActivePSs, and each one of these ActivePSs becomes responsible for a portion of the rows the ParamServ was originally responsible for. In this new architecture, workers send all their updates and read request to the ActivePSsinstead of the ParamServ. Each ActivePSin turn aggregates the updates, updates its local version of the solution state, and then pushes the updates to BackupPSsrunning on reliable machines instead of ParamServs. The purpose of the BackupPSis to serve as a hot backup, ready to step in whenever an ActivePS fails. As shown in Figure 5 the ActivePSsare able to run on transient resources. If the resource on which an ActivePSis running fails or is revoked, the state will still exist on the BackupPS.

In summary, in stage 2, TierMLswitches to a primary-backup model, using reliable machines for state required for continuous operation but not requiring them to serve as active parameter servers for the much larger number of workers. As shown in Section 4.3, the ActivePS-based configuration is much more efficient than stage 1 for higher transient-to-reliable ratios, but it still performs almost 2x slower than the traditional architecture at very high ratios (e.g., 63:1).

ParamServ	Contains solution state and always runs on reliable resources
BackupPS	When ratio of transient vs reliable resources grows too large, ParamServ transitions to BackupPS, serving as a form of a hot backup
ActivePS	Exist on transient resources, services client reads and writes, and periodically pushes delta of updates to BackupPS

Table 1: Types of solution state servers used by TierML

Stage 3: No workers on reliable machines. When the ratio of transient to reliable machines increases even further (e.g., beyond 15:1 in our experiments), we found that the ActivePSarchitecture failed to match the performance of the traditional architecture. This shortcoming was due to those workers operating on reliable machines becoming stragglers because of the heavy network load experienced by these machines. As shown in Section 4.3, removing these workers (or turning them off) enables TierML to reach near ideal performance. Note that this optimization is best applied only at high ratios, because it reduces the aggregate computational resources executing worker code. The threshold for turning off the workers on the reliable machines, to keep them from becoming stragglers, is a function of the quality of the network and the ratio of transient to reliable machines. Figure 6 shows an architecture (now with 20 spot instances and 1 on-demand instance) with no workers running on the reliable machine.

[5]In TierML, there is benefit in checkpointing the reliable resources in case they fail, however as we show later in this section, this check-pointing has no overhead on system performance.

TierMLdynamically adjusts to elasticity in the resources available for an ongoing ML application, by transitioning between these three stages as appropriate.

Elasticity Controller. Another modification to the traditional architecture is the addition of an elasticity controller, running on a reliable machine, to oversee elasticity. The *elasticity controller* is responsible for tracking which resources are participating in ongoing computations and the assignment of training data (input data) amongst the workers. When new resources are added to ongoing computations, the elasticity controller determines for which training data the workers on these new resources should take over responsibility. The elasticity controller is also responsible for deciding whether additional ActivePSsshould be started. Similarly, when resources are removed, the elasticity controller decides which resources take over responsibility of the training data they had been working on, and if any of the ActivePSsshould be shut down. The policies used by the elasticity controllerto make all these decisions are discussed next.

3.3 Handling Elasticity

In this section we will describe how TierML handles resources being added or removed from an ongoing computation. In Section 4.5 we evaluate TierML's effectiveness at handling such elasticity.

Adding resources. When new resources are added to an ongoing job, before they perform any computation, the resources must first start up, become accessible to the user, contact the elasticity controllerand load their portion of the input data. Previous work has shown that the average queuing delay for a spot request on AWS EC2 is 5 minutes [25]. Once the resource becomes available to the user, and the appropriate dependencies have been installed on the resource, it contacts the elasticity controller responsible for the job and receives the list of input data for which it will be responsible. Once the new resource loads the input data from storage (from S3 storage for AWS EC2), it notifies the elasticity controller that it is ready to join the computation. Upon receiving this notification, the elasticity controller notifies the resources previously responsible for this work to cease performing it. The resources giving up this work will continue to operate on the rest of their assigned work and input data.

