Resource Deflation: A New Approach For Transient Resource Reclamation

Prateek Sharma
Indiana University
prateeks@iu.edu

Ahmed Ali-Eldin
University of Massachusetts Amherst
ahmeda@cs.umass.edu

Prashant Shenoy
University of Massachusetts Amherst
shenoy@cs.umass.edu

Abstract

Data centers and clouds are increasingly offering low-cost computational resources in the form of transient virtual machines. Whenever demand for computational resources exceeds their availability, transient resources can reclaimed by *preempting* the transient VMs. Conventionally, these transient VMs are used by low-priority applications that can tolerate the disruption caused by preemptions.

In this paper we propose an alternative approach for reclaiming resources, called *resource deflation*. Resource deflation allows applications to dynamically shrink (and expand) in response to resource pressure, instead of being preempted outright. Deflatable VMs allow applications to continue running even under resource pressure, and increase the utility of low-priority transient resources. Deflation uses a dynamic, multi-level *cascading* reclamation technique that allows applications, operating systems, and hypervisors to implement their own policies for handling resource pressure. For distributed data processing, machine learning, and deep neural network training, our multi-level approach reduces the performance degradation by up to 2× compared to existing preemption-based approaches. When deflatable VMs are deployed on a cluster, our policies allow up to 1.6× utilization without the risk of preemption.

CCS Concepts • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Virtual machines**; *Operating systems*;

1 Introduction

A transient computing resource, such as a server or a virtual machine, is one that can be unilaterally revoked by the cloud or data center provider for use elsewhere [70, 72, 86]. In enterprise data centers, low priority applications can be preempted after having their resources revoked, upon resource pressure from high priority applications [79]. In cloud context, all three major cloud providers, Amazon [1], Azure [6], and Google [3], offer *preemptible instances* that can be unilaterally revoked during periods of high server demand.

The primary benefit of transient computing is that it enables data center operators and cloud providers to significantly increase server utilization. Idling servers can be allocated to lower priority disruption-tolerant jobs or sold at a discount to price-sensitive customers. In both cases, the resource provider has the ability to reclaim these resources when there is increased demand from higher priority or higher paying applications. Preemptible cloud servers have become popular in recent years due to their discounted prices, which can be 7-10x cheaper than conventional non-revocable servers. A common use case is to run data-intensive processing tasks on hundreds of inexpensive preemptible servers to achieve significant cost savings.

Despite the many benefits, the preemptible nature of transient computing resources remains a key hurdle. From an application standpoint, server revocations are essentially fail-stop failures, leading to disruptions and performance degradation. Consequently, recent work has developed transiency-specific fault-tolerance mechanisms and policies to alleviate the effects of preemptions for different classes of applications such as data processing [67, 84], machine learning [40], batch jobs [74], and scientific computing [56]. In enterprise data centers, using transient resources to increase utilization and minimize the performance impact of preemptions remains an important problem [58, 79, 84, 90]. Even with these proposed solutions, the preemptible nature of transient resources presents a significant burden for many applications as they require changes to the application (legacy) code in many cases.

In this paper, we present *resource deflation* as a new approach for managing transient computing resources in data centers and cloud platforms. We argue that resource preemption is only one approach, and an extreme one, for reclaiming erstwhile surplus resources from low-priority applications. In resource deflation, transient computing resources allocated to an application can be dynamically reduced and reclaimed. Such reclamation can be done at the operating system, the hypervisor, or the application levels, albeit with different tradeoffs. By reclaiming partial resources, applications

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/https://doi.org/10.1145/3302424.3303945

can continue execution rather than being forcibly preempted. This expands the set of applications that can be hosted on lower priority transient resources. Specifically, applications without built-in fault-tolerance support, legacy applications that are not disruption-tolerant, and inelastic applications that require a fixed set of servers such as MPI and distributed machine learning—all of which are challenging to run on preemptible servers—can all seamlessly run on deflatable transient resources. In fact, resource deflation is a generalization of many other resource management techniques, including elastic scaling [71], resource overcommitment [76], application brownout techniques [52], and preemption [79].

Since fractional reclamation of resources hampers application performance, we design mechanisms and policies that allow applications and cluster managers to cooperatively reclaim resources to minimize performance degradation across applications. We demonstrate the efficacy of our approach for distributed data processing, distributed machine learning as well as other clustered applications. In doing so, our paper makes the following contributions:

1. We develop a multi-level resource reclamation technique called *cascade deflation*, that reclaims resources using reclamation mechanisms found in applications, operating systems, and hypervisors. Cascade deflation uses a judicious combination of reclamation mechanisms across different layers to minimize performance degradation. Compared to conventional techniques for VM resource reclamation, cascade deflation improves performance by up to 6×.
2. We show how the flexibility provided by cascade deflation allows applications to define their own policies for responding to resource pressure. We design application deflation policies for a range of applications including memcached, JVM, and Spark-driven distributed data processing and machine learning. Our deflation policy for Spark voluntarily relinquishes resources to mitigate resource contention and stragglers. This policy adjusts according to the elasticity of Spark programs to minimize the expected running time, and is able to reduce performance degradation by up to 2× compared to the current preemption-based resource pressure handling found in today's public clouds.
3. We design cluster management policies for deflation, and show that we can completely remove the risk of preemption even at cluster utilization levels as high as 1.6×.

2 Background and Overview

In this section we motivate the need for resource deflation as an alternative to preemption of transient resources. We also compare our approach to other related resource management mechansims and disuss its merits.

2.1 Transient Computing

Most data centers today are virtualized where applications run in either VMs or containers multiplexed on to physical machines. Since data center capacity is provisioned for peak demand, the average utilization tends to be low [26, 79]. Data center operators can increase the overall system utilization or maximize revenue, in case of the cloud, by offering unused server capacity transiently to low-priority applications or at a discounted cost.

Thus, the data center is assumed to host two classes of applications—high and low priority workloads. Low priority applications are scheduled whenever there is enough surplus server capacity in the data center; however, resources allocated to VMs of low priority applications are assumed to be transient. Some or all of these resources may be reclaimed at short notice when server demand from high priority applications starts increasing.

Current systems implement resource reclamation in the form of revocations, where server resources are reclaimed through VM preemptions. Cloud offerings such as Amazon Spot instances [1], Google Preemptible VMs [3], and Azure batch VMs [6] are examples of such low-cost but preemptible VMs. Enterprise data centers similarly preempt low-priority jobs when high priority jobs arrive [79, 84, 90].

Preemptions in public clouds can occur at different frequencies depending on the provider's preemption policies and the demand of the *non-revocable* resources (such as on-demand and reserved instances). For instance, Google's policies for preemptible VMs result in a Mean Time To Failure (MTTF) of less than 24 hours [3]. The preemption rate of an Amazon spot instance depends on the supply and demand of instances of that particular instance type, and their MTTFs can range from a few hours to a few days [8]. These preemptions impose additional deployment and performance overheads on applications. While always-on stateful services require special fault-tolerance middleware [70], even batch applications such as distributed data processing can suffer from a significant (2×) decrease in performance [67] due to preemption-induced recomputation.

2.2 Resource Reclamation in Clusters

The need for resource reclamation is common in cluster environments and arises in scenarios such as VM preemptions, preemption of low-priority jobs, and for system maintenance. Typically, reclamation through preemptions imposes a high performance impact on running applications and may also impact data center goodput. A recent study has shown that unsuccessful execution accounts for 65% of machine time in a Google cluster, and a non-trivial fraction of these failures are cased by job evictions [64]. Preemptions can result in downtimes, loss of state, and starvation of low-priority jobs [21]. Masking the impact of preemptions requires fault tolerance techniques such as periodic checkpointing [56, 67]. Implementing such methods requires application modification or the use of middleware systems [70].

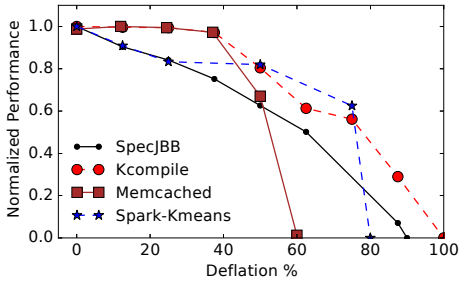


Figure 1. Many applications in virtualized environments can be deflated with only a small performance degradation.

In this paper, we consider partial reclamation of a resource such as a server or virtual machine, rather than "full" reclamation though preemptions. Our hypothesis is that partial reclamation, which we refer to as resource deflation, reduces the reclamation impact on applications when compared to preemption, and also enables a broader range of applications to run on transient resources. Resource reclamation *at a cluster level* has not received much attention, when compared to cluster-based resource allocation, which has been widely studied [34, 37]. We can view cluster-based resource reclamation as the inverse of cluster resource allocation. At an individual VM-level, partial resource reclamation can be implemented through *VM overcommitment*. Our work views VM overcommitment as a mechanism, available at the granularity of a single machine, for implementing cluster-wide reclamation policies. We also argue that VM overcommitment alone is not a sufficient mechanism for effective reclamation of resources from applications and present more general mechanisms for doing so.

2.3 Resource Overbooking and Elasticity

There are a number of cluster-wide resource management mechanisms related to resource reclamation. Resource overbooking is one approach where the cumulative peak resources needed by applications in the cluster exceed the total allocated resources [76, 77]; overbooking tends to be feasible since not all applications require their peak requested allocation at the same instant, allowing for statistical multiplexing of resources. VM overcommitment is an example of overbooking at the granularity of a single physical machine. Cluster-wide overbooking policies must consider careful placement and co-location of applications to minimize chances of overload due to concurrent peak demand. Effective overbooking typically requires knowledge of application workload characteristics and SLOs. In contrast, resource deflation does not require applications to specify explicit SLOs. Further, during periods of overload, overbooking techniques use these SLOs to degrade application performance. While deflation also degrades application performance during resource pressure, our methods are SLO agnostic in nature.

