History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters

Yunqi Zhang[†*] George Prekas[‡*] Giovanni Matteo Fumarola[δ]
Marcus Fontoura[δ] Íñigo Goiri[⋆] Ricardo Bianchini[⋆]

[†]*University of Michigan* [‡]*EPFL* [δ]*Microsoft* [⋆]*Microsoft Research*

Abstract

An effective way to increase utilization and reduce costs in datacenters is to co-locate their latency-critical services and batch workloads. In this paper, we describe systems that harvest spare compute cycles and storage space for co-location purposes. The main challenge is minimizing the performance impact on the services, while accounting for their utilization and management patterns. To overcome this challenge, we propose techniques for giving the services priority over the resources, and leveraging historical information about them. Based on this information, we schedule related batch tasks on servers that exhibit *similar* patterns and will likely have enough available resources for the tasks' durations, and place data replicas at servers that exhibit *diverse* patterns. We characterize the dynamics of how services are utilized and managed in ten large-scale production datacenters. Using real experiments and simulations, we show that our techniques eliminate data loss and unavailability in many scenarios, while protecting the co-located services and improving batch job execution time.

1 Introduction

Motivation. Purchasing servers dominates the total cost of ownership (TCO) of large-scale datacenters [4], such as those operated by Google and Microsoft. Unfortunately, the servers' average utilization is often low, especially in clusters that host user-facing, interactive services [4, 10]. The reasons for this include: these services are often latency-critical (*i.e.*, require low tail response times); may exhibit high peaks in user load; and must reserve capacity for unexpected load spikes and failures.

An effective approach for extracting more value from the servers is the co-location of useful batch workloads (*e.g.*, data analytics, machine learning) and the data they require on the same servers that perform other functions, including those that run latency-critical services. However, for co-location with these services to be acceptable, we must shield them from any non-trivial performance interference produced by the batch workloads or their storage accesses, even when unexpected events occur. If co-location starts to degrade response times, the scheduler must throttle or even kill (and re-start elsewhere) the culprit batch workloads. In either case, the performance of the batch workloads suffers. Nevertheless, co-location ultimately reduces TCO [37], as the batch workloads are not latency-critical and share the same infrastructure as the services, instead of needing their own.

Recent scheduling research has considered how to carefully select which batch workload to co-locate with each service to minimize the potential for interference (most commonly, last-level cache interference), *e.g.* [9, 10, 25, 42]. However, these works either assume simple sequential batch applications or overlook the resource utilization dynamics of real services. Scheduling data-intensive workloads comprising many distributed tasks (*e.g.*, data analytics jobs) is challenging, as scheduling decisions must be made in tandem for collections of these tasks for best performance. The resource utilization dynamics make matters worse. For example, a long-running workload may have some of its tasks throttled or killed when the load on a co-located service increases.

Moreover, no prior study has explored in detail the co-location of services with data for batch workloads. Real services often leave large amounts of spare storage space (and bandwidth) that can be used to store the data needed by the batch workloads. However, co-locating storage raises even more challenges, as the management and utilization of the services may affect data durability and availability. For example, service engineers and the management system itself may reimage (reformat) disks, deleting all of their data. Reimaging typically results from persistent state management, service deployment, robustness testing, or disk failure. Co-location and reimaging may cause all replicas of a data block to be

[*]This work was done while Zhang and Prekas were interns at MSR.

destroyed before they can be re-generated.

Our work. In this paper, we propose techniques for harvesting the spare compute cycles and storage space in datacenters for distributed batch workloads. We refer to the original workloads of each server as its "primary tenant", and to any resource-harvesting workload (*i.e.*, batch compute tasks or their storage accesses) on the server as a "secondary tenant". We give priority over each server's resources to its primary tenant; secondary tenants may be killed (in case of tasks) or denied (in case of storage accesses) when the primary tenant needs the resources.

To reduce the number of task killings and improve data availability and durability, *we propose task scheduling and data placement techniques that rely on historical resource utilization and disk reimaging patterns.* We logically group primary tenants that exhibit similar patterns in these dimensions. Using the utilization groups, our scheduling technique schedules related batch tasks on servers that have *similar* patterns and enough resources for the tasks' expected durations, and thereby avoids creating stragglers due to a lack of resources. Using the utilization and reimaging groups, our data placement technique places data replicas in servers with *diverse* patterns, and thereby increases durability and availability despite the harvested nature of the storage resources.

To create the groups, we characterize the primary tenants' utilization and reimaging patterns in ten production datacenters,[1] including a popular search engine and its supporting services. Each datacenter hosts up to tens of thousands of servers. Our characterization shows that the common wisdom that datacenter workloads are periodic is inaccurate, since often most servers do not execute interactive services. We target *all* servers for harvesting.

Implementation and results. We implement our techniques into the YARN scheduler, Tez job manager, and HDFS file system [11, 29, 36] from the Apache Hadoop stack. (Primary tenants use their own scheduling and file systems.) Stock YARN and HDFS assume there are no external workloads, so we also make these systems aware of primary tenants and their resource usage.

We evaluate our systems using 102 servers in a production datacenter, with utilization and reimaging behaviors scaled down from it. We also use simulations to study our systems for longer periods and for larger clusters. The results show that our systems (1) can improve the average batch job execution time by up to 90%; and (2) can reduce data loss by more than two orders of magnitude when blocks are replicated three times, eliminate data loss under four-way replication, and eliminate data unavailability for most utilization levels.

Finally, we recently deployed our file system in large-scale production (our scheduler is next), so we discuss our experience and lessons that may be useful to others.

Summary and conclusions. Our contributions are:

- We characterize the dynamics of how servers are used and managed in ten production datacenters.
- We propose techniques for improving task scheduling and data placement based on the historical behavior of primary tenants and how they are managed.
- We extend the Hadoop stack to harvest the spare cycles and storage in datacenters using our techniques.
- We evaluate our systems using real experiments and simulations, and show large improvements in batch job performance, data durability, and data availability.
- We discuss our experience with large-scale production deployments of our techniques.

We conclude that resource harvesting benefits significantly from a detailed accounting of the resource usage and management patterns of the primary workloads. This accounting enables higher utilization and lower TCO.

2 Related Work

Datacenter characterization. Prior works from datacenter operators have studied selected production clusters, not entire datacenters, *e.g.* [37]. Instead, we characterize *all* primary tenants in ten datacenters, including those used for production latency-critical and non-critical services, for service development and testing, and those awaiting use or being prepared for decommission.

Harvesting of resources without co-location. Prior works have proposed to harvest resources for batch workloads in the absence of co-located latency-critical services, *e.g.* [22, 23]. Our work focuses on the more challenging co-location scenario in modern datacenters.

Co-location of latency-critical and batch tasks. Recent research has targeted two aspects of co-location: (1) performance isolation – ensuring that batch tasks do not interfere with services, after they have been co-located on the same server [19, 20, 21, 24, 27, 31, 32, 38, 42]; or (2) scheduling – selecting which tasks to co-locate with each service to minimize interference or improve packing quality [9, 10, 12, 25, 37, 43]. Borg addresses both aspects in Google's datacenters, using Linux cgroup-based containers, special treatment for latency-critical tasks, and resource harvesting from containers [37].

Our work differs substantially from these efforts. As isolation and interference-aware scheduling have been well-studied, we leave the implementation of these techniques for future work. Instead, we reserve compute resources that cannot be given to batch tasks; a spiking primary tenant can immediately consume this reserve until our software can react (within a few seconds at most) to replenish the reserve. Combining our systems with finer grained isolation techniques will enable smaller reserves.