As shown in Section 4.3, TierML achieves best performance when running ActivePSs on half of the resources. When a substantial amount of resources are added to on-going computations, the number of ActivePSs is increased accordingly. Once TierML decides to start a new ActivePS, it determines for which partitions (sets of rows) of the solution state the new ActivePS will be responsible. It then notifies the resource on which it plans to start the ActivePS about the assignment, and the identity of the previous owner of the partitions. When the resource receives this assignment it starts the ActivePS and sends a message to the original owner of the partitions. The original owner then sends the current copy of the partitions being requested and notifies all the workers about the new owner of the partitions. While the change of ownership message is propagating to all the workers, the original owner forwards to the new owner all messages associated with the partitions being transitioned. Additions of workers and ActivePSs are performed in the background, and as shown in Section 4.5, they have negligible impact on system performance.

Evictions and failures. 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. TierMLhandles these cases differently.

In the case of evictions, TierMLruns a background thread that checks for eviction warnings and alerts the elasticity controllerwhen one occurs (on EC2 this thread must run on one of the spot instances; every 5 seconds it checks for warnings and returns the set of spot instances that are marked for eviction, if any). In the common scenario where an eviction is about to take back all transient resources, the elasticity controllercontacts all the ActivePSsand tells them to push their most recent consistent state to the BackupPSand to cease operations after that. These update messages from ActivePSsto BackupPSswill also include a special end-of-life flag signaling that this is the last message that the ActivePSwill ever send. When the BackupPSsreceive end-of-life messages from ActivePSs, they signal any workers 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 TierMLdesign makes this scenario simple and fast.

In less common scenarios where an eviction is about to take back only some of the transient resources, the elasticity controllersignals the ActivePSsthat are being evicted to either (i) move their partitions to the ActivePSsthat will survive the eviction, or (ii) move them to transient resources that are going to survive the eviction and do not yet have a ActivePSrunning on them. The process for relocating partitions is similar to the process of adding ActivePSsdescribed earlier, and includes notifying any surviving worker where to start sending its read/update requests and what part of the input training data it now owns.

In the case of failures, which are detected via heartbeat messages, or effective failures, which arise whenever the warning provided to TierMLis not early enough for all evicted ActivePSsto send their end-of-life messages to BackupPSs, TierMLperforms a form of on-line roll-back recovery. This roll-back recovery differs depending on how many resources have failed.

In scenarios where all or most of the transient resources fail (usually due to an effective failure), the BackupPSswill 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 ActivePSssend 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. In scenarios where a single or few resources running ActivePSsfail, the elasticity controllerreassigns responsibility for the partitions managed by those ActivePSsto other transient resources. This is done by having the BackupPSsend the solution state it has to the new owner of the ActivePS. The surviving ActivePSsthen roll-back to a state consistent with the current state of the BackupPSs. This roll-back to a consistent state is straightforward, because the ActivePSsalready 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 input data the worker was responsible for assumes responsibility for it. As described in Section 3.4, any input data assigned to a transient resource will have a previous owner running on a reliable resource, thus there will be no need to stop and load the input data from disk.

3.4 Usage

This section describes how TierMLtakes advantage of cheap, transient resources, such as spot market machines offered in AWS EC2. Although we focus here on AWS, these same concepts can be extended to other clusters that possess similar characteristics. To begin a job, TierML starts up enough reliable resources to be able to make progress on the submitted job without relying on additional resources.[7]

[6]Recall that parameter server systems often allow flexibility in progress synchronization among workers and shared state consistency. Often, workers see parameters values that do not yet reflect recent updates from all other workers, but a bound on the staleness is often enforced [10, 22]. In such systems, the consistent state would reflect all updates from all workers through a given iteration, and no updates beyond that point. Achieving a consistent state requires either synchronization of worker progress or (usually) some extra buffer memory.

[7]If reliable resources are not available, we choose the most reliable resources available. In the ActivePS-architecture, checkpointing reliable resources running BackupPSsdoes not carry an overhead, making this substitution possible.