Elastic scaling [55] is another dynamic resource management technique where the capacity of a clustered application is varied dynamically based on workload fluctuations. This approach has been explored for web clusters [15, 33], Hadoop [32], Spark [28], and scientific workflows [44]. While shrinking an application in horizontal scaling is a form of reclamation, it typically requires knowledge of application SLOs and reduces allocation to *match* a lower demand [36, 48, 60]. In contrast, deflation under resource pressure can, and often will, reduce the allocation to a level far *below* the application's current demand. Thus, despite some similarities, neither overbooking nor elastic scaling are directly applicable for transient resource deflation.

Many applications deployed in clouds and data centers are deflation friendly and can tolerate significant amounts of deflation without the proportional decrease in performance. For instance, Figure 1 shows the performance degradation of four applications, namely, SPEC-JBB, Kernel-Compile, Memcached, and Spark, when the VMs are deflated by different amounts[1]. We see that the in many cases, even when 50% of all resources (CPU, memory, and I/O bandwidth) are reclaimed, the decrease in performance is less than 30%.

Furthermore, a majority of VMs are usually *overprovisioned* and have a surplus of free and unused resources, thereby giving deflation enough "headroom", and avoiding severe performance degradation. A recent resource usage study of VMs in Microsoft Azure cloud [26] shows that more than half the VMs had an average CPU utilization of less than 30%, and a 95%ile utilization of less than 70%. Thus in many cases, deflation can reclaim unused resources with minimal performance degradation.

Thus, deflation is not only feasible as an availability maintaining reclamation technique, but can potentially increase the overall goodput of virtualized clusters. In many cases, performance degradation, rather than outright termination and downtimes, may be acceptable even for many interactive applications except for the most mission-critical ones. Batch application may also prefer temporary deflation to preemption to avoid wasteful restarts. Furthermore, our work shows resource deflation is a feasible approach even for inelastic applications (e.g., ones that are incompatible with horizontal scaling) and enables a broad range of workloads to exploit transient resources in cluster environments.

3 Multi-level Resource Reclamation

Our approach for increasing the utilization and performance of computing resources entails running virtualized clusters that run low-priority deflatable VMs and high-priority non-deflatable VMs. Launching applications on such a cluster can create *resource pressure*, if the resources (CPU, memory, I/O) required exceed their availability. Our deflation-aware cluster manager (§5) places newly launched VMs according to deflation-aware bin-packing policies, and proportionally reclaims resources from all the deflatable VMs on a server. To reclaim resources from a VM, we employ a new technique

[1]Details about workloads and execution environment can be found in §6

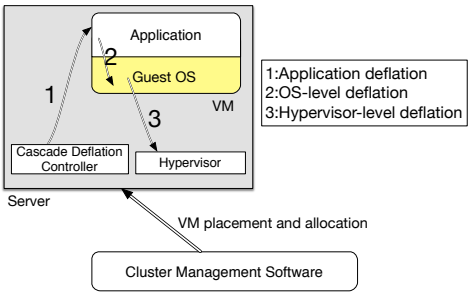


Figure 2. Overview of deflation-based cluster management

called *cascade deflation* that allows applications, operating systems, and hypervisors to work together to reclaim large amounts of resources, while minimizing performance degradation faced by the VM (Figure 2). Deflation allows applications the flexibility to define and implement their own policies to respond to resource pressure, and we describe application-level deflation policies for different applications in §4. Performance degradation due to dynamic resource reclamation is a primary concern with deflation, which we address through multi-level cascade deflation in this section.

3.1 Why multi-level reclamation?

Cluster resources in virtualized data centers and clouds are allocated and reclaimed by cluster managers based on resource availability and application priorities. VM-level cluster managers (like VMware vCenter [38] or OpenStack [12]) reclaim resources through hypervisor-level overcommitment techniques. On the other hand, bare-metal cluster managers such as Mesos [41], rely on applications to relinquish resources. Current cluster managers thus rely on a single reclamation technique (i.e., either hypervisor-level or application-level reclamation). This binary approach leads to two drawbacks: first, the reclamation functionality present in modern guest operating systems remains unused. Second, as we shall elaborate below, the reclamation mechanisms provided by different software layers expose different tradeoffs in their performance and safety, and relying on any single mechanism is suboptimal.

VM overcommitment can degrade performance. A common technique for fractional resource reclamation in virtualized clusters is to use hypervisor-level VM overcommitment. Since hypervisors virtualize resources and offer them to virtual machines, they can also *overcommit* these resources by multiplexing virtual resources onto physical ones. For example, CPU resources can be reclaimed by remapping a VM's vCPUs onto a smaller number of physical cores, and sharing the capacity of these cores using the hypervisor's built-in scheduling mechanism. Hypervisor-level VM overcommitment mechanisms treat the guest OS and the application as a "black box", and the VM itself has no knowledge of the deflation, which is done at the hypervisor level "outside" the VM. This allows resources to be reclaimed from all applications, including unmodified, inelastic ones.

However, black-box reclamation techniques can lead to significant performance degradation. Since the hypervisor has no knowledge of the relative importance of different resources to the application, it may overcommit the "wrong" resources. This problem frequently arises in memory overcommitment where the hypervisor (unknowingly because of the black box nature) swaps application pages to disk, instead of free pages. Similarly, overcommitting CPUs leads to complications like spin-lock preemption [29, 62, 75], wherein the multiplexing of vCPUs onto a smaller number of physical cores leads to excessive waiting for lock acquisition when the vCPUs holding locks get preempted by the hypervisor.

Reclaiming from higher layers is not always feasible. The performance concerns of black-box VM overcommitment can be alleviated by reclaiming resources from higher layers, i.e., the guest OS and the application. These higher layers have better knowledge of actual resource use, and can use control mechanisms to adjust resource usage in an application-aware manner. For instance, the OS can reduce the size of their disk caches instead of hypervisor-level swapping [16]. Similarly, distributed applications can reduce the number of parallel tasks to reduce resource contention, and mitigate lock holder preemptions mentioned previously. Thus, it is desirable and feasible to incorporate application support and cooperation in resource reclamation—something not currently used in VM-level cluster management.

However, reclaiming resources from applications alone is not sufficient in virtualized environments—resources freed by the application are *not* considered free by the hypervisor and cannot be directly reclaimed. Furthermore, relying on application support for reclamation may not always be feasible: the application may not have the control mechanisms, or may choose not to exercise them.

Thus, the current cluster management techniques that restrict resource reclamation to a single level are sub-optimal. We propose a multi-level reclamation policy that seeks to use the relative strengths of the different layers to deflate applications safely and gracefully, which we describe next.

3.2 Cascade Deflation

Our multi-level reclamation approach is called *cascade deflation* and determines how resources of different types (CPU, memory, I/O) are reclaimed across multiple software layers. Cascade deflation allows applications, operating systems, and hypervisors to define and use their own reclamation mechanisms and policies, as part of a common framework for reclaiming resources across multiple levels. The flow of resource reclamation across various layers is shown as pseudocode in Figure 3. When resources need to be reclaimed from a VM, cascade deflation starts by applying resource pressure at the highest layer (the application), and moves downwards to the OS and the hypervisor. The application may be able

```
#Reclamation target is vector of (CPU, Memory, Disk, Network)
 def Deflate_VM(target):
   app_r = application_self_deflate(target)
   unplug_r = hot_unplug(app_r, target)
   hypervisor_overcommit(unplug_r, target)
   return

def hot_unplug(app_r, target):
   unplug_target = max(app_r, get_system_free())
   #get_system_free() determines safely unpluggable resources
   unplug_target = min(unplug_target, target)
   unplug_r = try_unplug(unplug_target)
   #If resource is busy, unplug_r < unplug_target
    return unplug_r

def hypervisor_overcommit(unplug_r, target):
   if (unplug_r < target):
      #Unplugged resources released automatically
      VM_overcommit_mechanism(target − unplug_r)
```

Figure 3. Pseudo-code for cascade deflation

to free only some (or even none) of the resources, in which case the lower layers (the OS and hypervisor) are asked to reclaim the remaining amount of resources.

Thus, the reclamation *cascades* and moves down to the lower layers. If a layer fails to meet the reclamation target, then the lower layers pick up the slack. Having reclamation "fall-through" to the lower layers allows for safer deflation since applications and the OS can ignore excessive and unsafe reclamation requests. Thus, higher layers can free resources in a "best effort" manner in order to maximize their performance, while the lower layers seek to reclaim remaining resources to meet the reclamation target.

The intuition behind starting at the higher layers is that since applications and OSes have better knowledge of unused and underutilized resources, reliniquishing them reduces performance degradation. With casacde deflation, different amounts of resources can be reclaimed at different levels. Different layers can use their own reclamation mechanisms, as well as define policies on how to use those mechanisms. These policies are implemented by the different layers, and interact using the control-flow outlined in Figure 3. We present details on reclamation mechanisms and policies for the different layers below.