[1] For confidentiality, we omit certain information, such as absolute numbers of servers and actual utilizations, focusing instead on coarse behavior patterns and full-range utilization exploration.

Moreover, unlike services at Google, our primary tenants "own" their servers, and do not declare their potential resource needs. This means that we must harvest resources carefully to prevent interference with latency-critical services and degraded batch job performance. Thus, we go beyond prior works by understanding and exploiting the primary tenants' resource usage dynamics to reduce the need for killing batch tasks. With respect to resource usage dynamics, a related paper is [5], which derives Service-Level Objectives (SLOs) for resource availability from historical utilization data. We leverage similar data but for dynamic task scheduling, which their paper did not address.

Also importantly, we are the first to explore in detail the harvesting of storage space from primary tenants for data-intensive batch jobs. This scenario involves understanding how primary tenants are managed, as well as their resource usage.

For both compute and storage harvesting, we leverage primary and secondary tenants' historical behaviors, which are often more accurate than user annotations/estimates (e.g., [35]). Any system that harvests resources from latency-critical workloads can benefit from leveraging the same behaviors.

Data-processing frameworks and co-location. Researchers have proposed improvements to the Hadoop stack in the absence of co-location, *e.g.* [3, 8, 13, 14, 15, 18, 39]. Others considered Hadoop (version 1) in co-location scenarios using virtual machines, but ran HDFS on dedicated servers [7, 30, 41]. Lin *et al.* [22] stored data on dedicated and volunteered computers (idle desktops), but in the absence of primary tenants. We are not aware of studies of Mesos [16] in co-location scenarios. Bistro [12] relies on static resource reservations for services, and schedules batch jobs on the leftover resources. In contrast to these works, we propose dynamic scheduling and data placement techniques for the Hadoop stack, and explore the performance, data availability, and data durability of co-located primary and secondary tenants.

3 Characterizing Behavior Patterns

We now characterize the primary tenants in ten production datacenters. In later sections, we use the characterization for our co-location techniques and results.

3.1 Data sources and terminology

We leverage data collected by AutoPilot [17], the primary tenant management and deployment system used in the datacenters. Under AutoPilot, each server is part of an *environment* (a collection of servers that are logically related, *e.g.* indexing servers of a search engine) and executes a *machine function* (a specific functionality, *e.g.* result ranking). Environments can be used for production, development, or testing. In our terminology, each primary tenant is equivalent to an <environment, machine function> pair. Primary tenants run on physical hardware, *without* virtualization. Each datacenter has between a few hundred to a few thousand primary tenants.

Though our study focuses on AutoPilot-managed datacenters, our characterization and techniques should be easily applicable to other management systems as well. In fact, similar telemetry is commonly collected in other production datacenters, *e.g.* GWP [28] at Google and Scuba [2] at Facebook.

3.2 Resource utilization

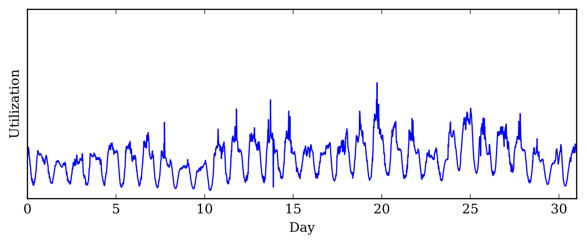
AutoPilot records the primary tenant utilization per server for all hardware resources, but for simplicity we focus on the CPU in this paper. It records the CPU utilization every two minutes. As the load is not always evenly balanced across all servers of a primary tenant, we compute the average of their utilizations in each time slot, and use the utilization of this *"average"* server for one month to represent the primary tenant.

We then identify trends in the tenants' utilizations, using signal processing. Specifically, we use the Fast Fourier Transform (FFT) on the data from each primary tenant individually. The FFT transforms the utilization time series into the frequency domain, making it easy to identify any periodicity (and its strength) in the series.

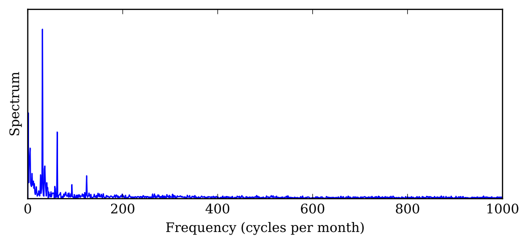
We identify three main classes of primary tenants: *periodic*, *unpredictable*, and (roughly) *constant*. Figure 1 shows the CPU utilization trends of a periodic and an unpredictable primary tenant in the time and frequency domains. Figure 1b shows a strong signal at frequency 31, because there are 31 days (load peaks and valleys) in that month. In contrast, Figure 1d shows a decreasing trend in signal strength as the frequency increases, as the majority of the signal derives from events that rarely happen (*i.e.*, exhibit lower frequency).

As one would expect, user-facing primary tenants often exhibit periodic utilization (*e.g.*, high during the day and low at night), whereas non-user-facing (*e.g.*, Web crawling, batch data analytics) or non-production (*e.g.*, development, testing) primary tenants often do not. For example, a Web crawling or data scrubber tenant may exhibit (roughly) constant utilization, whereas a testing tenant often exhibits unpredictable utilization behavior.

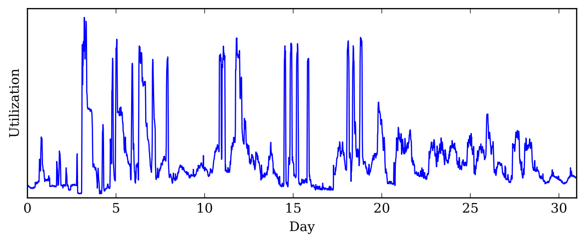
More interestingly, Figure 2 shows that user-facing (periodic) primary tenants are actually a small minority. The vast majority of primary tenants exhibit roughly constant CPU utilization. Nevertheless, Figure 3 shows that the periodic primary tenants represent a large percentage (~40% on average) of the servers in each datacenter.



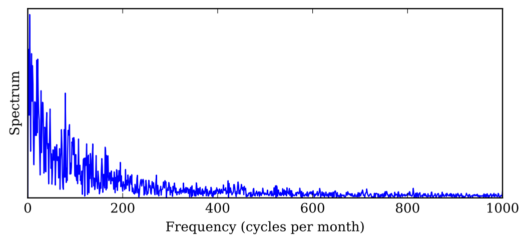
(a) Periodic – time



(b) Periodic – frequency



(c) Unpredictable – time



(d) Unpredictable – frequency

Figure 1: Sample periodic and unpredictable one-month traces in the time and frequency domains.

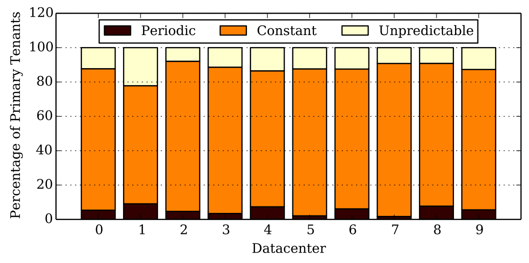


Figure 2: Percentages of primary tenants per class.

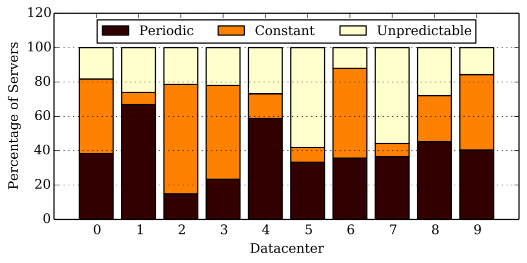


Figure 3: Percentages of servers per class.

