Flint: Batch-Interactive Data-Intensive Processing on Transient Servers

Prateek Sharma Tian Guo Xin He David Irwin Prashant Shenoy
University of Massachusetts Amherst
{prateeks,tian,xhe,shenoy}@cs.umass.edu irwin@ecs.umass.edu

Abstract

Cloud providers now offer *transient servers*, which they may revoke at anytime, for significantly lower prices than on-demand servers, which they cannot revoke. The low price of transient servers is particularly attractive for executing an emerging class of workload, which we call Batch-Interactive Data-Intensive (BIDI), that is becoming increasingly important for data analytics. BIDI workloads require large sets of servers to cache massive datasets in memory to enable low latency operation. In this paper, we illustrate the challenges of executing BIDI workloads on transient servers, where revocations (akin to failures) are the common case. To address these challenges, we design Flint, which is based on Spark and includes automated checkpointing and server selection policies that i) support batch and interactive applications and ii) dynamically adapt to application characteristics. We evaluate a prototype of Flint using EC2 spot instances, and show that it yields cost savings of up to 90% compared to using on-demand servers, while increasing running time by $< 2\%$.

1. Introduction

Cloud computing platforms are an increasingly popular choice for running large data processing tasks. To maximize their utilization and revenue, Infrastructure-as-a-Service (IaaS) cloud providers are beginning to offer their customers the option to rent *transient servers*, which the provider may *revoke* at any time [28]. Due to their lack of an availability guarantee, transient servers are typically much (∼70-90%) cheaper than *on-demand servers*, which providers cannot unilaterally revoke once allocated. Both Amazon's Elastic Compute Cloud (EC2) and Google's Compute Engine (GCE) now offer transient servers in different forms. While transient servers are attractive due to their low price, they are generally only used for non-interactive batch jobs that are tolerant to lost state or delays caused by revocation [26].

The distributed data-parallel processing frameworks, such as MapReduce [13], that now dominate cloud platforms have historically executed their workload as non-interactive batch jobs. Since these frameworks were intended to operate at large scales, they were also designed from the outset to handle server failures by replicating their input and output data in a distributed file system. As a result, they required few modifications to run efficiently on transient servers [11, 19], where revocations are akin to failures. However, recently, there has been an increasing interest in better supporting interactivity in data-parallel frameworks. Interactivity enables data exploration, stream processing, and data visualization through ad-hoc queries. These new *batch-interactive* frameworks, including Spark [36] and Naiad [22], execute both batch and interactive applications and effectively enable a new class of workload, which we call *Batch-Interactive Data-Intensive* (BIDI).

BIDI workloads differ from the Online Data-Intensive (OLDI) workloads [20] processed by web applications in that the magnitude and variance of their acceptable response latency is much larger. For example, to avoid frustrating users, web applications often target strict latency bounds for rendering and serving each web page, typically on the order of 100 milliseconds with low variance. In contrast, users interactively executing a BIDI workload often have much more relaxed latency expectations, in part, because the amount of data each operation acts on (and the time it takes to complete) varies widely. Thus, users may expect query latency to vary anywhere between a few seconds to a few minutes. We argue that BIDI workloads' relaxed performance requirements still make them amenable to transient servers. Further, the low price of transient servers is particularly attractive to these new frameworks, since they require large sets of servers to cache massive datasets in memory.

Applications can employ fault-tolerance mechanisms, such as checkpointing and replication, to mitigate the impact of server revocations without rerunning the application. Checkpointing intermediate state enables restarting an application on a new server, and requires only partial recom-

EuroSys '16, April 18 - 21, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901319

putation from the last checkpoint. Of course, each checkpoint introduces an overhead proportional to the size of the local disk and memory state. Likewise, replicating the computation across multiple transient servers enables the application to continue execution if a subset of servers are revoked. However, replication is only feasible if the cost of renting multiple transient servers is less than the cost of an on-demand server. Prior work has only applied such fault-tolerance mechanisms at the systems level, e.g., using virtual machines (VMs) or containers [30]. While a systems-level approach is transparent to applications, we argue that an *application-aware* approach is preferable for distributed BIDI workloads, as it can i) improve efficiency by adapting the fault-tolerance policy, e.g., the checkpoint frequency and the subset of state to checkpoint, to each application's characteristics and ii) avoid implementing complex distributed snapshotting [10] schemes.

Since BIDI workloads support interactivity and low latency by caching large datasets in memory, revocations may result in a significant loss of volatile in-memory state. To handle such losses, batch-interactive frameworks natively embed fault-tolerance mechanisms into their programming model. For example, Naiad periodically checkpoints the in-memory state of each vertex, and automatically restores from these checkpoints on failure [22]. Similarly, Spark enables programmers to explicitly checkpoint distributed in-memory datasets—if no checkpoints exist, Spark automatically recomputes in-memory data lost due to server failures from its source data on disk [36]. Importantly, since failures are rare, these systems do not exercise sophisticated control over these fault-tolerance mechanisms. However, an application-aware approach can leverage these existing mechanisms to implement automated policies to optimize BIDI workloads for transient servers.

Since cloud providers offer many different types of transient servers with different price and availability characteristics, selecting the set of transient servers that best balances the per unit-time price of resources, the risk of revocation, and the overhead of fault-tolerance presents a complex problem. To address the problem, we design Flint, a batch-interactive framework based on Spark tailored to run on, and exploits the characteristics of, transient servers. Specifically, Flint includes automated fault-tolerance and server selection policies to optimize the cost and performance of executing BIDI workloads on transient servers. Our hypothesis is that Flint's application-level approach can significantly decrease the cost of running Spark programs by using transient servers efficiently to maintain high performance—near that of using on-demand servers. In evaluating our hypothesis, we make the following contributions.

Checkpointing Policies. Flint defines automated checkpointing policies to bound the time spent recomputing lost in-memory data after a revocation. Flint extends prior work on optimal checkpointing for single node batch jobs in the presence of failures to a BIDI programming model that decomposes program actions into collections of fine-grained parallel tasks. Flint dynamically adapts its checkpointing policy based on transient server characteristics and the characteristics of each distributed in-memory dataset.

Transient Server Selection Policies. Flint defines server selection policies for batch and interactive workloads. For batch workloads, the policy selects transient servers to minimize expected running time and cost, while considering both the current price of resources and their probability of revocation. In contrast, for interactive workloads, the policy selects transient servers to provide more consistent performance by reducing the likelihood of excessively long running times that frustrate users (for a small increase in cost).

Implementation and Evaluation. We implement Flint on top of Spark and Mesos, and deploy it on spot instances on EC2. We evaluate its cost and performance benefits for multiple BIDI-style workloads relative to running unmodified Spark on on-demand and spot instances using existing systems-level checkpointing and server selection policies. Our results show that compared to unmodified Spark, Flint yields cost savings of up to 90% compared to on-demand instances and 50% when compared to spot instances, while increasing running time by $< 2\%$. For interactive workloads, Flint achieves 10× lower response times when compared to running unmodified Spark on spot instances.

2. Background and Overview

We first describe the characteristics of transient servers, and then outline important elements of Spark's design before providing an overview of Flint.

2.1 Transient Servers

Our work assumes cloud platforms support transient servers, which the platform may revoke at any time [28]. Commercial platforms, such as Amazon's Elastic Compute Cloud (EC2) and Google Compute Engine (GCE), now support transient servers. In EC2, users bid for transient servers, which are referred to as *spot instances*, by specifying a maximum per-hour price they are willing to pay. EC2 then allocates the instances if the user's bid price is higher than the instance's current market-determined *spot price*. The spot price in EC2 fluctuates continuously in real time based on the market's supply and demand [8]. Thus, if the spot price rises above a user's bid price due to increased market demand, EC2 may revoke the spot server from the user after a brief two-minute warning (presumably to allocate it to a higher-priority user). EC2 bills for spot servers for each hour of use based on the current spot price at the start of each hour. Thus, a spot server's cost is based on the average spot price over the duration of its use, and *not* the user's bid price.

As a policy, EC2 caps the maximum bid at 10× the on-demand price of the instance type. Thus, users can never entirely eliminate the chance of a revocation by simply bidding a high price. EC2 operates a separate spot market (or *spot* *pool*) for each instance type in each availability zone, i.e., data center, of each geographic region, which results in more than 4000 global spot pools, each with its own dynamic spot price. Of course, EC2 also offers non-revocable on-demand instances, which we model as a distinct spot pool with a stable price and zero revocation probability.