3.2.1 Application-level Reclamation

Mechanisms: Applications can partake in cascade deflation by relinquishing resources in response to deflation requests, by using their own resource control mechanisms and policies. Many distributed applications such as web server clusters, map-reduce style processing, key-value stores, etc., are *elastic*, and have mechanisms to adjust resource usage. For example, application-level caches (such as memcached, redis, etc.) can be shrunk using LRU-based object eviction. Similarly, web-clusters can reduce their CPU utilization by reducing the number of worker threads, and adjust the load-balancing rules accordingly (serve less traffic from deflated servers).

Application & Resource type	Reclamation Mechanisms
Memcached - memory	LRU object eviction to reduce memory footprint
JVM - memory	Trigger GC and reduce maximum heap size
Web servers - CPU	Reduce size of thread pool
Spark/Hadoop - All	Reduce number of tasks used

Table 1. Application-level deflation mechanisms for different application types

Distributed data and numerical processing applications can control their resource usage by adding and removing parallel-tasks and workers. Examples of deflation mechanisms for different application classes are presented in Table 1. Applications can use and combine these different mechanisms for reclaiming different resources (CPU, memory, I/O).

Policies: Application deflation policies determine how many resources (if any) to voluntarily relinquish. For inelastic applications that do not support dynamic reclamation mechanisms (synchronous MPI programs, single-VM legacy applications, etc.), the application deflation policy is to simply ignore the deflation request, and let the OS and hypervisor take care of the deflation. Elastic applications on the other hand can use application-level mechanisms to free resources and to *self-deflate*. Ofcourse, even elastic applications can choose to only partially deflate, or ignore the request entirely.

Since application self-deflation involves relinquishing resources, the degree of self-deflation is ultimately determined by safety and performance concerns. Applications can stop self-deflating if it risks loss of functionality or application failure. In some scenarios, even though the application may have the mechanisms for reclamation, doing so leads to excessive performance degradation. The degree of performance degradation depends on the application's performance model, and is determined by two main factors:

1. Short-term impact of deflation mechanism
2. Long-term impact of running on reduced resources

The short-term performance degradation is due to the overhead of the deflation mechanism itself. For example, some applications deflate by terminating tasks (such as in the case of Hadoop and Spark), which requires recomputing the lost program state, which increases the running time of the program. Similarly, the high garbage collection activity required to shrink the JVM heap size can temporarily degrade the performance of JVM-based applications. The long-term performance impact is due to the application running on reduced amount of resources, and depends on the application's utility curves (such as those shown in Figure 1) and the reclamation mechanism used.

Thus when determining the magnitude of self-deflation, application-level policies must account for both the short and long-term performance degradation, along with any safety constraints. Since the magnitude of deflation is fixed and decided by the cluster manager, application-level policies only need to *compare* the performance degradation for the

different deflation options, and thus utility curves are not required in our approach. Incorporating deflation mechanisms and policies requires minor application modifications, and we develop policies for different application types in §4. We also develop models for short and long term performance degradation for distributed data-parallel data processing and machine learning applications, and use them to design a dynamic running-time minimizing deflation policy for Spark.

Note that cascade deflation's multi-layer design is modular: it is not necessary for every layer to implement reclamation mechanisms for all resource types. If a reclamation mechanism is not implemented by a layer, the reclamation falls through to the lower layer. Thus, although it is beneficial to have application and OS level deflation, it is not necessary. We evaluate the performance of cascade deflation with and without application-level policies later in Section 6.

3.2.2 OS-level Reclamation

Mechanisms: Surplus resources in the VM, or those relinquished through application-level deflation, must still be reclaimed and released by the guest OS, since free resources inside a VM cannot be directly reclaimed by the hypervisor. To reclaim resources from the OS, we utilize resource hot-plug and hot-unplug mechanisms. Modern operating systems and hypervisors now support the ability to hot plug (and unplug) resources [4, 24], and these mechanisms can be used to explicitly change the resource allocation. Resources that are free or that have been recently relinquished by the application are "unplugged" from the VM, and returned to the hypervisor. Hot unplugging a resource (such as vCPUs) invokes the equivalent OS resource reclamation mechanisms. Hot-unplug also updates the resource allocation observed by the OS and applications (actual number of CPUs and memory available)—improving resource management at these layers.
Policies: For CPU reclamation, we unplug vCPUs until the CPU deflation target is reached. Hot plugging and unplugging is only possible at coarse granularity—it is not possible to unplug fractional CPUs. Therefore, the final amount of resources unplugged can be atmost $\lfloor \text{unplug\_target} \rfloor$. In case of memory, we use memory unplugging to explicitly reduce the memory seen by the guest OS. We don't hot unplug NICs and disks because it is generally unsafe.

In practice, hot unplugging of resources may fail or only succeed in partial reclamation, if the OS observes the resources to be busy. For instance, CPUs with tasks pinned on them are generally not safely unpluggable. Similarly, unplugging memory entails identifying blocks of free pages, and migrating pages to create a contiguous zone of pages that can be freed and unplugged. This operation may fail or result in a smaller amount of unplugged memory than the target. Our policy for hot-unplug based reclamation prioritizes safety and is best-effort: if an unplug operation fails due to busy resources, we seek to unplug a smaller target, and reclaim the rest with hypervisor-level reclamation.

3.2.3 Hypervisor level Reclamation

Mechanisms: Hypervisor level multiplexing of resources allows us to reclaim resources via traditional VM overcommitment mechanisms. We use CPU and I/O bandwidth throttling to reclaim CPU and I/O resources respectively [23]. Memory can be reclaimed through host-swapping or ballooning [80].
Policy: Cascade deflation invokes hypervisor deflation as the last step to reclaim remaining resources, and seeks to minimize its use because of its high performance degradation. The goal of hypervisor level reclamation is to simply reclaim all the resources to reach the deflation target. Resources freed through the OS-level reclamation are already freed and do not need reclamation. Reclaiming resources through hypervisor overcommitment is transparent to the application and the guest OS, and poses no direct risk to application availability, thus allowing us to reclaim large amounts of resources if required.

4 Application Deflation Policies

Casacde deflation allows applications, operating systems, and hypervisors to cooperate in the resource reclamation process and define and use their own reclamation mechanisms and policies. In this section, we will illustrate how elastic applications can develop and define deflation policies. We have developed deflation policies for multiple applications including memcached, JVM, and distributed data processing with Spark, to show that it is feasible to develop simple application deflation policies for a wide range of applications, with relatively modest implementation effort. For Spark applications, we present an online, running-time minimizing deflation policy that can serve as a case-study for distributed application deflation.
Memcached. Memcached is a popular user-space in-memory key-value store [5]. In conventional operation, the memcached server is started with a fixed, maximum cache size. Our application level policy for memcached dynamically adjusts the maximum cache size based on the memory availability inside the VM. When shrinking the cache size, the memcached object eviction algorithm (LRU) is invoked. Shrinking the cache size may result in a lower object hit-rate, but avoids paging in memory pages from the slow swap disk. This modification allows memcached to serve more traffic even when the memory is deflated to below the original cache size. Thus, because the long-term performance degradation with memory self-deflation is lower than VM-level deflation, our deflation policy for memcached uses application-level deflation for memory, and uses VM-level deflation for other resources. Our implementation is based on memcached v.1.3 and a previous dynamic memory-size version [42], and comprises of about 500 lines of modifications to the memcached server.
JVM. Application level deflation policies can also be implemented for garbage collected run-time environments such as Java Virtual Machines (JVM). In response to memory deflation, our application policy for JVM reduces the heap

size by triggering garbage collection. Reducing the heap size results in increased garbage collection overhead, but is nevertheless favorable to fetching pages from the swap disk. Prior work on JVM heap sizing have also explored this tradeoff [19, 85]. Our deflation-aware JVM allows the large class of JVM based applications to be made memory-deflation aware. Our deflation-policy for JVM-based applications uses application-level deflation for memory, and VM-level deflation for other resources. Of course, Java applications can specify their own application deflation policies to augment the JVM deflation policies. We use IBM's J9 JVM [7] that has the ability to change the maximum heap size during run-time. We set the max heap size to the actual physical memory availability to avoid swapping. We implement this in the application deflation agent using the JMX API in about 30 lines of Java code.

4.1 Spark

We now focus on distributed data processing and machine learning workloads, and use Spark as the representative data-parallel framework. Spark [88] is a general-purpose, widely used framework that is used for a wide range of applications like map-reduce style data processing, graph analytics [83], machine learning [57], deep learning [59, 82], relational data processing [17], interactive data mining, etc. The long and short-term performance degradation for Spark is thus highly variable and depends on the specific workload.

We design a *general* self-deflation policy for Spark that works across workload types, and is able to dynamically determine the extent of self-deflation required to minimize the running time of the workload. To do so, our policy uses simple models developed from first principles, and we therefore provide a brief discussion of Spark's runtime model next.

Spark Background. Spark uses Resilient Distributed Datasets (RDDs) [87] as the abstraction for data partitions, and RDDs are designed to be stored in a combination of memory and disk. Spark jobs are comprised of multiple data processing operations, and each operation (such as a `map`) operates on an RDD partition. Spark jobs can be viewed as a directed acyclic graph of RDD partition dependencies (Figure 4). If the output of a task is lost (due to task failure or termination), then Spark uses the RDD dependency graph to recursively *recompute* all missing RDD partitions. Of course, this recomputation may substantially increase the job running time.

A wide range of distributed data processing and compute-intensive applications have been built on top of Spark's RDD abstractions and data operations. These applications have different RDD dependency graph structures, demand for computing resources, and tolerance to deflation. While specific applications (such as parallel K-means, a popular machine learning workload) can define their policies for cascade deflation, Spark's common runtime environment presents an opportunity for a common, *general* application deflation policy that can work across multiple applications.