Still, the non-periodic primary tenants account for more than half of the tenants and servers.

Most importantly, the vast majority of servers (~75%) run primary tenants (periodic and constant) for which the historical utilization data is a good predictor of future behaviors (the utilizations repeat periodically or all the time). Thus, leveraging this data should improve the quality of both our task scheduling and data placement.

3.3 Disk reimaging

Disk reimages are relatively frequent for some primary tenants, which by itself potentially threatens data durability under co-location. Even worse, disk reimages are often correlated, *i.e.* many servers might be reimaged at the same time (*e.g.*, when servers are repurposed from one primary tenant to another). Thus, it is critical for data durability to account for reimages and correlations.

AutoPilot collects disk reimaging (reformatting) data per server. This data includes reimages of multiple types: (1) those initiated manually by developers or service operators intending to re-deploy their environments (primary tenants) or re-start them from scratch; (2) those initiated by AutoPilot to test the resilience of production services; and (3) those initiated by AutoPilot when disks have undergone maintenance (*e.g.*, tested for failure).

We now study the reimaging patterns using three years of data from AutoPilot. As an example of the reimaging frequencies we observe, Figure 4 shows the Cumulative Distribution Function (CDF) of the average number of reimages per month for each server in three years in five representative datacenters in our sample. Figure 5 shows the CDF of the average number of reimages per server per month for each primary tenant for the same years and datacenters. The discontinuities in this figure are due to short-lived primary tenants.

We make three observations from these figures. First and most importantly, there is a good amount of diver-

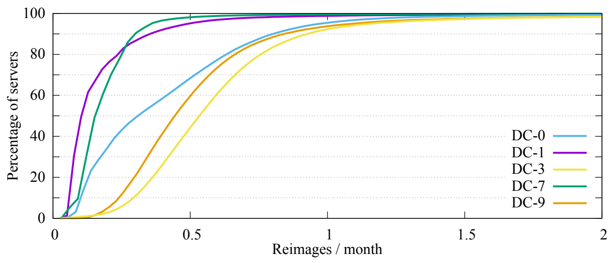


Figure 4: Per-server number of reimages in three years.

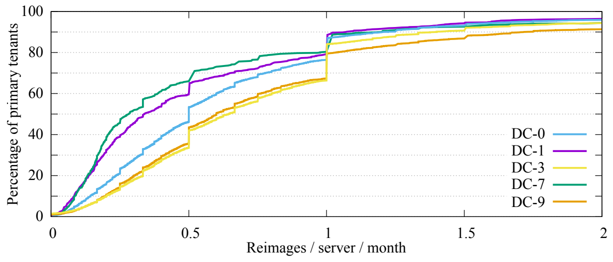


Figure 5: Per-tenant number of reimages in three years.

sity in average reimaging frequency across primary tenants in each datacenter (Figure 5 does not show nearly vertical lines). Second, the reimaging frequencies per month are fairly low in all datacenters. For example, at least 90% of servers are reimaged once or fewer times per month on average, whereas at least 80% of primary tenants are reimaged once or fewer times per server per month on average. This shows that reimaging by primary tenant engineers and AutoPilot is not overly aggressive on average, but there is a significant tail of servers (10%) and primary tenants (20%) that are reimaged relatively frequently. Third, the primary tenant reimaging behaviors are fairly consistent across datacenters, though three datacenters show substantially lower reimaging rates per server (we show two of those datacenters in Figure 4).

The remaining question is whether each primary tenant exhibits roughly the same frequencies month after month. In this respect, we find that there is substantial variation, as frequencies sometimes change substantially.

Nevertheless, when compared to each other, primary tenants tend to rank consistently in the same part of the spectrum. In other words, primary tenants that experience a relatively small (large) number of reimages in a month tend to experience a relatively small (large) number of reimages in the following month. To verify this trend, we split the primary tenants of a datacenter into three frequency groups, each with the same number of tenants: *infrequent*, *intermediate*, and *frequent*. Then, we track the movement of the primary tenants across these groups over time. Figure 6 plots the CDF of the number of times a primary tenant changed groups from one month to the next. At least 80% of primary tenants

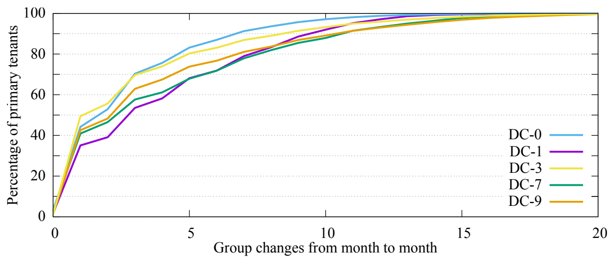


Figure 6: Number of times a primary tenant changed reimage frequency groups in three years.

changed groups only 8 or fewer times out of the possible 35 changes in three years. This behavior is also consistent across datacenters.

Again, these figures show that historical reimaging data should provide meaningful information about the future. Using this data should improve data placement.

4 Smart Co-location Techniques

In this section, we describe our techniques for smart task scheduling and data placement, which leverage the primary tenants' historical behavior patterns.

4.1 Smart task scheduling

We seek to schedule batch tasks (secondary tenants) to harvest spare cycles from servers that natively run interactive services and their supporting workloads (primary tenants). Modern cluster schedulers achieve high job performance and/or fairness, so they are good candidates for this use. However, their designs typically assume dedicated servers, *i.e.* there are no primary tenants running on the same servers. Thus, we must (1) modify them to become aware of the primary tenants and the primary tenants' priority over the servers' resources; and (2) endow them with scheduling algorithms that reduce the number of task killings resulting from the co-located primary tenants' need for resources. The first requirement is fairly easy to accomplish, so we describe our implementation in Section 5. Here, we focus on the second requirement, *i.e.* smart task scheduling, and use historical primary tenant utilization data to select servers that will most likely have the required resources available throughout the tasks' entire executions.

Due to the sheer number of primary tenants, it would be impractical to treat them independently during task scheduling. Thus, our scheduling technique first clusters together primary tenants that have similar utilization patterns into the same utilization *class*, and then select a class for the tasks of a job. Next, we discuss our clustering and class selection algorithms in turn.

Algorithm 1 Class selection algorithm.

```
1: Given: Classes $C$, Headroom($type$,$c$), Ranking Weights $W$
2: function SCHEDULE(Batch job $J$)
3:     $J$.type = Length (short, medium, or long) from its last run
4:     $J$.req = Max amount of concurrent resources from DAG
5:     for each $c \in C$ do
6:         $c$.weightedroom=Headroom($J$.type,$c$) $\times$ $W$[$J$.type,$c$.class]
7:     end for
8:     $F = \{\forall c \in C |$ Headroom($J$.type,$c$) $\geq J$.req$\}$
9:     if $F \neq \emptyset$ then
10:        Pick 1 class $c \in F$ probabilistically $\propto c$.weightedroom
11:        return $\{c\}$
12:    else if Job $J$ can fit in multiple classes combined then
13:        Pick $\{c_0,\ldots,c_k\} \subseteq C$ probabilistically $\propto c$.weightedroom
14:        return $\{c_0,\ldots,c_k\}$
15:    else
16:        Do not pick classes
17:        return $\{\emptyset\}$
18:    end if
19: end function
```

The **clustering** algorithm periodically (*e.g.*, once per day) takes the most recent time series of CPU utilizations from the average server of each primary tenant, runs the FFT algorithm on the series, groups the tenants into the three patterns described in Section 3 (periodic, constant, unpredictable) based on their frequency profiles, and then uses the K-Means algorithm to cluster the profiles in each pattern into classes. Clustering tags each class with the utilization pattern, its average utilization, and its peak utilization. It also maintains a mapping between the classes and their primary tenants.