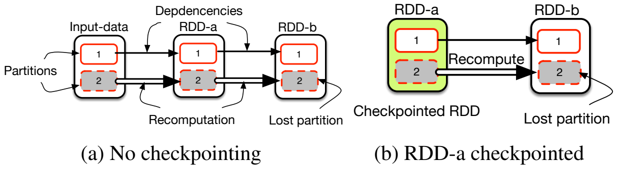
Transient servers in Google's Compute Engine (GCE) cloud are called *preemptible instances*. As with spot instances, GCE may revoke preemptible instances at anytime. However, preemptible instances differ in that GCE charges a fixed per-hour price for them, and their maximum lifetime is 24 hours. As we discuss, unlike prior work on intelligent bidding for spot instances, Flint's fault-tolerance and server selection policies also apply to GCE. Due to their revocable nature, both spot servers in EC2 and preemptible instances in GCE are significantly cheaper than their non-revocable on-demand counterparts (up to 70% in GCE and up to 10× in EC2). Hence, transient servers provide an opportunity to run BIDI workloads in the cloud at a very low cost.

2.2 Spark Background

Spark [36] is a general-purpose data-parallel processing engine that supports a rich set of data transformation primitives. Spark's growing popularity is due to its performance and scalability, as well as the ease with which many tasks can be implemented as Spark programs. For example, Spark supports batch and MapReduce jobs, streaming jobs [37], SQL queries [7], graph processing [34], and machine learning [21] tasks on a single platform with high performance.

Spark *programs* access APIs that operate on and control special distributed in-memory datasets called Resilient Distributed Datasets (RDDs). Spark divides an RDD into *partitions*, which are stored in memory on individual servers. Since RDDs reside in volatile memory, a server failure results in the loss of any RDD partitions stored on it. To handle such failures, Spark automatically recomputes lost partitions from the set of operations that created it. To facilitate efficient recomputation, Spark restricts the set of operations, called *transformations*, that create RDDs, and explicitly records these operations. In particular, each RDD is an immutable read-only data structure created from data in stable storage, or through a transformation on an existing RDD.

Spark records transformations in a *lineage graph*, which is a directed acyclic graph (DAG) where each vertex is an RDD partition and each incoming edge is the transformation that created the RDD. Importantly, transformations are coarse-grained in that they apply the same operation to each of an RDD's partitions in parallel. Thus, Spark may use the lineage graph to recompute any individual RDD partition lost due to a server failure from its youngest ancestor resident in memory, or, in the worst case, from its origin data on disk. In addition, Spark allows programmers to save, i.e., checkpoint, RDDs, including all of their partitions, to disk, e.g., in a distributed file system, such as HDFS [27]. In this case, rather than recompute a lost RDD partition from its origin data (or its youngest ancestor resident in memory), as depicted in Figure 1a, Spark may recompute it from its youngest saved ancestor, as depicted in Figure 1b.



(a) No checkpointing

(b) RDD-a checkpointed

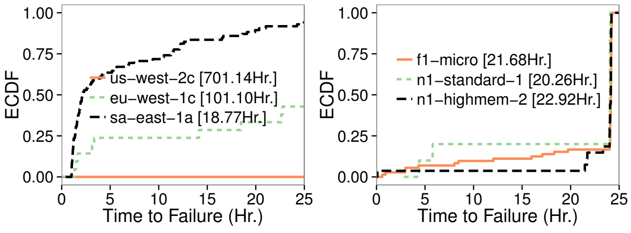
Figure 1. The loss of RDD-b's partition #2 results in recomputation using lineage information. Partitions may be computed in parallel on different nodes.

Spark's RDD abstraction is versatile and has been used for long-running "big-data" batch jobs, as well as interactive data processing. Interactive jobs may come in several varieties. For example, users can used a Read-Eval-Print-Loop (REPL) for interactive and exploratory analysis. As another example, Spark can be employed as a database engine with SQL queries executed via a translation layer such as Spark-SQL. Both examples require the Spark cluster to remain available for long periods of time: an exploratory REPL analysis may take several hours, and a database engine must be continuously available. Hence, if transient servers are used as cluster nodes, there is a risk of losing in-memory state, requiring significant overhead to regenerate and thus severely degrading interactivity. Flint includes automated policies to mitigate and respond to resources losses due to transient server revocations.

2.3 Flint Overview

Flint's goal is to optimize running BIDI workloads on transient servers. We design Flint as an application-aware framework for running Spark-based BIDI workloads that leverages Spark's built-in *mechanisms* to implement new *policies* to mitigate the impact of server revocations on performance. Flint's objective is to execute BIDI workloads, in this case in the form of Spark programs, at near the performance of on-demand servers, but at a price near that of transient servers. To achieve its objective, Flint provisions a fixed number of servers *N* to execute each BIDI job. Unlike a traditional cloud server, however, in Flint's case, these servers are transient and may be revoked at any time.

Since Flint's objective is to achieve performance near that of on-demand servers, on a revocation, it always requests and provisions a new transient server to maintain a cluster size at *N*. To achieve the highest performance at the lowest cost, Flint prefers transient servers with low prices, which are much cheaper than on-demand servers, and low revocation rates, which are near the 0% revocation of on-demand servers. Of course, in practice, the lowest prices might not always be associated with the lowest revocation rates. Thus, Flint must select transient servers to minimize the overall cost of running a BIDI job that takes into account the aver-



(a) EC2 spot instances (b) Google preemptible VMs

Figure 2. Empirically obtained availability CDFs and MTTFs of transient servers on Amazon EC2 and Google GCE.

age per-hour price of each transient server and the overhead of recomputing lost work based on the revocation rate.

As noted earlier, Spark exposes an interface to checkpoint RDD state to disk, but leaves it to the programmer to determine what RDDs to checkpoint, when, and how frequently. Flint exploits this flexibility to implement an intelligent, automated checkpointing strategy tailored for transient servers. While EC2 provides a two minute revocation warning, it is not sufficient to complete Spark checkpoints of arbitrary size and restarting from incomplete checkpoints is not safe. Google provides an even smaller warning of only 30 seconds. Thus, Flint does periodic checkpointing in advance so there is always some checkpoint of previous RDDs.

In general, the overhead of recomputing lost work due to a transient server revocation poses a challenging problem, since it requires Flint to balance the overhead of checkpointing RDDs with the time required to recompute them. At low revocation rates, checkpointing too frequently increases running time by introducing unnecessary checkpointing overhead, while similarly, at high revocation rates, checkpointing too rarely increases running time by causing significant recomputations. In addition, for *interactive* BIDI jobs, Flint must consider not only the overall cost of running a program to completion, but also the *latency* of completing each action within the program. For example, an interactive program might trade a small increase in overall cost (and running time) for a more consistent latency per action.

Finally, we structure Flint as a managed service that provisions and manages clusters on behalf of end-users executing BIDI jobs. As we discuss, Flint differs from EC2's Spark on Elastic MapReduce (Spark-EMR [2]) which runs unmodified Spark. Unlike Spark-EMR in which users have to select transient servers and handle the effects of their revocations, Flint embeds policies and mechanisms for selecting transient servers based on their price/revocation characteristics (monitored in real-time), and determines when and how often to checkpoint RDDs.

3. Flint Design

Flint is an application-aware framework for executing BIDI jobs on transient cloud servers. Flint's current design supports Spark-based BIDI jobs, implements application-aware, i.e., Spark-aware, policies for selecting and provisioning which transient servers to run on (based on their price and revocation rate characteristics), and determines when and how frequently to checkpoint application state, e.g., RDDs, based on expected transient server costs and revocation rates. To ensure transparency to end-users, Flint runs unmodified Spark programs. While Spark exposes a checkpointing interface for RDDs (via the `checkpoint()` operation), it requires the programmer to explicitly use it in Spark programs. In contrast, Flint automates the use of this checkpointing mechanism by intelligently determining what RDDs to checkpoint and how often to do so.

In addition, Flint's transient server selection policy runs as a separate node manager that monitors transient server characteristics, such as the recent spot price history for different instance types on EC2, to initially select transient servers for the cluster and to replace revoked transient servers while the program is running. We first discuss Flint's checkpointing and server selection policy for batch applications, and then extend it to support interactive applications.

3.1 Batch Applications

For batch BIDI jobs, Flint's goal is to execute batch-oriented Spark programs with near the performance of on-demand servers, but at a cost near that of transient servers. In this case, Flint provisions a *homogeneous* cluster of transient servers for each user. Since all transient servers in the cluster are of the same type, and the same bid price is used to provision them, it follows that when the market-driven spot price rises above this bid price, *all* servers in the cluster will be simultaneously revoked. Flint's checkpointing policy, discussed below, is derived from this insight and is specifically designed to handle the case where all servers of a cluster are concurrently revoked. We then outline the optimal server selection policy for batch applications, which leverages our assumption above that all transient servers are homogeneous.