Choosing transient resources. In the case of AWS EC2, there are many machine types available, each with its own spot market. Our analysis of traces of the AWS spot prices shows that there is no correlation between the spot market prices of one machine type and the spot market prices for a different machine type—this finding is supported by other studies [27]. On the other hand, the time to eviction *is* correlated with the current spot price: the lower the spot price relative to the bid price, the longer the expected time to eviction. Following common terminology for machine failures, we will refer to the mean time to eviction as the MTTF (mean time to failure). TierMLexploits this latter correlation by, at the beginning of a job, selecting the machine type with the highest MTTF. This has the added advantage that such machines are often the cheapest option, when using the bidding strategy discussed below. Moreover, as the job runs, TierMLcontinues to seek out the machine types with the highest MTTF, thereby improving reliability and reducing costs, as discussed next.

Changing transient resources. Throughout the run-time of a job, TierMLcontinues to monitor the MTTF of the different machine types. If the MTTF of the current machine type becomes lower, meaning that their spot price begins to grow and approach the user bid price, TierMLwill compare the current MTTF of the machine type being used to the MTTF of other machine types. If it finds a machine type with an MTTF significantly higher than the current type, it will request transient resources of that type. Once these machines start-up (5 minutes on average [25]) and are ready to join the computation, all ActivePSsexisting on the old machine type will be moved over to the new type, as described in Section 3.3. The old machines will not be terminated immediately, but instead TierMLwill continue to use those machines to run (stateless) workers until the end of the current billing hour. This is done because the AWS partial hour rule, which specifies that any spot instance revoked by AWS will not be charged for the partial hour. Thus if preemption does occur, the last partial hour on these resources will be free. As shown in Section 4.6, this strategy of changing transient machine types significantly reduces the cost of jobs.

AWS bidding strategy. TierMLalways bids the on-demand price. When the spot market price becomes greater than the on-demand price, TierMLtransitions to using on-demand resources until the spot price returns below the on-demand price. Since AWS charges the user the spot-market price and not user bid price, bidding the on-demand increases the MTTF, while not increasing the costs due to our technique of switching to different classes of transient resources when the spot-market begins to approach the on-demand price. A common prior technique used by AWS spot market users was to enter bids that were significantly higher than the on-demand price. These users hoped to avoid eviction while paying the generally low prices set by the spot market. However, this strategy is no longer effective due to AWS capping user bids at 10x the on-demand price, in addition to introducing big spikes into the spot market prices that often greatly exceed the on-demand price. As a result, users who attempt this strategy are stuck paying prices significantly greater than the on-demand price whenever these big price spikes occur.

If, in the future, AWS changes their policy to charging users their bid prices instead of the spot-market price, it will force users to use lower bidding prices in order to reap the benefits of the spot-market costs. These lower bids will bring higher volatilityand lower MTTFs. TierML's ability to efficiently handle big changes in resource availability would make it possible to still operate very efficiently under this new policy, as shown in Section 4.7.

4 Evaluation

This section evaluates TierML's effectiveness. The results support a number of findings, including: 1) TierML's elasticity support introduces minimal overhead to a traditional non-elastic parameter-server configuration; 2) TierML enacts bulk machine additions and revocations with minimal disruption, performing most setup actions in the background; 3) in the context of AWS, TierML's exploitation of spot market resources significantly reduces cost (e.g., by ≈75% compared to on-demand only) and outperforms checkpointing-based elasticity in terms of both cost (by 46%-50%) and runtime (by ≈17%).

4.1 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.

4.2 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 [13, 21, 10, 11], 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. They are configured to factor it into the product of two matrices with rank 1000. We also used a synthetically enlarged version of the *Netflix* dataset that is 256 times the original. It's 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, such as used in the last layer of deep learning models for image classification [20] or text classification [23]. In *MLR*, the likelihood that each (d-dimensional) observation $\boldsymbol{x} \in R^d$ 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 $\{w_j\}_{j=1}^K$ is 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 [7].

Our *MLR* experiments use the *ImageNet* dataset [26] with LLC features [32], 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 "Congress" than "super-nova", and thus "Sanders" and "Congress" 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.