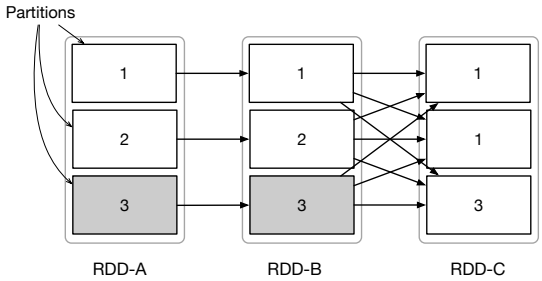


Figure 4. Spark jobs create a directed acyclic graph of RDD partitions. Partitions are computed by tasks that run on different VMs. Loss of a partition (RDD-B's 3rd partition) requires recomputing its dependencies.

Another class of applications that the Spark framework supports is distributed deep neural network training and inference. A popular technique for parallelizing these applications is to use data-parallel architectures such as parameter servers [53], and optimize the network model iteratively. During training, data is partitioned across workers, and the network model parameters are updated in a distributed fashion using optimization techniques such as stochastic gradient descent. At the end of each iteration, workers share and update model parameters. However, these updates are typically *synchronous* in nature, to ensure that all workers start with the same model state before each iteration [25]. Since a large portion of the training job is synchronous, the job is inelastic and cannot scale easily. However, the combination of cascade deflation and the model-driven Spark deflation policy allows us deflate deep-learning applications (along with other Spark applications), and run them on low-cost transient resources.

4.1.1 Cascade Deflation Policy For Spark

Our Spark deflation policy tries to minimize the performance impact of deflation. The basic mechanism we use for application-level deflation is terminating Spark tasks. Terminating tasks allows Spark to reduce its degree of parallelism, and the freed resources are returned to the hypervisor via cascade deflation. However, terminating tasks can trigger expensive recomputation of dependent tasks and results in high short-term performance degradation. With cascade deflation, if the application does not relinquish resources, then resources have to be reclaimed by the OS and hypervisor. We refer to the combination of OS and hypervisor level deflation as "VM-level deflation" for ease of exposition. However, with VM-level deflation, tasks on deflated VMs can turn into stragglers and result in a higher long-term impact.

Since different deflation mechanisms impose different tradeoffs for Spark application performance, we design a cascade deflation policy for Spark that is able to choose the "right" deflation mechanism. Our cascade deflation policy estimates the running time with application-level and VM-level deflation, and chooses the mechanism that minimizes the expected running time. Based on the application's recent

execution history, we use simple performance models to estimate T_{VM}, the running time with VM-level deflation, and T_{self}, the running time with self-deflation.

Our deflation policy for Spark is general-purpose and *online*, and does not require offline profiling or pilot jobs. When VMs of a Spark application are deflated, they send their reclamation targets to the Spark master, that executes the deflation policy and determines if application-level deflation would be desirable. We do not deflate the Spark master, and run it on a high-priority VM. Since multiple VMs may be deflated simultaneously, the Spark master collects all the deflation requests into the deflation vector $\mathbf{d}$, with d_i representing the deflation desired on VM-i. We model the slowdown of Spark applications using a simple performance model for VM and self deflation below:

Running Time With VM Overcommitment. When VM overcommitment is used, the deflated VMs will execute tasks slower than the non-deflated VMs, leading to resource contention on the deflated VMs and stragglers. Due to stragglers and BSP execution model [22], the running-time will be determined by the VM deflated the most. If the deflation occurs when c fraction of the job has finished, then the reminder of the job will be slowed down by a factor of $(1-c)/(1-\max\{d\})$. Thus we assume that the job will be slowed down linearly due to reduced resource availability. Furthermore, we model Spark jobs as a sequence of Bulk Synchronous Parallel (BSP) stages, and thus deflating even a single VM can result in a large slowdown because tasks on other non-deflated VMs need to "wait" for the slower tasks on the deflated VM. If T is the running time of the job without deflation, then the the total running time with VM-level deflation is:

$$T_{VM} = T \cdot \left[c + \frac{1-c}{1-\max\{\mathbf{d}\}} \right] \tag{1}$$

Running Time With Self-deflation. Spark self-deflation involves terminating tasks/executors. This controls the degree of parallelism, and can also mitigate stragglers, since it removes the imbalance caused by deflation of a subset of VMs. However, recursively recomputing output of terminated tasks increases the short-term cost of deflation. In general, the recomputation cost can be expressed as:

$$\text{Recomputation cost} = rcT \tag{2}$$

Here, r determines the fraction of the job that will be recomputed, and depends on the nature of the RDD DAG, whether the dependencies are already cached and do not require recomputation, and other application-specific factors. In the worst-case, $r = 1$, and the entire job so far has be recomputed.

Since the Spark master has knowledge of the DAG, the time required for various tasks, and the cached state of various RDDs, it can determine the recomputation cost by recursively tracing the DAG, and adding the recomputation cost for the various dependent tasks. However, a simple heuristic can also be used instead: $r = \frac{\text{Synchronous execution time}}{\text{Total running rime}}$

The intuition behind this heuristic is that in general, a larger number of (synchronous) shuffle stages implies a higher recomputation cost. Shuffle operations have a larger number of dependencies, and hence higher likelihood of missing dependencies which have to be recomputed. Spark applications thus have a choice of different recomputation cost estimates. They can either compute accurate estimates using the knowledge of the DAG and other execution characteristics; or use the worst-case estimate ($r = 1$); or use the synchronous execution time heuristic discussed above.

We use the synchronous execution time heuristic because it represents a middle ground between the application oblivious worst-case estimate, and the application-specific DAG-based estimate, and is general enough to work across a range of Spark applications. Our policy also determines if a shuffle operation is scheduled in the immediate future by looking at the RDD DAG, and accounts for that by setting $r = 1$, since the terminated tasks will not have their RDDs cached, and will require recomputation.

Note that the degree of slowdown with self-deflation and VM overcommitment is different. The Spark task scheduler scales back the number of tasks on deflated VMs, allowing for an even load distribution, and the degree of slowdown is the *average* of the deflation for each VM ($\bar{\mathbf{d}}$). In contrast, VM overcommitment faces a larger slowdown ($\max\{\mathbf{d}\}$) due to load imbalance and stragglers. The total running time with self-deflation is thus :

$$T_{self} = T \cdot \left[c + \frac{rc + 1 - c}{1 - \bar{\mathbf{d}}} \right] \tag{3}$$

Our policy compares T_{VM}, T_{self}, and selects whichever yields the lower running time estimate. Since T, the undeflated running time, is a common factor, it is not required. The job-progress (c) is estimated as the fraction of stages completed. Since self-deflation imposes the risk of high recomputation cost, our policy tends to use VM overcommitment for jobs that are close to completion (c close to 1).

Spark Policy Implementation: We have implemented the Spark policy for self-deflation described above as part of the Spark master in Spark v2.3.1. For self-deflation, we kill running tasks and blacklist their executors so that additional tasks are not launched on deflated VMs. We use the Spark HTTP API and application logs to get all relevant metrics for the self-deflation policy: job completion statistics (c), whether a shuffle stage is pending, and the shuffle-intensity of the job (α). The self-deflation policy is implemented as a HTTP service started by the Spark master (about 500 lines of Scala), and listens to the deflation requests from the hypervisor's local deflation controller. We also determine the number of tasks to kill based on the deflation requests and the size of tasks. Spark workers relay the deflation requests to the Spark master, which then executes the policy, and returns the amount of relinquished resources on each worker.

5 Implementing Deflation-based Cluster Management

Our deflation framework allows users to deploy applications using a combination of non-deflatable, non-preemptible high priority VMs and deflatable low-priority VMs. Our system is comprised of two main components. First, a centralized cluster manager allocates and reclaims resources through VM placement and proportional deflation policies at a cluster level. Second, each server runs a local deflation controller (Figure 2), which keeps track of resource allocation and availability, and implements proportional cascade VM deflation at a single machine level. We have implemented both the centralized cluster manager and the local-controllers in about 4,000 lines of Python. The two components communicate with each other via a REST API.

The implementation complexity of our prototype is comparable to that of other preemption-mitigation systems. As a point of comparison, ExoSphere's cluster management and application fault-tolerance policies are over 5,000 lines of code [68], inspite of being based on an existing cluster manager (Mesos). We now describe the design and implementation of our deflation-based cluster manager.

Bin-packing based VM placement. When a new application is launched on the cluster, its high and low priority VMs are individually placed onto the cluster (physical) servers. Servers host a mix of high and low priority VMs. Our VM-placement policies determine which physical server to place each VM on, by using a multi-dimensional bin-packing approach, where the multiple dimensions are the CPU, memory, network, and disk resources. Bin-packing VMs onto servers is the standard technique for VM placement [63], and it takes into account the free/available resources on each server. In our case, since low-priority VMs can be deflated to free-up server resources, we consider the sum of free *and* the deflatable resources, when placing VMs.

We use the notion of "fitness" to place a VM onto a server, which in our case is the cosine similarity between the VM's resource demand vector and the server's resource-availability vector: $\text{fitness}(\mathbf{D}, \mathbf{A_j}) = \frac{\mathbf{A_j} \cdot \mathbf{D}}{|\mathbf{A_j}||\mathbf{D}|}$. Since resources can be reclaimed from deflatable VMs already running on a server, the availability vector is given by:

$$A_j = \text{Free}_j + \text{Deflatable}_j \tag{4}$$

Deflatable_j is the total amount of resources (across all VMs) that can still be reclaimed by deflation. Using the above formulation, our cluster manager implements best-fit, first-fit, and a 2-choices policy that randomly selects two servers and places the VM on one with higher fitness (larger free+deflatable resources).