3.1.1 Checkpointing Policy

Running a batch-oriented Spark program on a homogeneous cluster of transient servers, where a revocation causes the entire cluster to be lost simultaneously, is analogous to a single-node batch job that experiences a node failure—in both cases the job loses all of its compute resources. We adapt a well-known result from the fault tolerance literature [12] for deriving the optimal checkpointing interval for single node batch jobs for a given Mean-Time-to-Failure (MTTF). This optimal checkpointing interval minimizes the running time of a batch application when considering the rate of failures (or revocations), the overhead of checkpointing, and overhead of recomputation. Note that minimizing a batch application's running time also minimizes its cost on cloud platforms, since cost is structured as a price per unit time of use.

For a single-node batch job, running on a server with a given MTTF and a time to checkpoint δ, a first-order approximation of the optimal checkpointing interval is $\tau_{\text{opt}} \sim$

$\sqrt{2 \cdot \delta \cdot \text{MTTF}}$ [12]. This approximation assumes the time to write the checkpoint is constant at all intervals and $\delta \ll MTTF$; if the MTTF is smaller than δ then there is no guarantee the job will finish, as it will continue to fail before completing each checkpoint and not make forward progress. In Flint's case, the $\delta \ll MTTF$ constraint holds, since δ is on the order of minutes (to write RDD partitions of varying sizes to remote disk) and the MTTF for transient servers in EC2 and GCE is on the order of hours (see Figure 2).

Note that in the single-node case, the optimal checkpointing interval depends only on the MTTF and the checkpointing time δ, and not the running time of the job. In Flint's case, we can derive the expected MTTF of each type of transient servers on both EC2 and GCE. Since Amazon EC2 revokes spot instances whenever their spot price rises above a user's bid price, we can use historical prices for each instance type to estimate their MTTF for a given bid price. Amazon provides three months of price history for each spot market, and longer traces are available from third-party repositories [17]. While GCE does not expose a similar type of indirect information about revocation rates, we know that GCE always revokes a server within 24 hours of launching it. In addition, users may estimate the MTTF for a small cost by issuing requests for different server types and recording their time to revocation. We performed such measurements, and found that currently in GCE the MTTF is near 24 hours (see Figure 2). In contrast, the MTTF varies much more widely between server types in EC2 due its dynamic pricing. For example, with a bid price equal to the on-demand price for the equivalent server, the MTTF ranges from 18–700 hours.

In addition to deriving the MTTF, Flint must also determine what in-memory state to checkpoint during each interval τ, which dictates the checkpointing time δ. Flint's checkpoint policy for batch applications is as follows.

Policy 1: *Every τ time units, checkpoint RDDs that are at the current frontier of the program's lineage graph.*

Thus, rather than checkpoint all state in the RDD cache on each server, which spans both memory and disk, every τ time units, Flint only checkpoints each new RDD at the *frontier* of the lineage graph every interval. The frontier of the lineage graph includes the most recent RDDs for which all partitions have been computed, and whose dependencies have not been fully generated.

Thus, in the lineage graph, the frontier includes all RDDs that have no descendants, i.e., the current set of sink nodes in the graph. Note that although the complete lineage graph is not known *a priori* since it generates new RDDs and evolves dynamically as the program executes, the lineage graph's frontier is always well-known. Specifically, Flint signals that a checkpoint is due every interval τ. After signaling, each new RDD generated at the frontier of its lineage graph is marked for checkpointing. Note that RDDs that are already (or are in the process of) being computed have no guarantee of being in memory, and may require recomputation. Spark maintains a cache of RDD partitions on each server that swaps RDD partitions to and from disk based on their usage, and may delete RDD partitions if the cache becomes full.

Once each RDD at the frontier of each lineage graph has been checkpointed, Flint will not checkpoint any subsequent RDDs that are derived from them in the lineage graph until the next interval τ. We assume here that the computation time for RDDs does not exceed the checkpointing interval τ. Since most RDD transformations, such as map and filter, have narrow dependencies, the computation time for any single RDD is brief. However, we treat shuffle actions with wide dependencies as a special case, since each RDD partition that results from a shuffle depends on all partitions in the dependent RDD, resulting in a longer computation time. Because shuffles involve a larger amount of recomputation due to their wide dependencies, we checkpoint shuffle RDDs more frequently at an interval of τ divided by the number of RDD partitions that are being shuffled *from*. If server revocations occur *during* a shuffle operation (which act as barriers), then it is possible that the other nodes might end up waiting until the shuffle data is recomputed, as in a bulk synchronous parallel system. In all other cases, the recomputation operation does not cause waiting.

Finally, unlike in the optimal formulation above, the number of RDDs and the time required to write them to disk, i.e., the checkpointing time, is not static, but dictated by each program. Thus, Flint maintains a current estimate of the checkpointing time δ based on the time it takes to write all RDD partitions, which have a well-known size, in parallel to the distributed file system. As δ changes, Flint dynamically updates the checkpointing interval τ as the application executes. Although an accurate δ estimate improves the accuracy of the checkpoint interval τ, we note that τ is proportional to the square root of δ, which reduces estimation errors. More importantly, since we only checkpoint when RDDs are generated and not at arbitrary times, the system is not particularly sensitive to an accurate estimation of τ.

Flint supports general Spark programs with arbitrary characteristics. Thus, Flint's dynamic checkpointing interval automatically adapts to the characteristics of the program, checkpointing more or less frequently as the overhead due to checkpointing falls and rises, respectively. Note that, since RDDs are read-only data structures, the checkpoint operation in Spark is asynchronous and does not strictly block the execution of other tasks. However, checkpointing tasks consume CPU and I/O resources that proportionally degrade the performance of other tasks run as part of a Spark program.

3.1.2 Server Selection Policy

Based on the checkpointing policy above, we can compute the expected percentage increase in running time for a Spark program when running on transient servers with different price and revocation rate characteristics. Our goal is to choose a single type of transient server that has the least increase in running time (and thus, the least cost) to provision a homogenous cluster to execute the program. Specifically, in the case of spot instances, for a market k with an $MTTF_k$ based on the revocation rate at a certain bid price over the recent spot price history, the overall expected running time $E[T_k]$ for the program with a running time of T without any revocations is as follows.

$$E[T_k] = T + \frac{T}{\tau} * \delta + \frac{T}{MTTF_k}\left(\frac{\tau}{2} + r_d\right) \qquad (1)$$

The first term is the running time of the program without any revocations, the second term is the additional overhead over the running time of the program due to checkpointing, and the third term is the additional overhead over the running time T due to provisioning new replacement servers (r_d) and recomputing the lost work, assuming that revocations are uniformly distributed over the checkpointing interval τ. The delay r_d for replacing a server is a constant—for EC2, it is typically two minutes. Factoring out T yields, $E[T_k] = T(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k}(\frac{\tau}{2} + r_d))$. Thus, we only need to know δ in addition to $MTTF_k$ to compute the percentage increase in running time on a market k. We conservatively estimate an initial δ by assuming our Spark cluster is properly sized for the application, and derive δ assuming that all memory is in use by active RDD partitions that must be checkpointed. We record the computation time for each RDD partition, and assume that the recomputation time for a partition will be the same as its initial computation time, given the same resources available. The immutable nature of RDDs, the lack of external state dependencies, and the static RDD dependency graph means that we can safely make this assumption.

Given $E[T_k]$ above, if the average per-unit price of each market k is p_k, we can derive the expected cost simply as :

$$E[C_k] = E[T_k] * p_k = T(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k} * \frac{\tau}{2}) * p_k \quad (2)$$

Since T is a constant, minimizing $E[C_k]$ requires choosing the market where the product of p_k and $(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k} * \frac{\tau}{2})$ is minimized. To do so, Flint simply evaluates this product across all potential spot markets, and provisions all servers in the cluster from the market that yields the minimum overall cost. Of course, for each market, a different bid price yields a different $MTTF_k$ and p_k. By default, Flint bids the equivalent on-demand price for all spot instances, as spot instances are cheaper if the spot price is less than the on-demand price. Note that we include on-demand instances as a spot market with an infinite MTTF (where checkpointing is not required). If the average price p_k exceeds the on-demand price, Flint transitions to using on-demand instances.

Note that, selecting servers from any other market than the one that yields the minimum overall cost will increase the overall cost for executing a batch application. Since batch applications are delay tolerant and are concerned with overall running time and cost, they can tolerate simultaneous revocations of all servers—the job can resume from a prior checkpoint. This insight enables us to model parallel Spark programs similarly to single-node batch applications. The performance characteristics under different number of simultaneous failures are shown in Section 5.3.

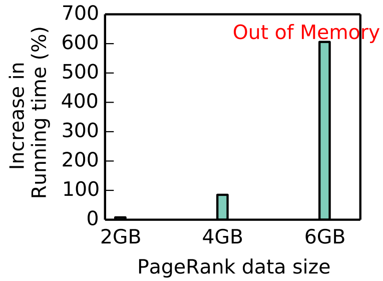


Figure 3. Simultaneous server revocations substantially increase running time if Spark runs out of available memory.