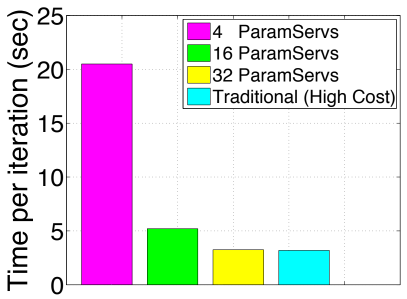

Figure 7: TierMLstage 1 with 4–32 reliable machines out of 64 total compared to traditional (all 64 reliable; cyan), for MF.

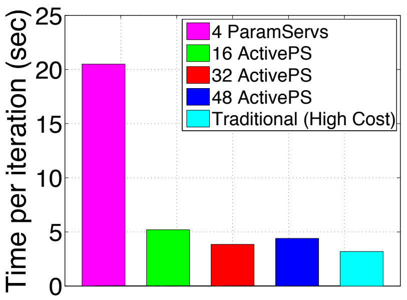

Figure 8: TierMLstage 2 with 4 reliable 60 transient compared to stage 1 (same ratio; magenta) and traditional (64 reliable).

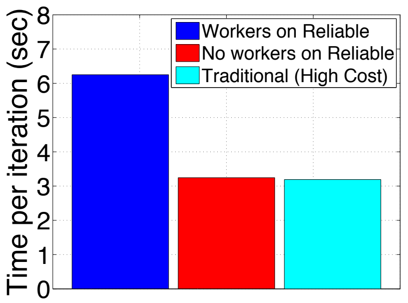

Figure 9: TierMLstage 3 (red) with 1 reliable and 63 transient compared to stage 2 (same ratio; blue) and traditional.

Our *LDA* solver implements collapsed Gibbs sampling [16]. 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 [4], containing 100m words in 300k documents with a vocabulary size of 100k. They are configured to classify words and documents into 1000 topics.

4.3 Efficiency with TierMLtiering

TierML enables execution on a mix of reliable and transient machines, while always maintaining state required for continued operation on reliable machines. To avoid the reliable machines becoming a bottleneck, TierMLuses three stages of functionality partitioning (see Section 3.2), decreasing the 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 70–90% cheaper.) This section evaluates TierML'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 TierMLavoids performance loss at least to a ratio of 63 transient machines to 1 reliable machine. All the results presented in this section are for the *MF* application with the *Netflix* data set on Cluster-A, but results for the other applications and Cluster-Bare 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 7 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 (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: ActivePSson 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 8 shows the time-per-iteration for different configurations in a 64-machine Cluster-Athat consists of 4 reliable machines and 60 transient machines. The "4 ParamServs" and "Traditional" bars described above for Figure 7 are included as well, for comparison. The other three bars represent running ActivePSson different numbers of the 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 TierMLstage 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 the worker communication. To solve this problem, stage 3 simply does not run workers on the reliable machines when the ratio is very high. While this reduces the worker computation power, stage 3 is only used when the reduction is small because the fraction of reliable machines is low.

Figure 9 shows time-per-iteration with and without workers on the one reliable machine in a 64-machine Cluster-Athat 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, TierML is able to match the performance of the traditional parameter-server architecture at a 63:1 ratio of transient to reliable machines.

4.4 TierML Scalability

This section confirms that TierMLscales well as machines are added, like the traditional parameter-server architecture has been shown to do. Figure 10 shows time-per-iteration for the *LDA* application, as a function of the number of Cluster-Amachines used. (We observe the same scaling behavior for the other ML applications tested.) So, strong scaling is evaluated, and the curve labelled "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 TierML scales effectively, exploiting available transient machines to speed up ML applications.

4.5 Efficiency of TierMLelasticity

This section confirms that TierML's mechanisms for bulk incorporation and extraction of machines induce minimal disruption of the ongoing ML application. Figure 11 shows time-per-iteration for each of 45 *MF* iterations on Cluster-Amachines. The first 10 iterations execute on 4 reliable machines. 60 transient machines are incorporated during iteration 11, resulting in immediate speedup consistent with Figure 10. 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, extracting 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 extraction 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 BackupPSsand transitioning them to being active ParamServs.