How much to deflate VMs by? In order to run a VM on a server, resources may need to be reclaimed, if there are insufficient free resources. Cluster-level policies determine how much to deflate each VM by—VMs are actually deflated using cascade deflation. We implement a simple proportional cascade deflation policy that deflates all low-priority VMs by an amount proportional to their size. For example, suppose a new high-priority VM of size R is placed on a server with no free resources available, and n deflatable VMs of size M_i. Then, the VMs are assigned deflation targets of x_i, such that $\sum x_i = R$, and $x_i = (M_i - m_i) - \alpha(M_i - m_i)$. Here, m_i denotes the minimum size of the VM, beyond which deflation is not feasible/safe, and the VM is preempted instead. Minimum sizes are optional in our framework and default to 0, but allow applications to control their deflatability and preemptions, and can be set based on application SLOs. Our cluster policies thus use bin-packing to globally balance the load across the cluster, and proportional deflation to reclaim resources within a single server.

Implementation details. Once the deflation amounts have been determined, we use cascade deflation to deflate individual VMs. The cascade deflation is orchestrated by the per-server local deflation controller, which performs the reclamation for each VM on a server concurrently. Our prototype deflation controller is implemented for the KVM hypervisor [51], and uses the libvirt API [10] for managing VM lifecycles, and for hypervisor and guest-OS level deflation.

For application-level deflation, applications use a deflation agent with a REST endpoint. The deflation agents listen to deflation requests (in the form of deflation vectors), invoke the application-level mechanisms, and respond with the amount of resources volunarily relinquished. The local controller then invokes OS and hypervisor level reclamation, if necessary.

For hot-plugging (and unplugging) of CPU and memory required for OS-level deflation, we rely on QEMU's agent-based hotplug. A QEMU hotplug agent runs inside the VMs as a user-space process, and listens for hotplug commands from the local deflation controller. The hotplug commands are passed to the guest OS kernel via this agent. This allows the hotplug to be "virtualization friendly". Unlike physical resource hotplug where unplug is a result of a fail-stop failure, the agent-based approach allows unplug operations to be executed in a best-effort manner by the guest OS kernel. This increases the safety of the unplugging operations. For example, if the guest kernel cannot safely unplug the requested amount of memory, the hot unplug operation is allowed to return unfinished. In this case, the memory reclaimed through hot plug will be lower, but the safety of the operation is increased.

For hypervisor-level deflation, we run KVM VMs inside Linux cgroups containers [11], which provide a unified interface for reclaiming resources, and also help limit the performance interference between VMs by limiting their resource usage. For CPU multiplexing, we adjust the cpu shares of the VM. For memory multiplexing, we limit the VM's physical memory usage by limiting the memory usage of the cgroup (mem.limit_in_bytes). Large memory reclamation

Workload	Description
Memcached	In-memory key-val store. YCSB and Redis `memtier_benchmark` for load generation
Kcompile	Linux kernel compile
SpecJBB	SpecJBB 2015 benchmark in "fixed IR" mode. IBM J9 JVM
ALS	Spark mllib Alternating Least Squares on 100GB dataset
K-means	Spark mllib dense K-means clustering with 50GB dataset
CNN	Resnet convolutional neural network with Spark-BigDL on Cifar-10 dataset. BatchSize=720, depth=20, classes=10
RNN	Recurrent neural network with Spark-BigDL on Shakespeare Texts corpus

Table 2. Workloads used for experimental evaluation

operations can often fail, and we use a control loop for incremental, gradual reclamation. Similarly, we throttle the disk and network bandwidth using the appropriate libvirt APIs. Deflation operations have a deadline that is primarily determined by the amount of memory reclamation. If a deflation operation times out, we proceed to the next level in cascade deflation. In some cases, partial deflation may be sufficient to meet the new resource demands. In the worst case, VMs that are farthest from their deflation target are preempted.

Finally, the cluster manager monitors VM lifecycle events (startup, shutdown, termination) to maintain consistent allocation and availability information of all servers. If some resources become available, then it reinflates VMs. Just as with deflation, we reinflate VMs proportionally. Cascade deflation can be used "in reverse" to reinflate individual VMs: it first increases the hypervisor-level allocation, then adds resources to the OS, and finally informs the application's deflation agent of the additional resource availability.

6 Experimental Evaluation

We now examine the behavior of our deflation framework using testbed experiments and a range of application workloads. Our evaluation is guided by the following questions:

1. How does cascade deflation compare with other reclamation techniques?
2. How does deflation affect the performance of distributed data processing and machine learning workloads?
3. What is the impact on cluster management metrics such as throughput, utilization, and overcommitment?

Environment and Workloads. We use the deflation-based cluster management system described previously in §5 to perform our empirical evaluation. We run applications in KVM VMs running on Ubuntu 16.04.3 (x86-64). The cluster servers are equipped with Intel Xeon E5-2670 v3 CPUs (2.3 Ghz). Unless otherwise stated, we run VMs with 4 vCPUs and 16 GB of memory. Cluster applications such as Spark workloads and Memcached are run on a cluster size of 9 VMs, unless otherwise stated.

We evaluate the performance of deflation techniques over a wide spectrum of workloads listed in Table 2. All our Spark workloads use Spark v2.3.1, and are run with a cluster of 8 worker VMs and 1 master VM. For the neural network training workloads (CNN and RNN), we use Intel's BigDL [82] library benchmarks [9] with default network parameters. Neural network training is an example of a *synchronous* and inelastic workload, i.e., the loss of any VM results in the entire application stalling. While *asynchronous* training is also a popular mode of operation, its effectiveness is reduced in heterogeneous cloud environments [46], and hence we use the synchronous mode of operation. Using Spark for neural network training provides us a uniform platform for implementing and evaluating our deflation policies. Evaluation of cascade deflation for specialized frameworks such as TensorFlow [13] is part of our future work.

We are primarily interested in the overhead of deflation, and all results are normalized to the "no deflation" case.

6.1 Application Performance with Deflation

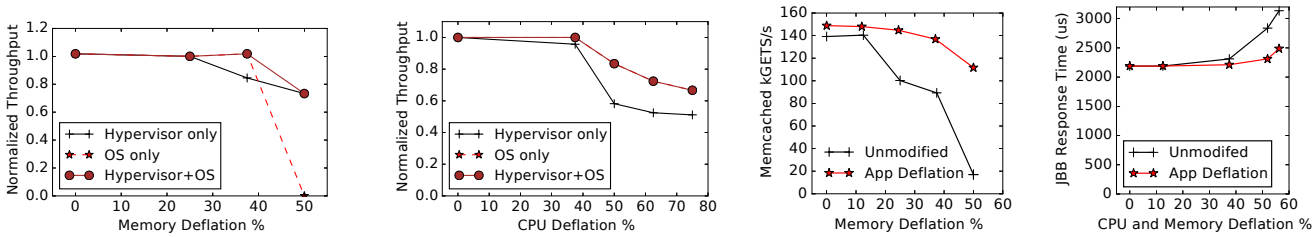
We begin by analyzing the performance impact of different fractional reclamation approaches outlined in §3. We are interested in evaluating the effectiveness of cascade deflation and comparing it with single-level reclamation approaches.

No Application Deflation: We first look at the performance of unmodified applications (without application-deflation support), to examine the behavior of hypervisor-level and OS-level deflation. The throughput of the memory-intensive memcached workload at different memory deflation levels is shown in Figure 5a, where we report successful GET requests (cache hits) per second. At 50% deflation, memcached throughput decreases by around 20% with hypervisor-level deflation (host-memory swapping in this case). While OS-level memory hot-unplug achieves superior performance up to 40% deflation, memcached runs out of memory and is terminated at higher deflation levels, making it impractical to rely on OS-level deflation alone. The combination of hypervisor and OS level techniques used with cascade deflation is able to "switch over" from OS to hypervisor level deflation to yield superior performance over a range of deflation levels.

Similarly, Figure 5b shows the performance of the CPU intensive kernel-compile benchmark at different CPU deflation levels. The performance with hypervisor-only deflation is inferior compared to OS-level techniques (vCPU hot-unplug) by up to 22%, likely due to lock-holder preemption [29]. Combining hypervisor and OS level deflation (which cascade deflation does) allows us to deflate the application by 75%, with only 30% decrease in performance.

With Application Deflation: We now evaluate the performance effects of the application self-deflation policies, which engage all three layers of cascade deflation. We compare against VM-level deflation (Hypervisor+OS), which does not use application deflation.

Figure 5c shows the performance of a memcached workload at different memory deflation levels. Our memcached application deflation policy evicts least recently used objects



(a) Memcached memory deflation **(b)** Kernel-compile CPU deflation **(c)** Memcached **(d)** JVM (SpecJBB)

Figure 5. Hybrid deflation improves performance for both CPU and memory deflation.

to reduce memory usage, and this results in a 6× improvement in throughput at 50% deflation. At high deflation levels, the unmodified version has to read some objects from swap, which is a slow operation bound by the disk-speed. Additionally, these slow GET requests (that hit swap), increase system load and decrease the overall throughput of the application. The deflation-aware memcached avoids this by sizing the cache to fit in the available memory, and sees a *higher* number of cache misses because it has evicted items that wouldn't fit in the memory available. But by doing so, it avoids swapping and obtains a much higher throughput, yielding a higher effective cache hit rate in terms of GETS/s.