Restoration Policy. Whenever a revocation event occurs, Flint uses the same process as above to immediately select a new market to resume execution. When selecting a new market, Flint does not consider the market that experienced the revocation event, since the instantaneous price of that market has risen and the servers are not available. In addition, while Flint bases its selection on the average market price over a recent window, it does not consider markets with an instantaneous price that is not within a threshold percentage, e.g., 10%, of the average market price. In the worst case, where prices across all markets are high, Flint resumes execution on on-demand servers, which are non-revocable.

3.2 Interactive Applications

While large simultaneous revocations do not degrade the running time of batch applications (relative to the same number of individual revocations), they do degrade the response latency of interactive BIDI jobs. Large simultaneous server revocations result in the need to concurrently recompute many RDD partitions, creating contention for resources on the surviving servers, which must multiplex their current work with recomputing lost RDD partitions. In the worst case, if the RDD working set is larger than the available memory on the remaining servers, Spark must swap RDD partitions between memory and disk as necessary to execute each task. Figure 3 shows the impact on performance under memory pressure due to large revocations. Such swapping increases the latency for interactive BIDI jobs.

Thus, applying the above policies designed for batch BIDI jobs to interactive ones will result in highly variable response latencies. Hence, rather than minimizing the expected cost and running time on transient servers, interactive BIDI jobs also value minimizing the variance between the maximum latency and the average latency of actions to provide more consistent performance, as opposed to excessively long latencies for some actions and short latencies for others. We can reduce this variance in latency by constructing a *heterogeneous* cluster for each interactive BIDI job—by mixing together different types of transient servers, e.g., from different spot markets, in the same cluster. Assuming that the demand and supply dynamics, and hence spot prices, for different transient server types are uncorrelated, a price spike in one market is independent of others. Hence, the revocation of one type of transient server due to a price increase in its market is independent of others and the cluster will only lose a fraction of the servers on each revocation event. In this case, the Spark program will continue execution on the remaining nodes, albeit more slowly, and can resume normal execution after Flint restores the revoked nodes to the cluster. We describe this policy as follows.

Policy 2: *Reduce risk through diversification—choose transient servers of different types with uncorrelated prices to reduce the risk of simultaneous revocations.*

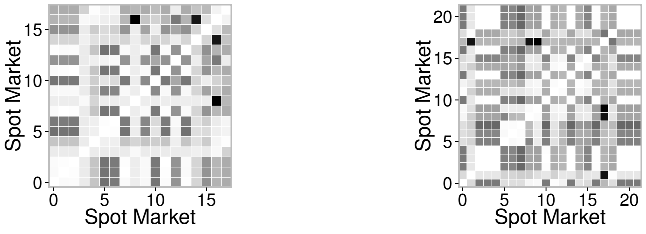
Figure 4 shows that revocations on many (but not all) spot markets in EC2 are in fact independent and uncorrelated. However, as discussed above, selecting any market other than the one that minimizes the overall cost will increase the overall cost and running time of the application. Thus, even for interactive BIDI jobs, it is important to intelligently mix transient servers from different markets to reduce the variance in latency without significantly increasing the overall cost and running time of the application.

3.2.1 Checkpointing Policy

Before discussing our selection policy for interactive applications, we must first determine the appropriate checkpointing policy. Of course, we could also reduce our variance in latency by checkpointing more frequently, and thus enabling recovery from each large revocation event by simply reading in the checkpointed state. However, checkpointing too frequently degrades the average case performance. Instead, we extend the same checkpointing policy as above, assuming that we equally divide our cluster of size N across transient servers selected from m markets. In this case, we must estimate the aggregate MTTF for the heterogeneous cluster by computing the harmonic mean of $MTTF_1 \cdots MTTF_m$ of each individual server type within the cluster.

$$MTTF = \frac{1}{\frac{1}{MTTF_1} + \cdots + \frac{1}{MTTF_m}} \tag{3}$$

Note that the aggregate MTTF will be smaller than the MTTF for each individual market, in that there will be more revocation events, but each one will only result in the revocation of N/m servers. Thus, our checkpointing interval above will decrease, causing more frequent checkpoints. However, the size of each revocation event will also decrease, compared to using a single market. If we assume the overhead of recomputation for a Spark program is linear in the number of revoked servers, then when using more markets, the overhead of recomputation due revocation events decreases, while the number of revocation events increases. This decrease in the recomputation overhead for each event tends to balance out the increased number of revocation events due to the lower MTTF, although we leave a formal proof of this property to future work.



(a) *us-east-1a*

(b) *m2.2xlarge*

Figure 4. Publicly available EC2 spot price traces show that prices (and hence revocations) are pairwise uncorrelated (shown by darker squares) for most pairs of markets.

3.2.2 Server Selection

To intelligently provision a heterogeneous cluster, we first construct a subset L of mutually uncorrelated markets among all the possible markets. We construct L for two reasons. First, published price histories show that revocations usually do not happen simultaneously in different spot markets (in Figure 4, darker squares indicate less correlated failures between any two markets). This observation confirms the feasibility of diversifying across markets to reduce concurrent revocation risk. Second, since there are potentially many markets to choose from—over 1000 markets in EC2's US-east region alone—constructing a smaller set of L markets prunes the search space. We greedily construct L by adding the most pairwise uncorrelated markets to L.

As before, we do not consider markets where the instantaneous risk of revocation is high, i.e., the spot price is not within some threshold of the average spot price. We then sort these candidate markets in order of their expected cost using the same approach as above for batch applications. After sorting the candidate markets, we greedily add markets to our set of selected markets S in order, as follows.

We first select the market that yields the minimum expected cost and then compute the expected variance in its running time based on the market's revocation rate. We compute the variance in the expected running time $\sigma^2 = E[T(S) - E[T(S)]]^2 = E[T(S)^2] - E[T(S)]^2$ for a set of markets S, where servers are equally distributed among the markets in S. In Equation 1 we have shown the scenario for a single market k; extending it to $m = |S|$ markets yields:

$$E[T(S)] = T + \frac{T}{\tau} * \delta + \frac{T}{MTTF(S)} \cdot \frac{1}{m} \cdot \left(\frac{\tau}{2} + r_d\right) \tag{4}$$

With multiple independent markets, when one market fails, only $1/m$ fraction of the servers are lost (because they are equally distributed among all markets). Note that the MTTF for multiple markets above is given by Equation 3.

We then select the market with the next lowest cost, and equally divide our servers between the two markets. For the mixed cluster, we again compute the expected variance in running time. If the expected variance is higher than the single market, we stop and do not add the second market; if the expected variance is lower, we evaluate the expected variance from dividing the servers among the three lowest cost markets. We continue adding markets until adding an additional market does not decrease the variance in running time. We also stop if the expected cost ever rises above the expected cost of running the application on on-demand servers. As shown in our evaluation, the result of this server selection algorithm is a mix of servers from different markets that decrease the variance in running time, providing consistent response latency, without significantly increasing the cost relative to the optimal cost in the batch case.

Restoration Policy. In addition to determining the initial mix of servers from different markets for the cluster, Flint must also replace a set of revoked instances from a particular market with instances from another market. To do so, Flint simply replaces these revoked instances with instances from the lowest-cost unused market in set *L*. As above, Flint does not consider markets with instantaneous prices that are significantly higher than their average price.

Bidding Policy. Flint uses a simple bidding policy to place bids for each spot server—we bid the on-demand price. Bidding in the current EC2 spot market has only a negligible effect on the average cost and the MTTF—these metrics stay the same for a very large range of bids. In Figure 11, which shows the expected costs for three instance types, we can see that the range of bids for which the cost remains unaffected is quite large. For example, bidding anywhere from 0.5 to 2× the on-demand price for the `m2.2xlarge` instance type yields the same cost. For all three types shown in the figure, the on-demand price is inside the wide range that yields the minimum price. Our simple bidding policy is thus motivated by the insensitivity of expected cost and MTTF on the bid [25], as well as a focus on systems mechanisms to handle spot revocations which can work even in environments where no bidding mechanisms apply, as in the case of GCE Preemptible Instances, which have a fixed price.

If market characteristics were to change, a modification to the simple bidding strategy might be necessary. Since Amazon provides up to three months of price history, the empirical relation between bids and the average price and MTTFs can be used to select bids that will minimize the expected cost using Equation 2. A similar approach can be found in [26, 25]. Lastly, we bid the same price for all the instances in a given market. However, a more sophisticated policy could stratify the bids within a market such that instances with different bid prices fail at different times. However, stratifying bids is not currently effective, as price spikes in the current spot markets are large and cause servers with a wide range of bids to all fail simultaneously [26].