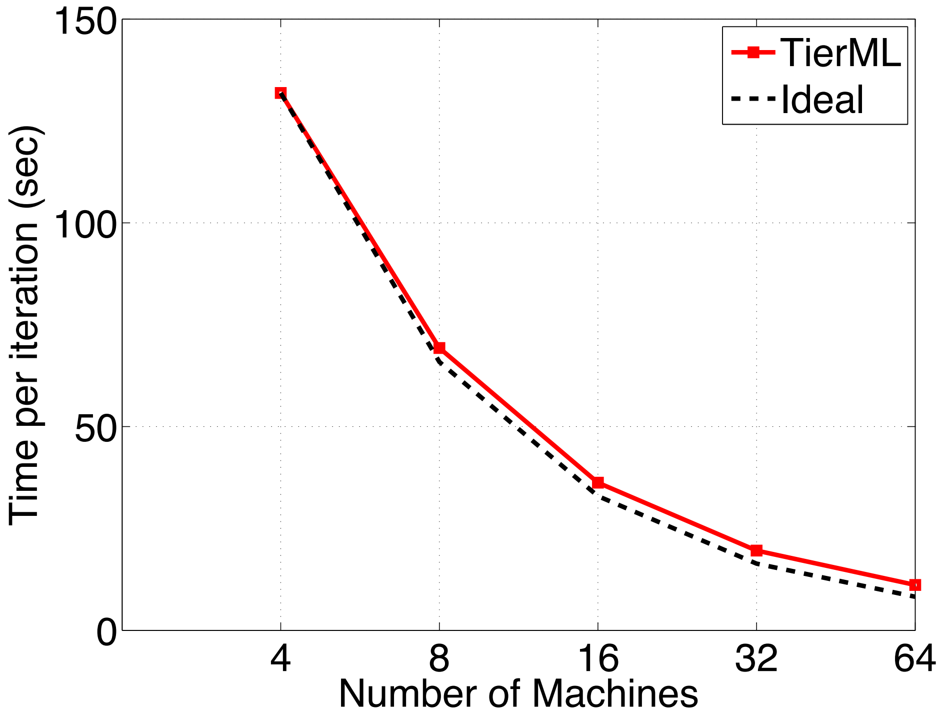

Figure 10: TierMLscalability for LDA.

4.6 Exploiting diverse machine classes

As discussed in Section 3.4, TierMLwill automatically switch between transient machine types when doing so would provide significant cost savings and address an anticipated near-term eviction. Specifically, in the context of EC2, TierMLtracks the spot markets of multiple similar machine classes (c4.xlarge and c4.2xlarge, in our experiments), which as noted earlier behave largely independently. When TierML finds that another class has a significantly higher MTTF than the one it is currently using, it will switch over to using it. Doing so provides two benefits. First, since the MTTF is determined by the spot market, the new class of machines will usually have a lower per-core cost. Second, by moving away from a lower MTTF, TierMLreduces the likelihood of being delayed waiting for EC2 to respond to new requests to replace machines taken back via eviction. Essentially, TierMLis proactively switching to avoid eviction-based periods of slow progress. And, if the transient machines TierML is switching away from are revoked during the remaining partial hour, they will have been free for that partial hour, based on the AWS billing policies.

Overall, we observed that TierML's use of this switching technique for choosing between Cluster-Aand Cluster-Breduces cost by 34% on average, regardless of which application is being performed. It also reduces average runtime by 8%, at the same time. We would expect even larger benefits from using more than two transient machine classes.

4.7 Cost savings with TierML

TierMLenables significant cost reductions when used on infrastructures that offer inexpensive transient machines. Figure 1 in Section 1 summarizes the cost and time savings using TierMLfor the *MLR* application.