Similarly, Figure 5d evaluates the performance of the SpecJBB workload across different CPU and memory deflation levels (both resources deflated by the same fraction). Our deflation policy for JVM-based applications minimizes swapping by reducing the heap size by triggering garbage collection. At higher deflation levels, this policy results in a 20% improvement in the response time.

Result: *Using OS-level reclamation is insufficient since it can lead to application failures. Using both hypervisor and OS level deflation can improve performance of unmodified applications by 20%. Cascade deflation with simple application deflation policies can improve performance by upto* 6×.

6.2 Distributed Processing On Deflatable VMs

While the previous subsection focused on evaluating cascade deflation for single-VM applications, we now turn our attention to the performance of distributed data processing and machine learning workloads. We evaluate the Spark deflation policy developed in §4.1 and compare it against alternative reclamation approaches. Our Spark deflation policy chooses either application-level self deflation, or VM-level deflation, and we compare these two approaches.

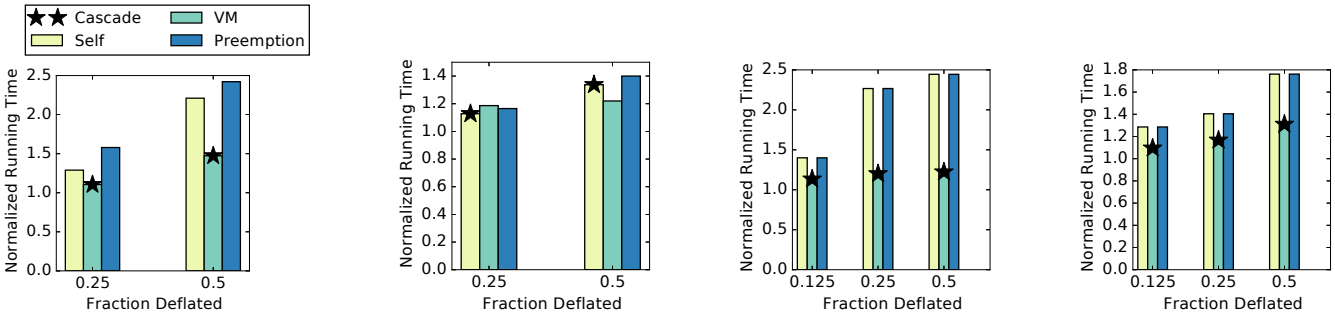
We deflate Spark applications by deflating all its VMs (CPU, memory, and I/O), and deflate the applications roughly 50% into their execution, and thus the applications run with 100% resources in the first half and then with reduced resources for the reminder of their execution. Figure 6 shows the normalized running time (relative to no deflation) for four different Spark workloads. We note that the deflation performance of Spark depends on the characteristics of the RDD computation graph, and hence each of the workloads in Figure 6 exhibit different performance characteristics.

The performance of the ALS workload (Figure 6a) scales fairly linearly with VM-level deflation—the running time increases to 1.5× at 50% deflation. However, using self-deflation increases the running time to 2.2×. Self-deflation for Spark involves terminating tasks, which requires recursive recomputation. The RDD recomputation graph for ALS is shuffle-heavy and involves significant amount of recomputation. Based on our Spark deflation models developed earlier in §4.1, our deflation policy (denoted by "Cascade" in Figure 6) chooses VM-level deflation for ALS, since it does not involve terminating tasks. With VM preemption, the running times increase to 2.5× at 50% deflation, again due to the recomputation costs. However, we note that the recomputation costs (and hence the running times) for self-deflation are lower by about 15% compared to preemption, because self-deflation allows recovering some RDD partitions from Spark's RDD cache instead of recomputing from input data sources.

K-means (Figure 6b) has lower recomputation costs, and hence lower degradation due to deflation. Self-deflation is preferred by our policy, resulting in an 18% and 38% increase in running time at 25% and 50% deflation respectively.

The performance characteristics of the deep neural network training workloads shown in Figures 6c and 6d differ significantly from conventional Spark workloads (like ALS and K-means). As described in §4.1, synchronous operations are used in neural network training, and loss of even a single task requires *restarting* the entire job, from a previous model checkpoint if available. Thus, self-deflation and preemption, which kill tasks, result in significantly higher running times compared to the VM-level deflation technique which does not require restarts. For CNN training (Figure 6c), the increase in running time even at 50% deflation is only 20% with VM-level deflation. Compared to preemption, the current transiency mechanism used by cloud providers, deflation results in a 2× decrease in running time.

Similarly, the RNN workload (Figure 6d) sees its running time increase by 25% at 50% deflation. Compared to preemption, the running time is lower by 25%.



(a) ALS **(b)** K-means **(c)** CNN Training **(d)** RNN Training

Figure 6. Performance of various Spark workloads with different deflation techniques and deflation amounts. Cascade deflation uses the Spark deflation policy developed in §4.1 to select the "best" deflation mechanism to minimize overhead.

In addition to the characteristics of the RDD graph, application performance also depends on *when* it was deflated (Equation 1). Figure 7a shows the running time of ALS when the application is deflated at different points in its execution. Early in the execution, self-deflation achieves better performance since the recomputation required is smaller, and a cross-over point is reached at around 30% deflation. Since deflating reduces resource allocation, the overhead trends downwards for both techniques since a smaller fraction of the job needs to be run with reduced resources.

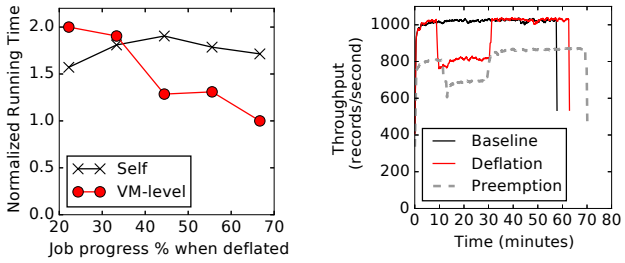
With deflation, long-running applications such as neural network training can respond gracefully to resource pressure. Figure 7b shows the throughput of the CNN training workload over time, when the application faces 50% deflation for 30 minutes. During the period of deflated execution, the application continues to run, albeit with throughput reduced by 20%. With preemption, periodic checkpointing is required, which reduces the throughput by 20% even during normal execution. Preemptions require restarting the entire job from the latest checkpoint, which further reduces the throughput. Thus compared to preemption, deflation improves CNN training throughput by 20%, even with transient resource pressure and periodic checkpointing.

Result: *For Spark applications, performance overhead of deflation is up to 2× lower than preemption. Deflation enables inelastic applications such as neural network training to gracefully respond to transient resource reclamation.*

6.3 Cluster-wide Behavior

So far we have seen the effect of deflation on individual applications. We now look at how deflation impacts the global behavior of virtualized clusters.

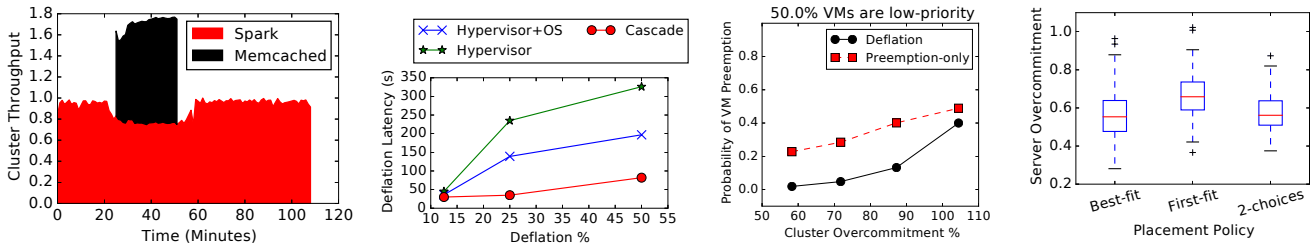
Throughput: We have already seen that the performance degradation with deflation is not always proportional to the amount of resources reclaimed. This facilitates overcommitting cluster resources while at the same time increasing the overall cluster throughput. We run Spark (CNN workload) on low-priority deflatable VMs, and introduce resource pressure by launching high-priority memcached VMs on the cluster, causing the Spark VMs to be deflated. When the memcached VMs start running, the Spark VMs are deflated by 50%, and the cluster is effectively overcommitted by 50%. Figure 8a shows the overall throughput of two applications : Spark (CNN Training) and a memcached cluster. While the Spark throughput decreases by around 20%, the total cluster throughput peaks at 1.8 when both memcached and Spark are running. Thus, deflation allows cluster managers to overcommit resources *and* significantly increase total throughput (or equivalently, revenue, in the case of cloud operators).



(a) Performance of ALS when deflated at different points. **(b)** CNN throughput with transient resource pressure (10–40 minutes).

Figure 7. Spark performance overheads.

Latency: An additional metric important for cluster management is the latency of resource allocation, which includes the time required to find free resources and reclaim resources if necessary. Since deflation performs gradual resource reclamation before new VMs can run, it increases the allocation latency. In general, the deflation speed is dominated by deflating memory, since it often entails saving memory state to stable storage (such as swap). We look at the worst-case deflation latency by deflating a single giant VM with 48 vCPUs and 100GB memory by 50% in Figure 8b. With full cascade deflation (including application deflation), the deflation latency even with 50% deflation is under 100 seconds. Even with application deflation, the freed resources still need to be reclaimed by the OS and the hypervisor, which contributes to the deflation latency. At high deflation levels, the deflation latency without application deflation is up to 2×—3× higher. Note that deflation is concurrent across VMs and the single



(a) Cluster throughput **(b)** Deflation latency **(c)** Preemption probability **(d)** Cluster imabalance

Figure 8. Cluster-wide properties with deflation

large VM deflation is the worst-case scenario, and this worst-case latency is comparable to the grace period required with preemptions (2 minutes for Amazon EC2 spot instances).