Arbitrage. Flint reduces costs by using low-cost spot instances and spreads revocation risks by exploiting the uncorrelated prices across different spot instance markets. A concern is that neither of these characteristics of the spot markets might hold if systems such as Flint gain in popularity and the demand for spot instances increases. This increased demand might drive up market prices and volatility, causing the cost savings of spot servers to vanish. However, we believe that as infrastructure clouds continue to grow and add capacity, the surplus capacity (which is sold as spot servers) would also grow, such that the increase in demand would be matched by an increase in supply. Furthermore, systems like Flint only represent a small portion of users of the spot market: other users and systems utilize spot instances in different ways and have different spot instance demand characteristics. A more detailed analysis of the second-order market effects and other "game-theoretic" analysis is outside the scope of this paper, and we assume that the large numbers and sizes of the spot market will absorb the effects of the arbitrage opportunities Flint exploits.

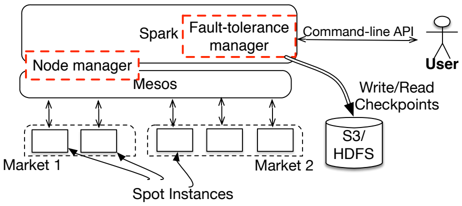


Figure 5. Flint architecture and system components.

4. Flint Implementation

We implemented a Flint prototype based on Spark 1.3.1 in about 3000 lines of Scala and Python. The prototype includes the policies for batch and interactive Spark applications from the previous section. Users interact with Flint via the command-line to submit, monitor, and interact with their Spark programs. Flint's implementation is split into two main components: a node manager that interfaces with Mesos and EC2, and implements the server selection policy, as well as a fault-tolerance manager embedded in Spark that implements the checkpointing policy (Figure 5). Our implementation primarily integrates with EC2 and supports spot instances. However, our approach is compatible with GCE, which offers transient servers at a fixed price per unit time.

To implement the server selection policy, the node manager accepts user requests for Spark clusters of size *N* to run their application, and selects the specific spot market(s) to provision the servers. To implement the batch and interactive server selection policy, the node manager monitors the real-time spot price in each EC2 spot market and maintains each market's historical average spot price and revocation rate (and MTTF) over a recent time window, e.g., the past week. The node manager acquires one or more servers in a particular market by placing a bid for them in the spot market at the on-demand price via EC2's REST API. Each Spark cluster in Flint runs in its own Virtual Private Cloud (VPC) that is isolated from other users. Flint launches the instances with its own customized disk image, e.g., an Amazon Machine Image (AMI), which contains a pre-configured version of a Spark master and worker. After the initial setup, Flint provides users with a web interface, as well as SSH connectivity, to the master and worker nodes to monitor job progress and use Spark interactively via the Spark shell.

Flint's fault-tolerance manager is written as a core Spark component so it may interact with Spark's internal APIs for scheduling, RDD creation, and checkpointing. The fault-tolerance manager monitors the set of RDDs at the frontier of the lineage chain, checkpoints them to stable storage every interval τ, and updates δ and τ as new RDDs are created. To compute τ, the fault-tolerance manager must interact with the node manager to retrieve the MTTF of each server in the cluster. To implement the checkpointing interval, the fault-tolerance manager maintains an internal timer for τ, and marks the first RDD in the queue from each active stage after the timer expires for checkpointing. If Flint marks an RDD for checkpointing, it checkpoints each individual partition of that RDD. To support automated checkpointing, we modify Spark's existing checkpointing implementation to enable fine-grained partition-level checkpointing. In Flint, once a task finishes computing its partition, it notifies Spark's DAG scheduler, which then invokes the fault-tolerance manager to check if it has marked the corresponding RDD for checkpointing. If so, it creates a new checkpointing task, which handles the asynchronous checkpoint write operation.

Checkpoint Garbage Collection. We have also implemented a garbage collector for checkpointed RDDs to reduce storage requirements. Checkpointing an RDD terminates its lineage graph and its ancestor RDDs are no longer "reachable." Checkpoints for these unreachable RDDs are redundant and thus periodically removed. Lastly, to mitigate the impact of spot instance revocations, the node manager monitors the 120 second spot termination warning provided by EC2's `/spot/termination-time` API. If Flint detects a warning on any worker, it immediately triggers the market selection on the node manager which selects and requests replacement instances. As part of this notification, the fault-tolerance manager informs the node manager of the most current values for δ and τ based on the collective size of the RDDs at the frontier of the lineage chain. The node manager needs these values to execute its server selection policy to replace revoked servers.

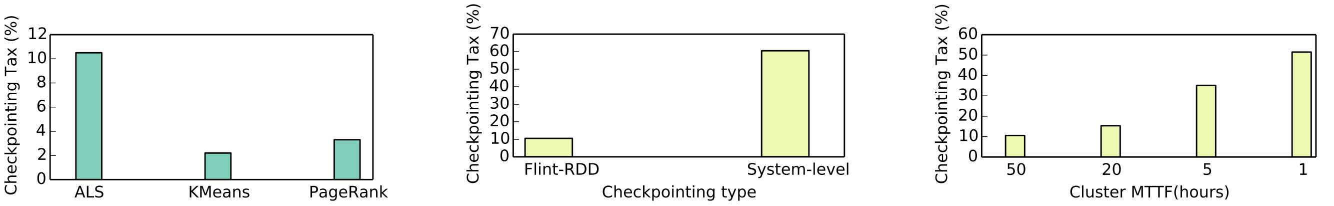
Checkpoint Storage. Flint stores all partition checkpoints that belong to a single RDD inside the same directory on HDFS. On recovery, we first check if the partition exists in the corresponding directory before any starting any RDD (re)computation. We use Elastic Block Store (EBS) volumes and treat them as durable storage. Using EBS instead of local, on-node storage has several advantages. Data on local disks is lost upon revocation, and a revocation event can cause the data loss such that HDFS cannot recover even using 3-way replication. In addition, the amount of local storage in EC2 is limited, e.g., 10GB on most nodes. However, unlike local storage which is free, there is an extra cost for EBS volumes which depends on their size and I/O rate. Because we use EBS to only store checkpoints, which we frequently garbage collect, the EBS disk size required is small. In addition, the I/O rate is limited as Flint judiciously regulates checkpointing frequency. We use the two minute revocation warning to pause all nodes, flush data, and cleanly unmount all EBS volumes. After revocation, we first let HDFS recover the missing chunks. If that fails for any reason, since the data on EBS volumes persists even after revocations, we copy all the data from the EBS volumes to the newly launched instances [5].

The SSD EBS volumes which Flint uses are currently charged $0.10 per GB per month by Amazon. Because Flint provides Spark as a managed service, these EBS volumes are reused among jobs, and the EBS costs are thus amortized. EBS volumes required for storing Flint checkpoints cost about 1-2% of on-demand instances, adding an overhead of about 10-20% to the final cost using spot instances. A more detailed cost analysis and breakdown of storage is presented in Section 5. We are not constrained in the choice of checkpoint storage and there are other options that are feasible as well. For example, Amazon's S3 object store is about 20 times cheaper than EBS, and is a viable option for reducing storage costs, albeit at worse read/write performance. Amazon's Elastic MapReduce File System (EMRFS [2]) uses a combination of S3 and DynamoDB database for low-cost storage for Spark. A similar storage configuration can be used for storing Flint checkpoints at low cost.

5. Experimental Evaluation

We conducted our experiments by running popular Spark programs on Amazon EC2 to quantify Flint's performance and cost benefits for both batch and interactive BIDI workloads. We run all experiments on a Spark cluster of 10 `r3.large` instances in EC2's US-East region. Each `r3.large` instance has 2 VCPUs, 15GB memory, and 32GB of local SSD storage. We use persistent network-attached disk volumes from Amazon's Elastic Block Store (EBS) to set up the HDFS filesystem (with a replication factor of three) and use it to store RDD checkpoints.

Our evaluation includes systems experiments using our Flint prototype to evaluate the effect of recomputation and checkpointing on real Spark applications, as well as simulation experiments to examine the cost and performance characteristics of Flint over long periods under realistic market conditions. We use a range of batch and interactive workloads in our evaluation, as described below. The input data sizes for each workload listed below were carefully chosen to max out the total cluster memory used by intermediate RDDs and to ensure stable Spark behaviour even under node revocations.



(a) Performance overhead due to Flint's RDD checkpointing.

(b) System level distributed checkpointing overhead is large compared to RDD checkpointing.

(c) Checkpointing overhead for the ALS workload increases with increasing market volatility.

Figure 6. Performance overhead of system- and application-level checkpointing.

5.1 Workloads

PageRank. PageRank is a graph-processing workload that computes the rank of each page in a web graph iteratively based on the rank of the pages that link to it. PageRank is a good candidate for evaluating our checkpointing policy, since it creates a large number of RDDs—proportional to the number of vertices in the graph—and involves a large number of shuffle operations. We use the optimized PageRank implementation from Spark's graphx library. For our experiments, we use the Live Journal [3] dataset of size 2GB.