This section drills down further by evaluating TierML's ability to reduce cost on EC2, relative to using only reliable on-demand machines, by analyzing the AWS Spot Market Traces from January 1, 2016 to March 14, 2016 for the US-EAST-1 region (all 4 zones). We also estimate the cost savings that would be achieved by a checkpointing-based scheme for exploiting spot market machines, as

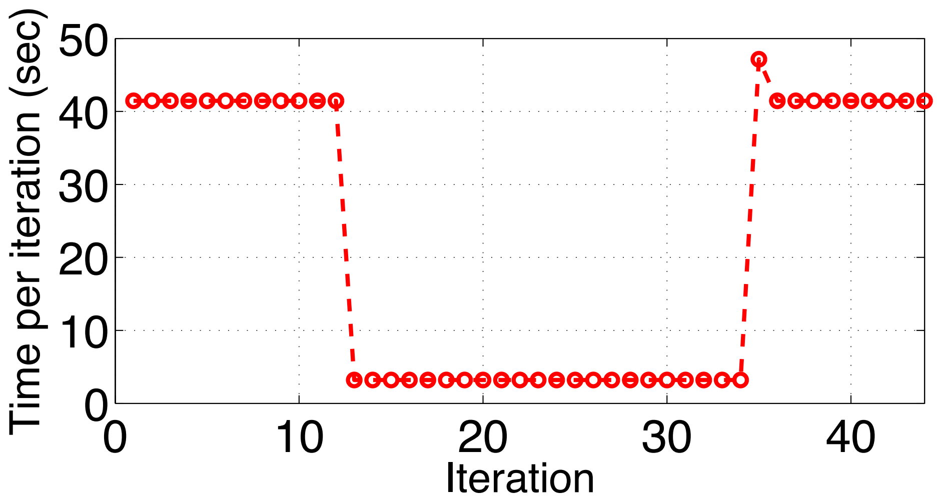

Figure 11: TierMLbegins with 4 reliable resources, adds 60 transient resources during iteration 11, then evicts these transient resources during iteration 35.

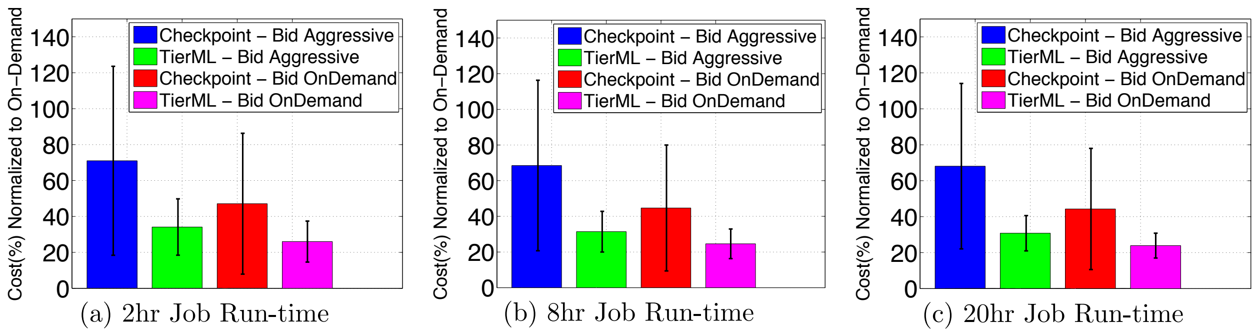

Figure 12: Average cost savings relative to on-demand. Error bars show standard deviation.

discussed in Section 2.3. We do cost savings analyses with long-term AWS traces, rather than experiments on EC2, so as to get a fuller picture of expected behavior than our budget-limited experiments would provide. For each scheme and bidding model considered, we present the average cost (relative to full on-demand price) across every possible starting minute in each zone.

Checkpointing-based scheme. As a comparison point, we consider a scheme that tries to run entirely on spot market machines, using checkpointing to recover from evictions. The scheme chooses the least expensive option (on-demand or either class of spot market machine considered) when starting initially or restarting from checkpoint after eviction. We assume an MTTF-based checkpointing frequency, like that used in Flint [27]. We observe a resulting average checkpointing overhead of 18% for *MF* on both Cluster-Aand Cluster-B, 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. When bid prices more closely track spot prices, as discussed below, the overhead rises as high as 50%. Both overhead values are consistent with those reported by others [27].