As we detail in Algorithm 1, our **class selection** algorithm relies on the classes defined by the clustering algorithm. When we need to allocate resources for a job's tasks, the algorithm selects a class (or classes) according to the expected job length (line 3) and a predetermined ranking of classes for the length. We represent the desired ranking using weights (line 6); higher weight means higher ranking. For a long job, we give priority to constant classes first, then periodic classes, and finally unpredictable classes. We prioritize the constant classes in this case because constant-utilization primary tenants with enough available resources are unlikely to take resources away from the job during its execution. At the other extreme, a short job does not require an assurance of resource availability long into the future; knowing the current utilization is enough. Thus, for a short job, we rank the classes unpredictable first, then periodic, and finally constant. For a medium job, the ranking is periodic first, then constant, and finally unpredictable.

We categorize a job as short, medium, or long by comparing the duration of its last execution to two predefined thresholds (line 3). We set the thresholds based on the historical distribution of job lengths and the current computational capacity of each preferred tenant class (*e.g.*, the total computation required by long jobs should be proportional to the computational capacity of constant primary tenants). Importantly, the last duration need *not* be an accurate execution time estimate. Our goal is much easier: to categorize jobs into three rough types. We assume that a job that has not executed before is a medium job. After a possible error in this first guess, we find that a job consistently falls into the same type.

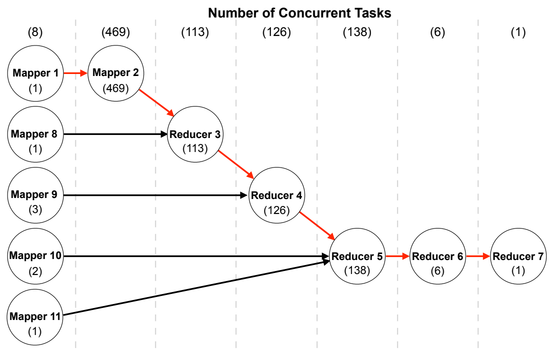


Figure 7: Example job execution DAG.

We estimate the maximum amount of concurrent resources that the job will need (line 4) using a breadth-first traversal of the job's directed acyclic graph (DAG), which is a common representation of execution flows in many frameworks [1, 29, 40]. We find this estimate to be accurate for our workloads. Figure 7 shows an example job DAG (query 19 from TPC-DS [34]), for which we estimate a maximum of 469 concurrent containers.

Whether a job "fits" in a class (line 8) depends on the amount of available resources (or the amount of *headroom*) that the servers in the class currently exhibit, as we define below. When multiple classes could host the job, the algorithm selects one with probability proportional to its weighted headroom (lines 9 and 10). If multiple classes are necessary, it selects as many classes as needed, again probabilistically (lines 12 and 13). If there are not enough resources available in any combination of classes, it does not select any class (line 16).

The headroom depends on the job type. For a short job, we define it as 1 minus the current average CPU utilization of the servers in the class. For a medium job, we use 1 minus Max(average CPU utilization, current CPU utilization). For a long job, we use 1 minus Max(peak CPU utilization, current CPU utilization).

4.2 Smart data placement

Modern distributed file systems achieve high data access performance, availability, and durability, so there is a strong incentive for using them in our harvesting scenario. However, like cluster schedulers, they assume dedicated servers without primary tenants running and storing data on the same servers, and without primary tenant owners deliberately reimaging disks. Thus, we

Algorithm 2 Replica placement algorithm.

```
1:  Given: Storage space available in each server, Primary reimaging
2:         stats, Primary peak CPU util stats, Desired replication $R$
3:  function PLACE REPLICAS(Block $B$)
4:      Cluster primary tenants wrt reimaging and peak CPU util
5:          into 9 classes, each with the same total space
6:      Select the class of the server creating the block
7:      Select the server creating the block for one replica
8:      for $r = 2$; $r \leq R$; $r = r + 1$ do
9:          Select the next class randomly under two constraints:
10:             No class in the same row has been picked
11:             No class in the same column has been picked
12:         Pick a random primary tenant of this class as long as
13:             its environment has not received a replica
14:         Pick a server in this primary tenant for the next replica
15:         if ($r$ mod 3) == 0 then
16:             Forget rows and columns that have been selected so far
17:         end if
18:     end for
19: end function
```

must (1) modify them to become co-location-aware; and (2) endow them with replica placement algorithms that improve data availability and durability in the face of primary tenants and how they are managed. Again, the first requirement is fairly easy to accomplish, so we discuss our implementation in Section 5. Here, we focus on the second requirement, *i.e.* smart replica placement.

The challenge is that the primary tenants and the management system may hurt data availability and durability for any block: (1) if the replicas of a block are stored in primary tenants that load-spike at the same time, the block may become unavailable; (2) if developers or the management system reimage the disks containing all the replicas of a block in a short time span, the block will be lost. A replica placement algorithm must then account for primary tenant and management system activity.

An intuitive best-first approach would be to try to find primary tenants that reimage their disks the least, and from these primary tenants select the ones that have lowest CPU utilizations. However, this greedy approach has two serious flaws. First, it treats durability and availability independently, one after the other, ignoring their interactions. Second, after the space at all the "good" primary tenants is exhausted, new replicas would have to be created at locations that would likely lead to poor durability, poor availability, or both.

We prefer to make decisions that promote durability and availability at the same time, while consistently spreading the replicas around as evenly as possible across all types of primary tenants. Thus, our replica placement algorithm (Algorithm 2) creates a two-dimensional clustering scheme, where one dimension corresponds to durability (disk reimages) and the other to availability (peak CPU utilization). It splits the two-dimensional space into 3×3 classes (infrequent, intermediate, and frequent reimages versus low, medium, and high peak utilizations), each of which has the same amount of available storage for harvesting $S/9$, where S is the total amount of currently available storage (lines 4 and 5). This idea can be applied to splits other than 3×3, as long as they provide enough primary tenant diversity.

The above approach tries to balance the available space across classes. However, perfect balancing may be impossible when primary tenants have widely different amounts of available space, and the file system starts to become full. The reason is that balancing space perfectly could require splitting a large primary tenant across two or more classes. We prevent this situation by selecting a single class for each tenant, to avoid hurting placement diversity. The side effect is that small primary tenants get filled more quickly, causing larger primary tenants to eventually become the only possible targets for the replicas. This effect can be eliminated by not filling the file system to the point that less than three primary tenants remain as possible targets for replicas. In essence, there is a tradeoff between space utilization and diversity. We discuss this tradeoff further in Section 7.

When a client creates a new block, our algorithm selects one class for each replica. The first class is that of the server creating the block; the algorithm places a replica at this server to promote locality (lines 6 and 7). If the desired replication is greater than 1, it repeatedly selects classes randomly, in such a way that no row or column of the two-dimensional space has two selections (lines 9, 10, and 11). It places a replica in (a randomly selected server of) a randomly selected primary tenant in this class, while ensuring that no two primary tenants in the same environment receive a replica (lines 12, 13, and 14). Finally, for a desired replication level larger than 3, it does extra rounds of selections. At the beginning of each round, it forgets the history of row and column selections from the previous round (lines 15, 16, and 17).