KMeans Clustering. KMeans is a clustering algorithm that partitions data points into k clusters with the nearest mean. We use KMeans clustering as an example of a compute-intensive application: it consists of applying a series of narrow dependencies to an RDD and then a large shuffle operation per iteration. We use the optimized implementation from Spark's `mllib.DenseKMeans` library with a randomly generated dataset of size 16GB. KMeans is a prototypical example of a iterative machine learning technique.

Alternating Least Squares. Alternating Least Squares (ALS) is a linear regression model that fits a set of data points to a function with the minimum sum of squared errors between the model and the data points. ALS's RDD lineage graph is similar in structure to KMeans. However, ALS is more shuffle-intensive where each transformation takes more time than with KMeans. We use Spark's `mllib.MovieLensALS` implementation on a 10GB dataset.

TPC-H. We use Spark as an in-memory database server that services clients issuing SQL queries from the TPC-H database benchmark with a data size of 10GB [4]. Since TPC-H queries are data-intensive, to accelerate query execution, Flint de-serializes and re-partitions the raw files stored on disk first and then persists them in memory as RDDs. Each time a new query arrives, Flint executes it using in-memory data rather than loading the data from disk again. TPC-H is an interactive workload where the query response latency is the primary metric, rather than the running time. The workload is shuffle- and join-intensive, as many SQL queries translate to shuffle and join operations on RDDs.

5.2 Quantifying the Checkpointing Overhead

We first verify and quantify the overhead due to checkpointing RDDs in Flint and compare it with both the performance of running on on-demand servers without checkpointing, and with a systems-level checkpointing approach. The overhead due to checkpointing dictates how close Flint's performance on transient servers comes to the performance of on-demand servers. For these experiments, we use a relatively low MTTF of 50 hours to highlight the checkpointing overhead—the MTTFs in current EC2 spot markets range from 20 to 2000 hours.

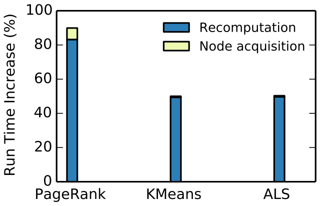


Figure 7. Recomputation of lost RDD partitions due to a single revocation causes a 50-90% increase in running time.

We first measure the checkpointing overhead for three batch workloads when the MTTF is equal to 50 hours using Flint's intelligent checkpointing algorithm. As Figure 6a shows, the performance overhead due to checkpointing, as percentage increase in running time, for all three batch applications is small, ranging from 2% to 10%. Of these three applications, ALS has the largest collective set of RDDs and hence also has the highest checkpointing overhead. Due to the larger data sizes and higher network utilization (the most constrained and bottlenecked resource for Flint), the checkpointing overhead for ALS is also the highest.

Next, we compare the performance overhead of Flint's intelligent application-level checkpointing with a systems-level approach using the same checkpointing frequency. A systems-level distributed checkpointing approach must checkpoint the entire memory state of each Spark worker, including all active RDDs, cached RDDs, shuffle buffers etc. In contrast, by checkpointing only the frontier of the RDD lineage graph from within the application, Flint can avoid checkpointing stale application state or unnecessary system state. As shown in Figure 6b, the systems-level approach increases the running time by 50% compared to our application-level approach that only checkpoints selective RDDs. The result demonstrates the benefit of leveraging fault-tolerance mechanisms that are already embedded into data-parallel frameworks for high failure-rate environments like transient servers.

Last, we measure the change in checkpointing overhead when running the ALS workload on transient servers with varying volatility. Figure 6c shows that, as expected, the checkpointing overhead increases as the transient servers become more volatile (with a higher revocation rate and a lower MTTF). With a highly volatile market, where the MTTF is 1 hour, Flint's checkpointing overhead increases from 10% to 50% of the application's typical running time. This result represents an upper bound on Flint's checkpointing overhead, since any further increase in the checkpointing overhead will exceed the RDD recomputation time.

Spreading the application nodes across multiple availability zones also does not seem to hurt application performance significantly. We found no noticeable decrease in the performance of KMeans, and only a 7% degradation for the ALS workload. While the inter-availability zone network latencies are certainly much higher compared to within a zone, we conjecture that the large sized checkpoint writes are bandwidth-sensitive and not latency-sensitive, and multiple availability zones can thus be used without a large performance penalty.

Result: *Flint's checkpointing overhead is low, increasing application running time between 2 and 10% even with relatively low MTTF values. In addition, even for extremely volatile markets, Flint's checkpointing overhead increases running time by less than 50%. Further, Flint's application-level approach significantly reduces the overhead relative to systems-level checkpointing (from a 50% increase in running time to a 10% increase in running time for ALS).*

5.3 Impact of Server Revocations

We now consider various transient server revocation scenarios and system configurations to demonstrate their impact on running time. We are interested in evaluating the overhead of recomputation triggered by server revocations. In all the experiments, revoked servers are replaced by new transient spot servers, such that Flint maintains a cluster size of ten.

Figure 7 shows the performance impact of a single server revocation out of a cluster of size ten without Flint's intelligent checkpointing policy. The figure illustrates that a single revocation can cause running time to increase sharply, up to 90% in the case of PageRank. Since Flint immediately replaces any revoked server, the increase in running time is due to two factors: i) recomputing RDD partitions lost due to the revocation and ii) the time to acquire replacement servers. For PageRank, the time to acquire a new server contributes 5% of the increase in running time with the rest of the increase coming from recomputing RDDs. For the other two applications, which have longer running times, the time to acquire replacement servers is negligible, and all of the increase is due to recomputing lost RDDs.

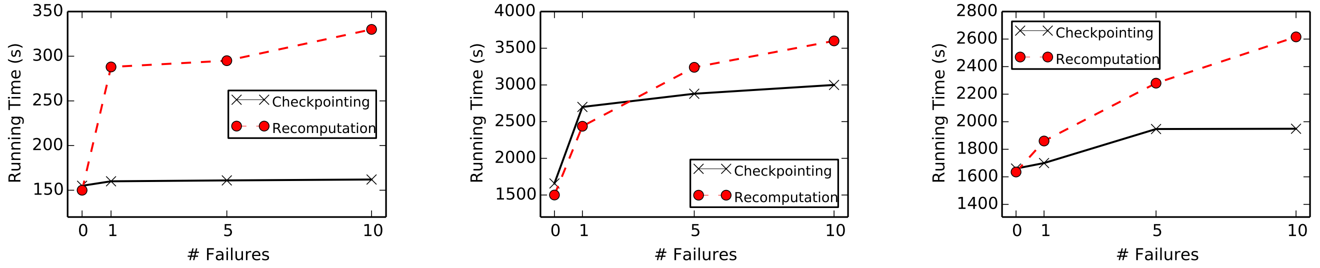
We also evaluate the impact of the number of concurrent revocations on performance. Figure 8 shows the total running time for the three batch applications when varying the number of concurrent server revocations without checkpointing. Here, a value of zero represents the baseline case with no failures. Figure 8 shows that application running time increases as the number of concurrent revocations increases, by up to 100%. The large overhead is due to the recomputation of lost RDD partitions, as well as their recursive dependencies. The graph also shows that running time is not strictly a linear function of the number of concurrent revocations: the impact on performance decreases with each additional revocation. Thus, for batch jobs, Flint's approach of using only a single market where all transient servers are concurrently revoked incurs less overhead than spreading servers across multiple markets with more frequent, but smaller, revocation events.

Result: *Without checkpointing, recomputation due to revocation of even a few servers, causes a significant increase in running time and cost. The impact on running time for batch applications tends to decrease as the size of the revocation event increases, which supports Flint's batch checkpointing policy for that selects spot instances from the same market.*

Figure 8 also compares the running time with and without Flint's checkpointing policy as the size of the revocation events increase. Since checkpointing bounds the amount of recomputation, the running time with checkpointing is significantly smaller than the recomputation-only configuration for all the three workloads. For PageRank(Figure 8a), checkpointing is particularly beneficial—periodically saving the shuffle output drastically reduces and bounds the recomputation required on a revocation. Similarly, checkpointing the RDDs in KMeans (Figure 8c) bounds the performance degradation when moving from 5 to 10 simultaneous failures. Further, the sublinear relationship between the size of the revocation event and the running time is even stronger when using checkpointing. That is, as the size of the revocation event increases, with checkpointing, the increase in running time flattens out, reflecting the bound on performance degradation due to checkpointing. Of course, with no revocation events, applying the checkpointing policy slightly increases running time due to the overhead of checkpointing, although this increase is not significant.