Preemption: While deflation permits overcommitment, VMs can only be deflated up to their minimum resource levels. In cases of extreme overcommitment, our cluster manager preempts VMs when they are deflated below their minimums. We evaluate the probability of VM preemption with deflation on a large 100 node cluster using a simulation approach. Our cluster simulator implements the proportional deflation and VM placement policies described in §5. We use the Eucalyptus cloud traces [2] to obtain VM arrivals, lifetimes, and VM sizes. We assign some fraction of VMs as low-priority VMs that are either deflated or preempted. We use empirically determined minimum levels for Spark, memcached, and SpecJBB application VMs, and determine the VM preemption probability when the cluster is overcommitted to different degrees. Figure 8c shows that the preemption probability with deflation is negligible even at 60% overcommitment, or 1.6× cluster utilization. Cluster overcommitment levels as high as 60% are rare even in agressive cluster operation [79], and overcommitment levels tend to be around 20%. Thus, preemption is a rare event with deflation, and it is not necessary for applications to implement preemption-mitigation.

VM Placement: Deflation uses modified VM placement techniques that we develop in §5. Careful VM placement is important for cluster load balancing and for increasing overcommitment. We again use the trace-driven simulator to evaluate the *server* level overcommitment using different deflation-aware bin-packing policies. With deflation, our goal is to maximize the overcommitment of servers, while at the same time reducing the preemptions. Figure 8d shows that all placement policies yield similar levels of server overcommitment. The differences in the placement algorithms are masked by the use of deflatable VMs, since suboptimal online VM placement can be "fixed" by deflation.

Result: *Deflation permits high cluster overcommitment, while yielding high cluster throughputs (up to* 1.8×*), and low preemption probabilities. Cascade deflation reduces reclamation (and hence allocation) latency by* 2×–3×.

7 Related Work

Our proposed deflation system draws upon many related techniques and systems.

Systems for running applications on transient servers use a combination of fault tolerance [40, 56, 67, 86] and resource allocation policies [68, 70, 74] to ameliorate preemptions. Deflation is designed to avoid the performance, development, and deployment costs associated with preemption.

Resource overcommitment mechanisms have been well studied and optimized to allow for more efficient packing for VMs onto physical servers. Memory overcommitment mechanisms such as ballooning have received significant attention [16, 69, 80], but ballooning generally yields inferior performance to hotplug due to memory fragmentation [47, 54]. The use of hotplug has also been proposed for reducing energy consumption [89]. Our use of CPU hotplugging is partly motivated by mitigating lock-holder preemption problems in overcommitted vCPUs [29, 62]. Application-level ballooning [65] reclaims memory from database and JVM applications—cascade deflation generalizes this to multiple resource types, and does not require guest OS modifications.

Application deflation. Improving elasticity for popular applications is an increasingly common pursuit. Dynamic heap sizing [19, 20, 85] is a popular technique for improving memory-elasticity of applications. The memory elasticity of data-parallel applications is enhanced in [30, 45]. Applications can also respond to deflation by serving less optional content [52], by reducing the quality of their results [73], or by giving them incentives for improved efficiency [18, 66]. Cascade deflation can make use of these elasticity control mechanisms. Incorporating elasticity into neural network training [40] presents multiple challenges due to the synchronous and inelastic nature of most deep learning frameworks. However unlike prior work, our approach does not require extensive application-level modifications.

Cluster resource management. Improving the utilization and performance of large computing clusters is a long standing challenge, and is typically tackled via resource allocation [28, 34, 37] and scheduling [35, 41]. However, many of the optimizations for fast job and task scheduling [27, 49, 50] are not relevant for VMs which are longer running, have

strict resource reservation requirements, no notion of completion times, and do not expose application-level performance metrics. Dynamic VM resource allocation [38, 39, 61] and bin-packing based VM placement [78] are common techniques for increasing the efficiency of virtualized clusters. Our work extends these ideas to multiple resource classes (deflatable and non-deflatable), and adds application-level deflation into a unified cascade deflation framework. Incorporating predictive resource management [26] for deflatable VMs is part of our future work.

8 Discussion and Future Work

Deflation is a departure from preemption, and can affect the execution and deployment of VMs in the cloud. In this section, we discuss how deflation can fit into cloud ecosystems, and potential impact on cloud providers and applications.

Impact on Cloud Providers: While preemption is relatively straightforward to implement, deflation introduces additional policy decisions for cloud and data center operators, which can further increase the complexity of resource management, both at a server and at a cluster level.

Further research is required on how our policies for placement and VM deflation interact with existing resource allocation and pricing policies. To this end, we deliberately developed relatively simple policies in Section 5, so that they can be composed with other existing cluster management policies for admission control, SLO-aware allocation, VM placement, global cluster-wide optimization, pricing, etc.

As a possible pathway to adoption, running internal and first-party workloads (which make up a non-negligible portion (20%) of cloud workloads [26]), can allow providers to test and refine deflation policies before they are rolled out to third party VMs.

On a per-machine level, deflation introduces additional complexity to VM management, especially due to the dynamic resource allocation. We argue that the additional complexity would be at-par with burstable VMs [81] that are already being offered by cloud providers. While deflation also adjusts memory allocation (in addition to dynamic CPU and I/O allocation that even burstable VMs offer), the key difference is that deflation is only performed under resource pressure, and not over the entire lifetime of the VM as is the case with burstable VMs.

Finally, while VM overcommitment mechanisms have long been studied and implemented in the context of smaller, private clouds and enterprise clusters [38], more research is required on their robustness at cloud-scale. For instance, while our system runs all VMs inside cgroups to limit performance interference, the large-scale implications of co-locating deflatable and non-deflatable VMs remain to be explored.

Pricing: Given their similar roles in clearing surplus cloud resources, we envision that deflatable VMs will be offered at similar discounted rates as the current preemptible VMs. Deflation is amenable to multiple pricing models. Providers could continue to offer flat discounted prices, or dynamic supply-demand based pricing. The resource-as-a-service model [14] also fits well for deflatable VMs: providers can dynamically charge VMs based on the amount of resources allocated. If deflatable VMs present a higher utility to applications (which we believe they do), then they can allow providers to charge higher prices for their surplus resources.

Impact on Applications: Deploying applications on deflatable VMs also introduces additional complexity in the deployment model. Implementing application-level deflation policies that is required for cascade deflation is the primary concern when deploying applications on deflatable VMs. However, we have shown that these policies can be easily implemented for popular cloud applications [31] such as key-value stores, Java-based enterprise applications, distributed data processing, and machine learning [2].

There are also questions about whether applications prefer frequent fail-stop failures (current preemptible VMs), or the occasional performance variation imposed by deflation. High deflation levels, albeit rare, could increase the likelihood of gray failures [43]. Finally, the superior performance of deflatable VMs and their significantly higher availability may prove to be a significant driving force behind their adoption.

9 Conclusion

We proposed the notion of resource deflation as an alternative to preemption, for running low-priority applications. Deflatable VMs allow applications to continue running even under resource pressure, albeit at a lower performance. Our cascade deflation approach uses hypervisor, OS, and application level reclamation mechanisms and policies. This multi-level approach allows many applications, such as distributed deep learning training, to run with only 20% performance degradation even when half their resources are dynamically reclaimed. Deflation is a promising cluster-management primitive, and compares favorably to preemption, in terms of cluster throughput, utilization, and application preemptions.

Acknowledgements. We thank our shepherd John Regehr and all the reviewers for their insightful comments that helped us improve the paper. This work is supported in part by NSF grants #1422245 and #1229059, and by Amazon Web Services (AWS) Cloud Credits. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/, September 24th 2015.
[2] Eucalyptus workload traces. https://www.cs.ucsb.edu/~rich/workload/, 2015.

[2] Web-application clusters are another popular cloud workload, and can use a deflation-aware load-balancer for cascade deflation.

[3] Google preemptible instances. https://cloud.google.com/compute/docs/instances/preemptible, September 24th 2015.

[4] Linux CPU Hotplug Documentation. https://www.kernel.org/doc/html/v4.18/core-api/cpu_hotplug.html, December 2016.

[5] Memcached. https://memcached.org/, 2016.

[6] Azure low priority batch VMs. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms, June 2017.

[7] IBM J9 Java Virtual Machine. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/user/java_jvm.html, 2017.

[8] Amazon EC2 Spot Instance Advisor. https://aws.amazon.com/ec2/spot/instance-advisor/, July 2018.

[9] Intel BigDL sample models. https://github.com/intel-analytics/BigDL/tree/master/spark/dl/src/main/scala/com/intel/analytics/bigdl/models, September 2018.

[10] Libvirt: The virtualization API. http://www.libvirt.org, September 2018.

[11] Linux cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, September 2018.

[12] Openstack. https://www.openstack.org/, September 2018.

[13] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[14] Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. The rise of RaaS: the resource-as-a-service cloud. *Communications of the ACM 57*, 7 (2014), 76–84.

[15] Ali-Eldin, A., Tordsson, J., and Elmroth, E. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS),* (2012), IEEE, pp. 204–212.

[16] Amit, N., Tsafrir, D., and Schuster, A. Vswapper: A memory swapper for virtualized environments. *VEE* (2014).

[17] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[18] Ben-Yehuda, M., Agmon Ben-Yehuda, O., and Tsafrir, D. The nom profit-maximizing operating system. In *VEE* (2016), ACM.