The environment constraint is the only aspect of our techniques that is AutoPilot-specific. However, the constraints generalize to any management system: avoid placing multiple replicas in any logical (*e.g.*, environment) or physical (*e.g.*, rack) server grouping that induces correlations in resource usage, reimaging, or failures.

Figure 8 shows an example of our clustering scheme and primary tenant selection, assuming all primary tenants have the same amount of available storage. The rows defining the peak utilization classes do not align, as we ensure that the available storage is the same in all classes.

5 System Implementations

We implement our techniques into YARN, Tez, and HDFS. Next, we overview these systems. Then, we describe our implementation guidelines and systems, called YARN-H, Tez-H, and HDFS-H ("-H" refers to history).

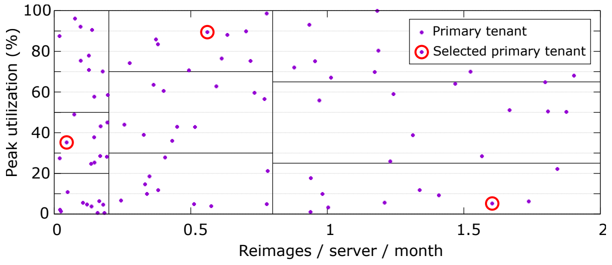


Figure 8: Two-dimensional clustering scheme.

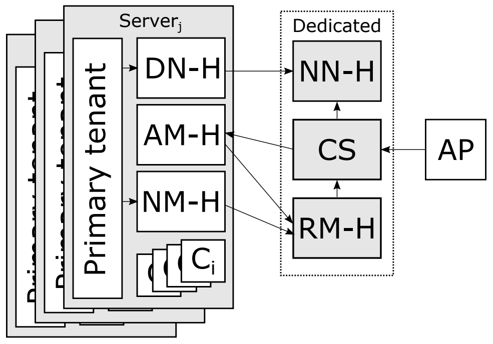


Figure 9: Overview of YARN-H (RM-H and NM-H), Tez-H (AM-H), and HDFS-H (NN-H and DN-H) in a co-location scenario. Our new clustering service (CS) interacts with all three systems. The arrows represent information flow. C_i = Container i; AP = AutoPilot.

5.1 Background

YARN [36] comprises a global Resource Manager (RM) running on a dedicated server, a Node Manager (NM) per server, and a per-job Application Master (AM) running on one of the servers. The RM arbitrates the use of resources (currently, cores and memory) across the cluster. The (primary) RM is often backed up by a secondary RM in case of failure. Each AM requests containers from the RM for running the tasks of its job. Each container request specifies the desired core and memory allocations for it, and optionally a "node label". The RM selects a destination server for each container that has the requested resources available and the same label. The AM decides which tasks it should execute in each container. The AM also tracks the tasks' execution, sequencing them appropriately, and re-starting any killed tasks. Each NM creates containers and reports the amount of locally available resources to the RM in periodic "heartbeats". The NM kills any container that tries to utilize more memory than its allocation.

Tez [29] is a popular framework upon which MapReduce, Hive, Pig, and other applications can be built. Tez provides an AM that executes complex jobs as DAGs.

HDFS [11] comprises a global Name Node (NN) running on a dedicated server, and a Data Node (DN) per server. The NN manages the namespace and the mapping of file blocks to DNs. The (primary) NN is typically backed up by a secondary NN. By default, the NN replicates each block (256 MBytes) three times: one replica in the server that created the block, one in another server of the same rack, and one in a remote rack. Upon a block access, the NN informs the client about the servers that store the block's replicas. The client then contacts the DN on any of these servers directly to complete the access. The DNs heartbeat to the NN; after a few missing heartbeats from a DN, the NN starts to re-create the corresponding replicas in other servers without overloading the network (30 blocks/hour/server).

System	Main extensions
YARN	Report primary tenant utilization to the RM Kill containers due to primary tenant needs Maintain resource reserve for primary tenant Probabilistically balance load
Tez	Leverage information on the observed job lengths Estimate max concurrent resource requirements Track primary tenant utilization patterns Schedule tasks on servers unlikely to kill them Schedule tasks on servers with similar primaries
HDFS	Track primary tenant utilization, deny accesses Report primary tenant status to the NN Exclude busy servers from info given to clients Track primary disk reimaging, peak utilizations Place replicas at servers with diverse patterns
General	Create dedicated environment for main components

Table 1: Our main extensions to YARN, Tez, and HDFS.

5.2 Implementation guidelines

We first must modify the systems to become aware of the primary tenants and their priority over the servers' resources. Because of this priority, we must ensure that the key components of these systems (RMs and NNs) do not share their servers with any primary tenants. Second, we want to integrate our history-based task scheduling and data placement algorithms into these systems.

Figure 9 overviews our systems. The arrows in the figure represent information flow. Each shared server receives one instance of our systems; other workloads are considered primary tenants. Table 1 overviews our main extensions. The next sections describe our systems.

5.3 YARN-H and Tez-H

Design goals: (G1) ensure that the primary tenant always gets the cores and memory it desires; (G2) ensure that there is always a reserve of resources for the primary tenant to spike into; and (G3) schedule the tasks on servers where they are less likely to be killed due to the resource needs of the corresponding primary tenants.

Primary tenant awareness. We implement goals G1 and G2 in YARN-H by modifying the NM to (1) track the primary tenant's core and memory *utilizations*; (2) round them up to the next integer number of cores and the next integer MB of memory; and (3) report the sum of these rounded values and the secondary tenants' core and memory *allocations* in its heartbeat to RM-H. If NM-H detects that there is no longer enough reserved resources, it replenishes the reserve back to the pre-defined amount by killing enough containers from youngest to oldest.

Smart task scheduling. We implement goal G3 by implementing a service that performs our clustering algorithm, and integrating our class selection algorithm into Tez-H. We described both algorithms in Section 4.1.

Tez-H requests the estimated maximum number of concurrent containers from RM-H. When Tez-H selects one class, the request names the node label for the class. When Tez-H selects multiple classes, it uses a disjunction expression naming the labels. RM-H schedules a container to a heartbeating server of the correct class with a probability proportional to the server's available resources. If Tez-H does not name a label, RM-H selects destination servers using its default policy.

Overheads. Our modifications introduce negligible overheads. For primary tenant awareness, we add a few system calls to the NM to get the resource utilizations, perform a few arithmetic operations, and piggyback the results to RM-H using the existing heartbeat. The clustering service works off the critical path of job execution, computes headrooms using a few arithmetic operations, and imposes very little load on RM-H. In comparison to its querying of RM-H once per minute, *every* server heartbeats to RM-H *every 3 seconds*. Tez-H requires a single interaction with the clustering service per job.

5.4 HDFS-H

Design goals: (G1) ensure that we never use more space at a server than allowed by its primary tenant; (G2) ensure that HDFS-H data accesses do not interfere with the primary tenant when it needs the server resources; and (G3) place the replicas of each block so that it will be as durable and available as possible, given the resource usage of the primary tenants and how they are managed.

Note that full data durability cannot be *guaranteed* when using harvested storage. For example, service engineers or the management system may reimage a large number of disks at the same time, destroying multiple replicas of a block. Obviously, one can increase durability by using more replicas. We explore this in Section 6.

Primary tenant awareness. For goal G1, we use an existing mechanism in HDFS: the primary tenants declare how much storage HDFS-H can use in each server.