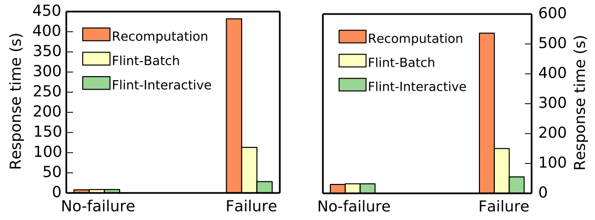
So far we have evaluated Flint's performance on a cluster with ten machines. As cluster size grows, the system scalability is governed by the scalability of the underlying Spark engine, as well as the checkpoint storage backend (HDFS in our case). Flint's policies for market selection are applicable when an application starts and after revocation events and thus incur little run-time overhead. Both Spark and HDFS have been known to scale well to cluster sizes in the hundreds of nodes [6]. However, we leave a more detailed analysis of Flint on larger cluster sizes to future work.

Result: *Flint's checkpointing policy significantly reduces the increase in running time due to revocations, by 15-100% for our three representative batch applications.*



(a) PageRank (b) ALS (c) KMeans

Figure 8. Application running times under various failure scenarios with and without checkpointing. Applications are running on a cluster of size ten. Zero indicates the baseline case of no failures.



(a) Short query (b) Medium-length query

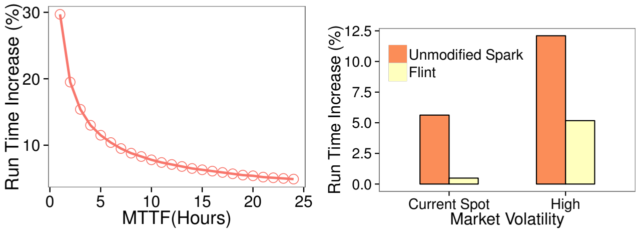
Figure 9. Flint's interactive mode results in 10-20× improvement in TPC-H response times during failures.

5.4 Support for Interactive Workloads

Checkpointing is even more essential for interactive applications. Figure 9 shows the response time of two queries—query three and query one of TPC-H—with and without revocations. In this case, our revocation scenario is either all ten servers are concurrently revoked (when using either recomputation without checkpointing or Flint's batch checkpointing policy), or a single server is revoked ten times (when using Flint's interactive policies).

Without revocations, the checkpointing overhead for Flint's batch and interactive modes is low (~10%). The response time without revocations is low for all three of our policies: recomputation without checkpointing, Flint's batch checkpointing policy, and Flint's interactive checkpointing policy. For a small query, the latency is a few seconds, and for a larger query, the latency remains less than ten seconds. However, with revocations, the response time rises substantially to 400-500 seconds for both query types without any checkpointing. The rise occurs because recomputing the RDDs lost due to revocation requires re-fetching the input data from Amazon's S3 storage service, and then again re-partitioning and de-serializing the data.

Using Flint's batch checkpointing policy, the response time reduces by a factor of 4×. In addition, using Flint's interactive checkpointing policy, which is explicitly designed to trade-off cost for interactive performance, reduces the response time even further: from 100-150 seconds with the batch checkpointing policy to 28-55 seconds with the interactive checkpointing policy. This additional reduction (3×) in the response time is due to the interactive checkpointing policy and the market selection that mixes different types of servers in the same cluster. Flint's batch policies select markets to minimize the expected cost, while Flint's interactive policy also considers the variance in response time when selecting markets. This experiment demonstrates the benefit of considering response time.



(a) Performance vs. MTTF (b) Flint vs. Unmodified Spark

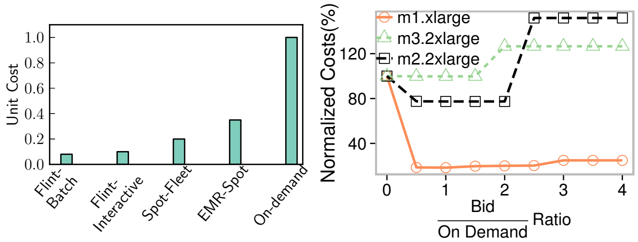
Figure 10. Flint's increase in running time compared to using on-demand servers is small in today's spot market, and is low even for highly volatile markets equivalent to GCE.

Result: *Flint's checkpointing and server selection policies decrease the response time of interactive workloads by an order of magnitude (~10×). Flint's interactive policy results in lower response times than its batch policy, since it spreads risk by mixing transient servers from different markets.*

5.5 Cost-Performance Tradeoff

To quantify the impact of Flint on cost and performance, we use traces of EC2 spot prices from January to June 2015. We also use empirically collected availability statistics for over 100 GCE Preemptible Instances that we requested and were revoked over a one month period in August 2015. In addition to examining Flint's cost and performance on real data, we also present results on synthetic data with lower MTTFs to demonstrate Flint's performance under extreme conditions, i.e., with high market volatility. For these experiments, we simulate the performance of a canonical program that checkpoints 4GB RDD partitions every interval.

We first demonstrate the decrease in running time as the MTTF of the transient servers increases. As shown in Figure 10a, once the MTTF extends beyond twenty hours, Flint's increase in running time is less than 10% compared to using on-demand servers. Since MTTFs of twenty hours are on the lower end for EC2 spot markets (assuming a bid equal to the on-demand price), Flint's performance on transient servers will be on par with on-demand servers. Figure 10b quantifies this performance by showing the increase



(a) Flint cost saving.

(b) Cost as a function of the bid price.

Figure 11. Flint determines the bid for each market based on our expected cost model. Flint is able to run both batch and interactive applications at 10% of the on-demand cost.

in running time when using Flint on spot instances compared to using on-demand servers. The graph shows that in the current spot market there is little increase (<1%) in running time when using Flint versus using on-demand servers. By contrast, when running unmodified Spark on spot instances (while still employing Flint's server selection policies), the increase in running time is over 5%.

Of course, the existing spot market in EC2 is underutilized and not particularly volatile. Thus, we also show results for a higher volatility market based on our GCE data with an MTTF close to 20 hours. In this case, unmodified Spark on spot instances has an increase in running time of 12%—Flint's increase in running time is <5%.

Result: *Flint causes a small increase in running time (1%-7%) compared to on-demand servers for transient servers with both high and low volatilities, represented by EC2 spot markets and GCE preemptible instances, respectively.*

Lastly, we quantify Flint's cost savings for batch and interactive workloads compared to running on equivalent on-demand instances. We compare Flint's server selection policies from Section 3 with multiple existing approaches for running Spark on EC2 spot instances. In particular, we compare against EC2's Elastic MapReduce (EMR) managed service to execute unmodified Spark programs on spot instances. Note that Spark-EMR on EC2 incurs an additional flat fee of 25% of the on-demand price per hour in addition to the cost of the spot instances. We also examine using SpotFleets in EC2, since this is an application-agnostic service that EC2 provides to automatically replace revoked spot instances with a spot instance from another market. Interestingly, this EC2 service, like Flint, automatically bids the on-demand price for spot instances on behalf of users. SpotFleets enable users to set a policy that automatically selects an instance from either the cheapest market or the least volatile market (without considering the impact of revocations on performance). Thus, comparing Flint with SpotFleet represents the benefit of embedding the server selection and replacement policy into Flint and making these policy decisions based on the characteristics of the application.

For this experiment, we configure SpotFleets to use two `r3` instance types in the fleet. Flint's cost-aware server selection (for both batch and interactive jobs) results in 90% cost savings compared to executing on on-demand servers. Combined with our previous result that showed the overhead of Flint compared to using on-demand servers in the current spot market, this demonstrates that Flint achieves its goal of executing BIDI workloads at a performance level near that of on-demand servers, but at a price near that of transient servers. In addition, Flint's batch and interactive policies also lower costs relative to using Spark-EMR on spot instances by 66%, and lower costs relative to using SpotFleets by 50%. These results are important in that they demonstrate Flint's cost savings are not simply due to the fact that spot instances are significantly cheaper than on-demand servers. Since Spark-EMR and SpotFleets also use spot instances, the savings stem solely from Flint's intelligent application-aware checkpointing and server selection policies.

At current spot prices, improving on the cost of using on-demand servers is not challenging—even simple strategies for using spot instances are capable of improving on on-demand costs. In contrast, by comparing with Spark-EMR and SpotFleets, we show that Flint not only results in lower costs than using on-demand servers, but also lower costs than using spot instances when using unmodified Spark and application-agnostic bidding strategies, respectively.