[19] Bobroff, N., Westerink, P., and Fong, L. Active control of memory for Java virtual machines and applications. In *ICAC* (2014), pp. 97–103.

[20] Cameron, C., Singer, J., and Vengerov, D. The judgment of forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 143–156.

[21] Cavdar, D., Chen, L. Y., and Alagoz, F. Reducing execution waste in priority scheduling: a hybrid approach. In *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)* (2016).

[22] Cheatham, T., Fahmy, A., Stefanescu, D., and Valiant, L. Bulk synchronous parallel computing-a paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[23] Checconi, F., Cucinotta, T., Faggioli, D., and Lipari, G. Hierarchical multiprocessor cpu reservations for the linux kernel. In *Proceedings of the 5th international workshop on operating systems platforms for embedded real-time applications (OSPERT 2009), Dublin, Ireland* (2009), pp. 15–22.

[24] Chehab, M. C., et al. Memory hotplug. *Linux Kernel Documentation* (2007). Available online: https://github.com/torvalds/linux/blob/486088bc4689f826b80aa317b45ac9e42e8b25ee/Documentation/memory-hotplug.txt.

[25] Chen, J., Pan, X., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting distributed synchronous SGD. *ICLR* (2017).

[26] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, ACM, pp. 153–167.

[27] Delgado, P., Dinu, F., Kermarrec, A.-M., and Zwaenepoel, W. Hawk: Hybrid datacenter scheduling. In *USENIX ATC* (2015).

[28] Delimitrou, C., and Kozyrakis, C. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 127–144.

[29] Ding, X., Gibbons, P. B., Kozuch, M. A., and Shan, J. Gleaner: mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *USENIX Annual Technical Conference* (2014), pp. 73–84.

[30] Fang, L., Nguyen, K., Xu, G., Demsky, B., and Lu, S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, ACM, pp. 394–409.

[31] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 37–48.

[32] Gandhi, A., Dube, P., Kochut, A., and Zhang, L. Model-driven autoscaling for hadoop clusters. In *2015 IEEE International Conference on Autonomic Computing (ICAC)* (2015), IEEE, pp. 155–156.

[33] Gandhi, A., Harchol-Balter, M., Raghunathan, R., and Kozuch, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 14.

[34] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI* (2011).

[35] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M., and Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, 2016), USENIX, pp. 99–115.

[36] Gong, Z., Gu, X., and Wilkes, J. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management* (2010), IEEE.

[37] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 455–466.

[38] Gulati, A., Holler, A., Ji, M., Shanmuganathan, G., Waldspurger, C., and Zhu, X. VMware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal 1*, 1 (2012), 45–64.

[39] Gupta, V., Lee, M., and Schwan, K. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Virtual Execution Environments (VEE)* (2015), ACM, pp. 79–92.

[40] Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *European Conference on Computer Systems* (2017), EuroSys '17, ACM, pp. 589–604.

[41] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), USENIX.

[42] Hines, M. R., Gordon, A., Silva, M., Da Silva, D., Ryu, K., and Ben-Yehuda, M. Applications know best: Performance-driven memory overcommit with Ginkgo. In *Cloud Computing Technology and Science (CloudCom)* (2011), IEEE, pp. 130–137.

[43] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS âĂŹ17* (2017), ACM Press, p. 150âĂŞ155.

[44] Ilyushkin, A., Ali-Eldin, A., Herbst, N., Bauer, A., Papadopoulos, A. V., Epema, D., and Iosup, A. An experimental performance evaluation of autoscalers for complex workflows. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 3*, 2 (2018), 8.

[45] Iorgulescu, C., Dinu, F., Raza, A., Hassan, W. U., and Zwaenepoel, W. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 97–109.

[46] Jiang, J., Cui, B., Zhang, C., and Yu, L. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 463–478.

[47] Joel H Schopp, K. F., and Silbermann, M. J. Resizing Memory with Balloons and Hotplug. In *Ottawa Linux Symposium (OLS)* (2006), pp. 313–319.

[48] Kalyvianaki, E., Charalambous, T., and Hand, S. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing* (New York, NY, USA, 2009), ICAC '09, ACM, pp. 117–126.

[49] Kambatla, K., Yarlagadda, V., Goiri, Ã., and Grama, A. Ubis: Utilization-aware cluster scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2018), pp. 358–367.

[50] Karanasos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G. M., Heddaya, S., Ramakrishnan, R., and Sakalanaga, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX ATC* (2015).

[51] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: The Linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[52] Klein, C., Maggio, M., Årzén, K.-E., and Hernández-Rodriguez, F. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 700–711.

[53] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014), vol. 14, pp. 583–598.

[54] Liu, H., Jin, H., Liao, X., Deng, W., He, B., and Xu, C.-z. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems 26*, 5 (2015), 1350–1363.

[55] Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing 12*, 4 (2014), 559–592.

[56] Marathe, A., Harris, R., Lowenthal, D., De Supinski, B. R., Rountree, B., and Schulz, M. Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2. In *HPDC* (2014), ACM.

[57] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research 17*, 1 (2016), 1235–1241.

[58] Misra, P. A., Goiri, Í., Kace, J., and Bianchini, R. Scaling distributed file systems in resource-harvesting datacenters. In *2017 USENIX Annual Technical Conference* (2017), USENIX Association, pp. 799–811.

[59] Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. Sparknet: Training deep networks in spark. *ICLR* (2016).

[60] Nguyen, H., Shen, Z., Gu, X., Subbiah, S., and Wilkes, J. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (2013), pp. 69–82.

[61] Nitu, V., Teabe, B., Fopa, L., Tchana, A., and Hagimont, D. StopGap: Elastic VMs to enhance server consolidation. *Software: Practice and Experience 47*, 11 (2017), 1501–1519.

[62] Ouyang, J., and Lange, J. R. Preemptable ticket spinlocks: improving consolidated performance in the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 191–200.

[63] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[64] Rosa, A., Chen, L. Y., and Binder, W. Understanding the dark side of big data clusters: An analysis beyond failures. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2015), IEEE, pp. 207–218.

[65] Salomie, T.-I., Alonso, G., Roscoe, T., and Elphinstone, K. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 337–350.

[66] Shahrad, M., Klein, C., Zheng, L., Chiang, M., Elmroth, E., and Wentzlaf, D. Incentivizing self-capping to increase cloud utilization. In *ACM Symposium on Cloud Computing 2017 (SoCC'17)* (2017), Association for Computing Machinery (ACM).

[67] Sharma, P., Guo, T., He, X., Irwin, D., and Shenoy, P. Flint: Batch-interactive data-intensive processing on transient servers. In *EuroSys* (2016), ACM.

[68] Sharma, P., Irwin, D., and Shenoy, P. Portfolio-driven resource management for transient cloud servers. In *Proceedings of ACM Measurement and Analysis of Computer Systems* (June 2017), vol. 1, p. 23.

[69] Sharma, P., and Kulkarni, P. Singleton: system-wide page deduplication in virtual environments. In *HPDC* (2012), ACM.

[70] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P. Spotcheck: Designing a derivative IaaS cloud on the spot market. In *EuroSys* (2015), ACM, p. 16.

[71] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Symposium on Cloud Computing* (2011), ACM.

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

[73] Stockhammer, T. Dynamic adaptive streaming over HTTP: Standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (2011), ACM, pp. 133–144.

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

[75] Teabe, B., Nitu, V., Tchana, A., and Hagimont, D. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *European Conference on Computer Systems* (2017), EuroSys '17, ACM, pp. 286–297.

[76] Tomas, L., and Tordsson, J. An autonomic approach to risk-aware data center overbooking. In *Transactions on Cloud Computing* (2014), vol. 2, IEEE, pp. 292–305.

[77] Urgaonkar, B., Shenoy, P., and Roscoe, T. Resource overbooking and application profiling in shared hosting platforms. *SOSP* (2002).

[78] Vanga, M., Gujarati, A., and Brandenburg, B. B. Tableau: A high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 28:1–28:16.

[79] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *EuroSys* (2015), ACM.

[80] Waldspurger, C. A. Memory resource management in VMware ESX Server. *OSDI* (2002), 181–194.

[81] Wang, C., Urgaonkar, B., Gupta, A., Kesidis, G., and Liang, Q. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *European Conference on Computer Systems* (2017), EuroSys '17, ACM, pp. 620–634.

[82] Wang, Y., Qiu, X., Ding, D., Zhang, Y., Wang, Y., Jia, X., Wan, Y., Li, Z., Wang, J., Huang, S., et al. BigDL: A distributed deep learning framework for big data. *arXiv preprint arXiv:1804.05839* (2018).

[83] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[84] Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., and Moscibroda, T. TR-Spark: Transient Computing for Big Data Analytics. In *SoCC* (October 2016), ACM.

[85] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 103–116.

[86] Yang, Y., Kim, G.-W., Song, W. W., Lee, Y., Chung, A., Qian, Z., Cho, B., and Chun, B.-G. Pado: A data processing engine for harnessing transient resources in datacenters. In *European Conference on Computer Systems* (2017), EuroSys '17, ACM, pp. 575–588.

[87] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[88] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. *HotCloud* (2010).

[89] Zhang, D., Ehsan, M., Ferdman, M., and Sion, R. Dimmer: A case for turning off dimms in clouds. In *Symposium on Cloud Computing* (2014), ACM, pp. 1–8.

[90] Zhang, Y., Prekas, G., Fumarola, G. M., Fontoura, M., Goiri, I., and Bianchini, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 755–770.