Implementing goal G2 is more difficult. To make our changes seamless to clients, we modify the DN to deny data accesses when its replica is unavailable (*i.e.*, when allowing the access would consume some of the resource reserve), causing the client to try another replica. (If all replicas of a desired block are busy, the block becomes unavailable and Tez will fail the corresponding task.) In addition, DN-H reports being "busy" or available to NN-H in its heartbeats. If DN-H says that it is busy, NN-H stops listing it as a potential source for replicas (and stops using it as a destination for new replicas as well). When the CPU utilization goes below the reserve threshold, NN-H will again list the server as a source for replicas (and use it as a destination for new ones).

Smart replica placement. For goal G3, we integrate our replica placement algorithm (Section 4.2) into NN-H.

Overheads. Our extensions to HDFS impose negligible overheads. For primary tenant awareness, we add a few system calls to the DN to get the primary tenant CPU utilization, and piggyback the results to NN-H in the heartbeat. Denying a request under heavy load adds two network transfers, but this overhead is minimal compared to that of disk accesses. For smart replica placement, our modifications add the clustering algorithm to the NN, and the extra communication needed for it to receive the algorithm inputs. The clustering and data structure updates happen in the background, off the critical path.

6 Evaluation

6.1 Methodology

Experimental testbed. Our testbed is a 102-server setup, where each server has 12 cores and 32GB of memory. We reserve 4 cores (33%) and 10GB (31%) of memory for primary tenants to burst into based on empirical measurements of interference. (Recall that performance isolation technology at each server would enable smaller resource reserves.) To mimic realistic primary tenants, each server runs a copy of the Apache Lucene search engine [26], and uses more threads (up to 12) with higher load. We direct traffic to the servers to reproduce the CPU utilization of 21 primary tenants (13 periodic, 3 constant, and 5 unpredictable) from datacenter DC-9. We also reproduce the disk reimaging statistics of these primary tenants. For the batch workloads, we run 52 different Hive [33] queries (which translate into DAGs of relational processing tasks) from the TPC-DS benchmark [34]. We assume Poisson inter-arrival times (mean 300 seconds) for the queries.

We use multiple baselines. When studying scheduling, the first baseline is stock YARN and Tez. We call it *"YARN-Stock"*. The second baseline combines primary-tenant-aware YARN with stock Tez, but does not implement smart task scheduling. We call it *"YARN-PT"*.

We call our full system *"YARN-H/Tez-H"*. Given the workload above, we set the thresholds for distinguishing task length types to 173 and 433 seconds. Jobs shorter than 173 seconds are short, and longer than 433 seconds are long. These values produce resource requirements for the jobs of each type that roughly correspond to the amount of available capacity in the preferred primary tenant class for the type. We use HDFS-Stock with YARN-Stock, and HDFS-PT with the other YARN versions. The latter combination isolates the impact of primary tenant awareness in YARN from that in HDFS.

When studying data placement and access, the first baseline is *"HDFS-Stock"*, *i.e.* stock HDFS unaware of primary tenants. The second baseline is *"HDFS-PT"*, which brings primary tenant awareness to data accesses but does not implement smart data placement. We call our full system *"HDFS-H"*. We use YARN and Tez with HDFS-Stock, and YARN-PT and Tez with the other HDFS versions. Again, we seek to isolate the impact of primary tenant awareness in HDFS and YARN.

Simulator. Because we cannot experiment with entire datacenters and need to capture long-term behaviors (*e.g.*, months to years), we also built a simulator that reproduces the CPU utilization and reimaging behavior of all the primary tenants (thousands of servers) in the datacenters we study. We simulate servers of the same size and resource reserve as in our real experiments. To study a spectrum of utilizations, we also experiment with higher and lower traffic levels, each time multiplying the CPU utilization time series by a constant factor and saturating at 100%. Because of the inaccuracy introduced by saturation, we also study a method in which we scale the CPU utilizations using n^{th}-root functions (*e.g.*, square root, cube root). These functions make the higher utilizations change less than the lower ones when we scale them, reducing the chance of saturations.

When studying task scheduling and data availability, we simulate each datacenter for one month. When studying data durability, we simulate each datacenter for one year. We use the same set of Hive queries to drive our simulator, but multiply their lengths and container usage by a scaling factor to generate enough load for our large datacenters (many thousands of servers) while limiting the simulation time.

In the simulator, we use the same code that implements clustering, task scheduling, and data placement in our real systems. The simulator also reproduces key behaviors from the real systems, *e.g.* it reconstructs lost replicas at the same rate as our real HDFS systems. However, it does not model the primary tenants' response times. We compare our systems to the second baseline (YARN-PT) in task scheduling, and the first baseline (HDFS-Stock) in data placement and access.

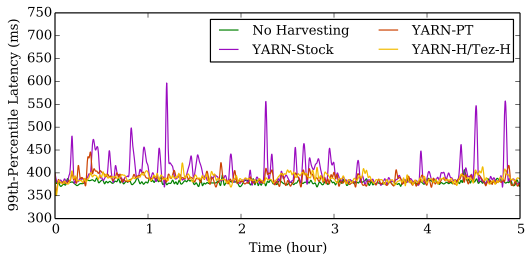


Figure 10: Primary tenant's tail latency in the real testbed for versions of YARN and Tez.

6.2 Performance microbenchmarks

The most expensive operations in our systems are the clustering and class selection in task scheduling and data placement. For task scheduling, clustering takes on average 2 minutes for the primary tenants of DC-9, when running single-threaded. (Recall that this clustering happens in the clustering service once per day, off the critical scheduling path.) The clustering produces 23 classes (13 periodic, 5 constant, and 5 unpredictable) for DC-9. For this datacenter, class selection takes less than 1 msec on average. For data placement, clustering and class selection take on average 2.55 msecs per new block (0.81 msecs in HDFS-Stock) for DC-9. (Clustering here can be done off the critical data placement path as well.)

6.3 Experimental results

Task scheduling comparisons. We start by investigating the impact of harvesting spare compute cycles on the performance of the primary tenant. Figure 10 shows the average of the servers' 99th-percentile response times (in ms) every minute during a five-hour experiment. The curve labeled "No Harvesting" depicts the tail latencies when we run Lucene in isolation. The other curves depict the Lucene tail latencies under different systems, when TPC-DS jobs harvest spare cycles across the cluster. The figure shows that YARN-Stock hurts tail latency significantly, as it disregards the primary tenant. In contrast, YARN-PT keeps tail latencies significantly lower and more consistent. The main reason is that YARN-PT actually kills tasks to ensure that the primary tenant's load can burst up without a latency penalty. Finally, YARN-H/Tez-H exhibits tail latencies that nearly match those of the No-Harvesting execution. The maximum tail latency difference is only 44 ms, which is commensurate with the amount of variance in the No-Harvesting execution (average tail latencies ranging from 369 to 406 ms). The improved tail latencies come from the more balanced utilization of the cluster capacity in YARN-H.

Another key characteristic of YARN-H/Tez-H is its smart scheduling of tasks to servers where they are less

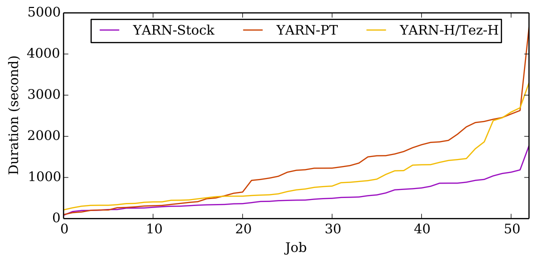


Figure 11: Secondary tenants' run times in the real testbed for versions of YARN and Tez.