Flint uses EBS for checkpoint storage, which incurs an extra cost. Due to the low space requirements of periodic checkpointing and garbage collection, these storage costs are also low. EBS volumes cost $0.1 per GB *per month*, and because Flint provides Spark "as a service," these volumes can be reused among different jobs, and thus their monthly cost is amortized. The `r3.large` servers we use have 15GB of main memory, and we conservatively provision twice that memory for storing checkpoints. Note that Spark only uses 40% of RAM for storing the RDD data—the rest is used as an RDD cache—thus we effectively over-provision by more than a factor of four, and can always add more EBS volumes if storage space is running low by dynamically extending HDFS. The hourly cost for EBS volumes has an overhead of $0.1 * 30/(24 * 30) = 0.004$. This extra cost is ~2% of the on-demand cost and 20% of the average spot instance costs. We account for these storage costs in our cost analysis.

Finally, Figure 11b shows the cost of using different spot instance types on EC2 as a function Flint's bid price. This figure demonstrates that in the current EC2 spot market, Flint's default policy of bidding the on-demand price results in the lowest cost. As the figure shows, there is a wide range of bid prices for each market, ranging from roughly half the on-demand price to 1.5× the on-demand price that yield the lowest cost. This behavior results from the spot prices in EC2 being "peaky" where they frequently spike from very low to very high, and then return to a low level. As a result, placing a bid price somewhere above the low steady state, but below the average peak, results in the same cost. Thus, unlike prior work that focuses on optimizing bidding strategies for EC2 spot instances, we find that in practice simply bidding the on-demand price is optimal, and that there is actually a wide range of bid prices that result in this optimal cost.

Result: *Flint executes applications at near the performance of on-demand servers (within 2-10%) but at a cost near that of spot servers, which is 90% less than using on-demand servers and 50-66% less than using existing managed services such as SpotFleets and Spark-EMR.*

6. Related Work

Our work builds upon a large amount of prior work on spot instances, as well as fault tolerance mechanisms.

Spot Markets. Since servers in the spot market are significantly cheaper than the equivalent on-demand servers, they are attractive for running delay-tolerant batch jobs [30, 16, 1]. Checkpointing is a common fault-tolerance mechanism for mitigating the impact of revocations on batch jobs in the spot market [32, 18, 35]. However, Flint employs fine-grained application-level checkpointing, rather than systems-level checkpointing, as in previous work. In addition, Flint focuses on distributed data-parallel jobs and not simple single-node batch jobs, as in recent work [30].

Prior work has also used spot instances for data-parallel tasks. For example, EC2's EMR service that we compare against [2] allows Hadoop and Spark jobs to run on spot instances, and may be combined with SpotFleets to define an automated policy to replace revoked spot instances. However, these services are application-agnostic and, as we show, by not considering the application characteristics they may make non-optimal decisions. In addition, prior work has explored running Hadoop jobs on spot instances [19, 11]. However, Hadoop lacks the built-in checkpointing and recomputation mechanisms that Flint leverages in Spark. Prior work has also explored running a distributed database on spot instance [9, 24]. This work addresses the problem of deciding serialization points for database operations, which differs from Flint's focus on defining checkpointing and server selection policies. Finally, Flint also supports interactive workloads. Prior work demonstrates that single-node interactive applications can be run on spot instances using continuous system-level checkpointing and nested virtualization [26]. However, Flint is a *distributed* data-parallel system for running BIDI workloads on transient servers.

Fault-tolerance Mechanisms. The performance effects of server failures has been well studied for Hadoop [14, 15]. Similarly, our work models the impact of server failures and revocations in Spark. Flint's intelligent checkpointing approach to minimize running time is based on the optimal approach for single-node batch jobs [12]. Other checkpointing mechanisms and policies have been developed for other types of applications. For example, Zorro uses checkpointing and other optimizations to recover from failures in distributed graph processing frameworks [23]. Similarly, Naiad also includes a policy for automatically checkpointing vertices and recovering from server failures [22]. Spark Streaming [37] incorporates automated periodic checkpointing of RDDs to enable real-time data processing, but does not take into account recomputation overhead and cluster volatility. These systems' policies may also benefit BIDI workloads running on transient servers, and are future work.

Bidding Policies. Spot market prices are determined by a second price auction and have been modeled in prior work [8]. Numerous bidding strategies for individual spot markets to optimize the cost/performance of batch jobs exist [40, 33, 38, 39, 29, 31]. However, as we show, a simple bidding strategy of bidding the on-demand price is optimal for Flint. By focusing on the checkpointing and server selection, Flint is applicable to transient servers that do not permit bidding, such as GCE's Preemptible Instances that offer transient servers at a fixed price.

7. Conclusion

The low price of transient servers is attractive for recent cluster-based in-memory data-parallel processing frameworks, since these frameworks need to cache large datasets in memory across a large number of servers. However, transient server revocations degrade the performance and increase the cost of these frameworks. In this paper, we design Flint, which includes intelligent, application-specific checkpointing and server selection policies to optimize the use of transient servers for data-parallel processing. In particular, Flint's policies support BIDI workloads that may be either batch or interactive. Our results show Flint enables a 90% cost saving compared to using on-demand instances and a slight decrease in performance of 2%.

Acknowledgements. We thank all the reviewers and our shepherd Joseph Gonzalez for their insightful comments, which improved the quality of this paper. This work is supported in part by NSF grants #1422245 and #1229059.

References

[1] PiCloud. http://www.multyvac.com, May 1st 2014.

[2] Amazon Elastic Map Reduce for Spark. `https://aws.amazon.com/elasticmapreduce/details/spark/`, June 2015.

[3] Livejournal Social Network Dataset. `https://snap.stanford.edu/data/soc-LiveJournal1.html`, June 2015.

[4] Transaction Processing Performance Council - Benchmark H. `http://www.tpc.org/tpch/`, June 2015.

[5] Hadoop Recovery. `https://twiki.grid.iu.edu/bin/view/Storage/HadoopRecovery`, March 2016.

[6] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling Spark in the real world: performance and usability. *VLDB*, 8(12):1840–1843, 2015.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark.

In *SIGMOD*, 2015.

[8] O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. In *CloudCom*, November 2011.

[9] C. Binnig, A. Salama, E. Zamanian, M. El-Hindi, S. Feil, and T. Ziegler. Spotgres-Parallel Data Analytics on Spot Instances. In *ICDEW*, 2015.

[10] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1), 1985.

[11] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *HotCloud*, June 2010.

[12] J. T. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3), 2006.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, December 2004.

[14] F. Dinu and T. Ng. Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *HPDC*, June 2012.

[15] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders. Failure Scenario as a Service (FSaaS) for Hadoop Clusters. In *Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, 2012.

[16] N. Jain, I. Menache, and O. Shamir. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *ICAC*, June 2014.

[17] B. Javadi, R. Thulasiram, and R. Buyya. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *UCC*, December 2011.

[18] S. Khatua and N. Mukherjee. Application-centric Resource Provisioning for Amazon EC2 Spot Instances. In *EuroPar*, August 2013.

[19] H. Liu. Cutting MapReduce Cost with Spot Market. In *HotCloud*, June 2011.

[20] D. Meisner, C. Sadler, L. Barroso, W. Weber, and T. Wenisch. Power Management for Online Data-Intensive Services. In *ISCA*, June 2011.

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine Learning in Apache Spark. *arXiv preprint arXiv:1505.06807*, 2015.

[22] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *SOSP*, October 2013.

[23] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost Reactive Failure Recovery in Distributed Graph Processing. In *SOCC*, August 2015.

[24] A. Salama, C. Binnig, T. Kraska, and E. Zamanian. Cost-based Fault-tolerance for Parallel Data Processing. In *SIGMOD*, 2015.

[25] P. Sharma, D. Irwin, and P. Shenoy. How Not to Bid the Cloud. University of Massachusetts Technical Report UM-CS-2016-002, 2016.

[26] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *EuroSys*, April 2015.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, May 2010.

[28] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. Ramakrishnan. Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers. *IEEE Internet Computing*, 18(4), July/August 2014.

[29] Y. Song, M. Zafer, and K. Lee. Optimal Bidding in Spot Instance Market. In *Infocom*, March 2012.

[30] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. SpotOn: A Batch Computing Service for the Spot Market. In *SOCC*, August 2015.

[31] S. Tang, J. Yuan, and X. Li. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *IEEE CLOUD*, June 2012.

[32] W. Voorsluys and R. Buyya. Reliable Provisioning of Spot Instances for Compute-Intensive Applications. In *AINA*, March 2012.

[33] S. Wee. Debunking Real-Time Pricing in Cloud Computing. In *CCGrid*, May 2011.

[34] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013.

[35] S. Yi, D. Kondo, and A. Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *IEEE CLOUD*, July 2010.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, April 2012.

[37] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[38] S. Zaman and D. Grosu. Efficient Bidding for Virtual Machine Instances in Clouds. In *IEEE CLOUD*, July 2011.

[39] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic Resource Allocation for Spot Markets in Clouds. In *Hot-ICE*, March 2011.

[40] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to Bid the Cloud. In *SIGCOMM*, August 2015.