Bidding the on-demand price. We present data for two approaches to spot market bids. First, as discussed in Section 3.4, a common practice is to always bid the on-demand price, because Amazon will charge the spot-market price instead but only evict if that price rises above the bid. So, bidding the on-demand price guarantees paying the lesser of the two and also leads to fewer evictions than bidding closer to the spot-market price. There are still regular evictions, because the compute machines appropriate for ML applications are among the more popular ones and thus have more volatile spot markets.

Bidding aggressively (near spot price). While Amazon currently charges customers the spot-market price, independent of their bids, it is not difficult to imagine that generous pricing model changing. To consider TierML's behavior in a more free market, we also evaluate savings if we assume an alternate policy of charging users what they bid. Under such a policy, customers would need to more aggressively track the spot prices in setting their bids, resulting in more frequent evictions (lower MTTF) due to market variation. For these evaluations, we use this simple approach: set the bid price by rounding the current spot price up to the nearest dime for Cluster-Amachines (e.g., for a spot market price of \$0.23, we would bid \$0.3) and to the nearest nickel for Cluster-Bmachines. More advanced bidding strategies would likely increase the savings further.

Cost savings results. Figure 12 shows expected cost savings for TierMLand the checkpoint-based scheme, for each of the two bidding approaches. The three graphs represent cost savings for jobs of compute duration 2, 8, and 20 hours when run on 64 machines from Cluster-Aor 128 machines from Cluster-B. (Figure 1 shows the results for the 4-hour MLR application on Cluster-A.) The trace analysis accounts for runtime expansion caused by checkpointing overheads and eviction delays.

The results demonstrate that TierML decidedly outperforms on-demand and checkpointing, in terms of cost savings. On average, TierML reduces cost by 74%-78% compared to traditional execution on on-demand machines and 46%-50% compared to the checkpointing-based scheme, when bidding the on-demand price. With the more aggressive pricing, TierML's cost savings over on-demand decreases to 66%-70%, but it remains 52%-55% lower than checkpointing. Note that the variance of TierML's relative cost is much smaller than checkpointing's, in large part because of TierML's online backups and proactive machine type switching (Section 4.6); for example, the 95th percentile cost is still 59% lower than on-demand for BidOnDemand and 32% lower for BidAggressive. For the checkpointing scheme, no benefit is realized at the 95th percentile. While costs go down relative to full-size full-price on-demand clusters, the average runtimes do increase somewhat (11% and 23% for the BidOnDemand and BidAggressive approaches, respectively), because of delays incurred acquiring new machines after evictions. The average runtime increases are much larger (38% and 62%, respectively) for the checkpointing scheme, which has those overheads plus checkpointing overhead and is unable to perform proactive machine type switching the way TierMLdoes.

5 Summary

TierMLaggressively exploits transient revocable machines in its state-of-the-art parameter server framework to complete ML model training faster and cheaper. For example, TierMLcan exploit EC2's spot market to save $\approx$75% compared to using only on-demand machines. Moreover, by combining non-transient (e.g., on-demand) and transient (spot) machines, TierMLcan rapidly and efficiently incorporate new transient resources and deal with revocations, saving 45%-50% compared to a checkpointing-based approach while also being faster.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`.

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

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

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

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[6] 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.

[7] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 4. Springer, New York, 2006.

[8] 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.

[9] 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. USENIX Association, 2014.

[10] 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 *USENIX ATC*, pages 37–48, 2014.

[11] 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 ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[12] 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 ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[13] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 2011.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012.

[15] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

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

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

[18] 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 *NSDI*, volume 11, pages 22–22, 2011.

[19] 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 *NIPS*, 2013.

[20] 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.

[21] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[22] 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 *Proc. OSDI*, pages 583–598, 2014.

[23] 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.

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

[25] 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 symposium on High-performance parallel and distributed computing*, pages 279–290. ACM, 2014.

[26] 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.

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

[28] 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*, pages 329–341. ACM, 2015.

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

[30] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[31] 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 Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

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

[33] 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 Sixth ACM Symposium on Cloud Computing*, pages 381–394. ACM, 2015.

[34] 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 *KDD'15*, pages 1335–1344. ACM, 2015.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[36] 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.