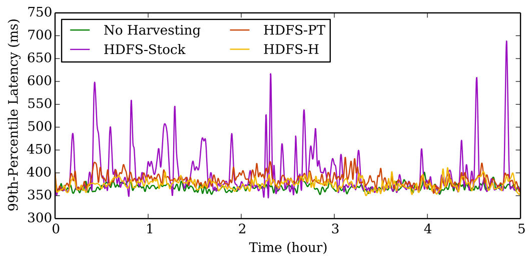


Figure 12: Primary tenant's tail latency in the real testbed for versions of HDFS.

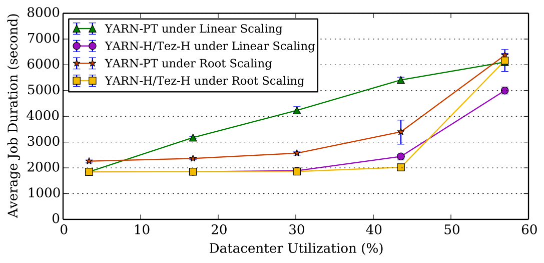


Figure 13: Secondary tenants' run time improvements in DC-9 under YARN-H/Tez-H for root and linear scalings.

likely to be killed. Figure 11 shows the execution times of all jobs in TPC-DS for YARN-Stock, YARN-PT, and YARN-H/Tez-H. As one would expect, YARN-Stock exhibits the lowest execution times. Unfortunately, this performance comes at the cost of ruining that of the primary tenant, which is unacceptable. Because YARN-PT must kill (and re-run) tasks when the primary tenant's load bursts, it exhibits substantially higher execution times, 1181 seconds on average. YARN-H/Tez-H lowers these times significantly to 938 seconds on average.

In these experiments, YARN-H/Tez-H improves the average CPU utilization from 33% to 54%, which is a significant improvement given that we reserve 33% of the CPU for primary tenant bursts. The utilization improvement depends on the utilization of the primary tenants (the lower their utilization, the more resources we can harvest), the resource demand coming from secondary tenants (the higher the demand, the more tasks we can schedule), and the resource reserve (the smaller the reserve, the more resources we can harvest).

Overall, these results clearly show that YARN-H/Tez-H is capable of both protecting primary tenant performance and increasing the performance of batch jobs.

Data placement and access comparisons. We now investigate whether HDFS-H is able to protect the performance of the primary tenant and provide higher data availability than its counterparts. Figure 12 depicts the average of the servers' 99th-percentile response times (in ms) every minute during another five-hour experiment. As expected, the figure shows that HDFS-Stock degrades tail latency significantly. HDFS-PT and HDFS-H reduce the degradation to at most 47 ms. The reason is that these versions avoid accessing/creating data at busy servers. However, HDFS-PT actually led to 47 failed accesses, *i.e.* these blocks could not be accessed as all of their replicas were busy. By using our smart data placement algorithm, HDFS-H eliminated all failed accesses.

6.4 Simulation results

Task scheduling comparisons. We start our simulation study by considering the full spectrum of CPU utilizations, assuming the size and behavior of our real production datacenters. Recall that we use two methods to scale utilizations (up and down) from the real utilizations: linear and root scalings. To isolate the benefit of our use of historical primary tenant utilizations, we compare YARN-H/Tez-H to YARN-PT. Figure 13 depicts the average batch job execution time in DC-9 under both systems and scalings, as a function of utilization. Each point along the curves shows the average of five runs, whereas the intervals range from the minimum average to the maximum average across the runs. As one would expect, high utilization causes higher queuing delays and longer execution times. (Recall that we reserve 33% of the resources for primary tenants to burst into, so queues are already long when we approach 60% utilization.) However, YARN-PT under linear scaling behaves differently; the average execution times start to increase significantly at lower utilizations. The reason is that linear scaling produces greater temporal variation in the CPU utilizations of each primary tenant than root scaling. Higher utilization variation means that YARN-PT is more likely to have to kill tasks, as it does not know the historical utilization patterns of the primary tenants. For example, at 45% utilization, YARN-PT under linear scaling kills 4× more tasks than the other system-scaling combinations.

Because YARN-H/Tez-H uses our clustering and smart task scheduling, it improves job performance significantly across most of the utilization spectrum. Under linear scaling, the average execution time reduction

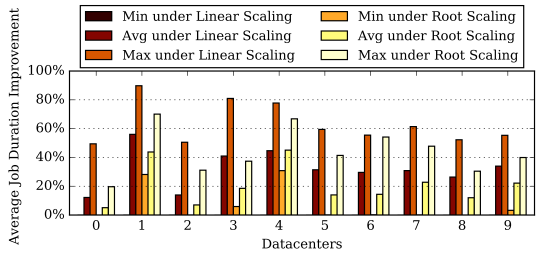


Figure 14: Secondary tenants' run time improvements from YARN-H/Tez-H for root and linear scalings.

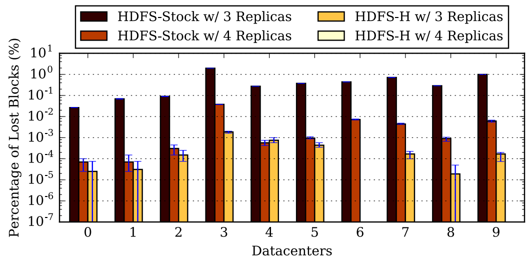


Figure 15: Lost blocks for two replication levels.

ranges from 0% to 55%, whereas under root scaling it ranges between 3% and 41%. The YARN-H/Tez-H advantage is larger under linear scaling, since the utilization pattern of each primary tenant varies more over time.

To see the impact of primary tenants with different characteristics than in DC-9, Figure 14 depicts the minimum, average, and maximum job execution time improvements from YARN-H/Tez-H across the utilization spectrum for each datacenter (five runs for each utilization level). The average improvements range from 12% to 56% under linear scaling, and 5% to 45% under root scaling. The lowest average improvements are for DC-0 and DC-2, which exhibit the least amount of primary tenant utilization variation over time. At the other extreme, the largest average improvements come for DC-1 and DC-4, as many of their primary tenants exhibit significant temporal utilization variations. The largest maximum improvements (~90% and ~70% under linear and root scaling, respectively) also come from these two datacenters, regardless of scaling type.

Data placement and access comparisons. We now consider the data durability in HDFS-H. Figure 15 shows the percentage of lost blocks under two replication levels (three and four replicas per block), as we simulate one year of reimages and 4M blocks. Each bar depicts the average of five runs, and the intervals range from the minimum to the maximum percent data loss in those simulations. The missing bars mean that there is no data loss in any of the corresponding five simulations. Note that a single lost block represents a 10^{-5} ($< 100 \times 1/4M$) percentage of lost blocks, *i.e.* 6 nines of durability.

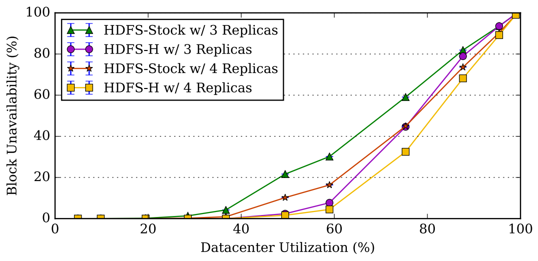


Figure 16: Failed accesses under linear scaling.

The figure shows that HDFS-H reduces data loss more than two orders of magnitude under three-way replication, compared to HDFS-Stock. Moreover, for one of the datacenters, HDFS-H eliminates all data loss under three-way replication. The maximum number of losses of HDFS-H in any datacenter was only 81 blocks (DC-3). Under four-way replication, HDFS-H completely eliminates data loss for all datacenters, whereas HDFS-Stock still exhibits losses across the board. These results show that our data placement algorithm provides significant improvements in durability, despite the harvested nature of the disk space and the relatively high reimage rate for many primary tenants. In fact, the losses with HDFS-H and three-way replication are lower than those with HDFS-Stock and four-way replication for all but one datacenter; *i.e.* our algorithm almost always achieves higher durability at a lower space overhead than HDFS-Stock.

Our data availability results are also positive. Figure 16 depicts the percentage of failed accesses under the two replication levels and linear scaling, as a function of the average utilization. The figure includes range bars from five runs, but they are all too small to see. The figure shows that HDFS-H exhibits no data unavailability up to higher utilizations (~40%) than HDFS-Stock, and low unavailability for even higher utilization (50%), under both replication levels. At 50% utilization, HDFS-Stock already exhibits relatively high unavailability under both replication levels. Around 66% utilization, unavailability starts to increase faster (accesses cannot proceed if CPU utilization is higher than 66%). More interestingly, our smart data placement under three-way replication achieves lower unavailability than HDFS-Stock under four-way replication below 75% utilization. The trends are similar under root scaling, except that HDFS-H exhibits no unavailability up to a higher utilization (50%) than with linear scaling. Regardless of the scaling type, HDFS-H can achieve higher availability at a lower space overhead than HDFS-Stock for most utilizations.

7 Experiences in Production

As a first rollout stage, we deployed HDFS-H to a production cluster with thousands of servers eleven months ago. Since then, we have been enabling/adding features as our deployment grows. For example, we extended the set of placement constraints beyond environments to include machine functions and physical racks. In addition, we initially configured the system to treat the replica placement constraints as "soft", *e.g.* the placement algorithm would allow multiple replicas in the same environment, to prevent the block creation from failing when the available space was becoming scarce. This initial decision promoted space utilization over diversity. Section 4.2 discusses this tradeoff.

Since its production deployment, our system has eliminated all data losses, except for a small number of losses due to corner-case bugs or promoting space over diversity. Due to the latter losses, we started promoting diversity over space utilization more than nine months ago. Since then, we have not lost blocks. For comparison, when the stock HDFS policy was activated by mistake in this cluster for just three days during this period, dozens of blocks were lost.

We also deployed YARN-H's primary tenant awareness code to production fourteen months ago, and have not experienced any issues with it (other than needing to fix a few small bugs). We are now productizing our scheduling algorithm and will deploy it to production.

In the process of devising, productizing, deploying, and operating our systems, we learned many lessons.

1. Even well-tested open-source systems require additional hardening in production. We had to create watchdogs that monitor key components of our systems to detect unavailability and failures. Because of the nontrivial probability of concurrent failures, we increased the number of RMs and NNs to four instead of two. Finally, we introduced extensive telemetry to simplify debugging and operation. For example, we collect extensive information about HDFS-H blocks to estimate its placement quality.

2. Synchronous operations and unavailability. Synchronous operations are inadequate when resources or other systems become unavailable. For example, our production deployments interact with a performance isolation manager (similar to [24]). This interaction was unexpectedly harmful to HDFS-H. The reason is that the manager throttles the secondary tenants' disk activity when the primary tenant performs substantial disk I/O. This caused the DN heartbeats on these servers to stop flowing, as the heartbeat thread does synchronous I/O to get the status of modified blocks and free space. As a result, the NN started a replication storm for data that it thought was lost. We then changed the heartbeat thread to become asynchronous and report the status that it most recently found.

3. Data durability is king. As we mention above, our initial HDFS-H deployment favored space over diversity, which caused blocks to be lost and the affected users to become quite exercised. By default, we now monitor the quality of placements and stop consuming more space when diversity becomes low. To recover some space, we still favor space usage over diversity for those files that do not have strict durability requirements.

4. Complexity is your enemy. As others have suggested [6], simplicity, modularity, and maintainability are highly valued in large production systems, especially as engineering teams change and systems evolve. For example, our initial task scheduling technique was more complex than described in Section 4.1. We had to simplify it, while retaining most of the expected gains.

5. Scaling resource harvesting to massive datacenters requires additional infrastructure. Stock YARN and HDFS are typically used in relatively small clusters (less than 4k servers), due to their centralized structure and the need to process heartbeats from all servers. Our goal is to deploy our systems to much larger installations, so we are now in the process of creating an implementation of HDFS-H that federates multiple smaller clusters and automatically moves files/folders across them based on primary tenant behaviors, and our algorithm's ability to provide high data availability and durability.

6. Contributing to the open-source community. Though our techniques are general, some of the code we introduced in our systems was tied to our deployments. This posed challenges when contributing changes to and staying in-sync with their open-source versions. For example, some of the YARN-H primary tenant awareness changes we made to Hadoop version 2.6 were difficult to port to version 2.7. Based on this experience, we refactored our code to isolate the most basic and general functionality, which we could then contribute back; some of these changes will appear in version 2.8.

8 Conclusion

In this paper, we first characterized all servers of ten large-scale datacenters. Then, we introduced techniques and systems that effectively harvest spare compute cycles and storage space from datacenters for batch workloads. Our systems embody knowledge of the existing primary workloads, and leverage historical utilization and management information about them. Our results from an experimental testbed and from simulations of the ten datacenters showed that our systems eliminate data loss and unavailability in many scenarios, while protecting primary workloads and significantly improving batch job performance. Based on these results, we conclude that our systems in general, and our task scheduling and data placement policies in particular, should enable datacenter operators to increase utilization and reduce TCO.

Acknowledgments

We thank our shepherd, Michael Stumm, for his help in improving the paper and his patience with us. We also thank Paulo Tomita, Sekhar Pasupuleti, Robert Grandl, and Srikanth Kandula for their help with our experimental setup. We are indebted to Karthik Kambatla for his help with open-sourcing some of our changes to YARN. We are also indebted to Sriram Rao, Carlo Curino, Chris Douglas, Vivek Narasayya, Manoj Syamala, Sameh Elnikety, Thomas F. Wenisch, and Willy Zwaenepoel for our many discussions about this work and their comments to our paper. Finally, we thank Gaurav Sareen and Eric Boyd for their support of this project.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation*, 2016.

[2] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *Proceedings of the VLDB Endowment*, 2013.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2013.

[5] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 2008.

[7] R. B. Clay, Z. Shen, and X. Ma. Accelerating Batch Analytics With Residual Resources From Interactive Clouds. In *Proceedings of the 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2013.

[8] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[10] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[11] A. Foundation. HDFS Architecture Guide, 2008.

[12] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[13] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[14] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM SIGCOMM Conference*, 2014.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[17] M. Isard. Autopilot: Automatic Data Center Management. *SIGOPS Operating Systems Review*, 2007.

[18] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[19] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[20] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[21] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems*, 2014.

[22] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[23] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[24] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[26] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[27] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[28] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 2010.

[29] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[30] B. Sharma, T. Wood, and C. R. Das. HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, 2013.

[31] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*, 2012.

[32] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2009.

[34] Transaction Processing Performance Council. TPC Benchmarks.

[35] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling Using System-Generated Predictions Rather Than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, 2015.

[38] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[39] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[41] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing Interference and Maximizing Progress for Hadoop Virtual Machines. *SIGMETRICS Performance Evaluation Review*, 2015.

[42] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[43] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.