RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation

Andrew Newell, Dimitrios Skarlatos[‡*], Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty[†], Apurva Samudra[†], Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, Chunqiang Tang

{newella, dskarlat, jfan5, pavanka, maximk, mpundir, yirui, mingjunzhang, lyl, linhle, brendond, asamudra, prashasti, jamesk, ikabiljo, dmitrs, andrerod, sdmich, benjchristensen, kaushikv, tang}@fb.com

Facebook Inc. [‡]Carnegie Mellon University

Abstract

Capacity reservation is a common offering in public clouds and on-premise infrastructure. However, no prior work provides capacity reservation with SLO guarantees that takes into account random and correlated hardware failures, datacenter maintenance, and heterogeneous hardware. In this paper, we describe how Facebook's region-scale Resource Allowance System (RAS) addresses these issues and provides guaranteed capacity. RAS uses a capacity abstraction called *reservation* to represent a set of servers dynamically assigned to a logical cluster. We take a two-level approach to scale resource allocation to all datacenters in a region, where a mixed-integer-programming solver continuously optimizes server-to-reservation assignments off the critical path, and a traditional container allocator does real-time placement of containers on servers in a reservation. As a relatively new component of Facebook's 10-year old cluster manager Twine, RAS has been running in production for almost two years, continuously optimizing the allocation of millions of servers to thousands of reservations. We describe the design of RAS and share our experience of deploying it at scale.

CCS Concepts

• **Computer systems organization → Distributed architectures**; *Maintainability and maintenance*; • **Software and its engineering**;

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.

https://doi.org/10.1145/3477132.3483578

Keywords

Datacenter, Resource Allocation, Capacity Management, Container Orchestration

ACM Reference Format:

Andrew Newell, Dimitrios Skarlatos[‡*], Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty[†], Apurva Samudra[†], Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, Chunqiang Tang. 2021. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483578

1 Introduction

Cluster managers [28, 31, 45] place containers or virtual machines on servers and manage their lifecycle. In the past decade, a significant amount of research has focused on developing efficient resource allocation solutions in datacenters. Numerous approaches have been adopted by public clouds [27], open-source systems such as Kubernetes [31], and proprietary systems such as Google's Borg [45], Facebook's Twine [39], and Microsoft's Protean [27].

Capacity reservation is a common offering in public clouds [3, 6, 23, 34, 35], which *"ensures the peace of mind that you have capacity availability when you need it during critical events, such as disaster recovery or unexpected workload spikes [35]."* However, there is scant literature on how it works, especially, how to provide *guaranteed capacity* despite large-scale failures in datacenters. In this paper, we describe how we solve this problem for our on-premise infrastructure, whose size is comparable to that of the largest public clouds.

There are several challenges in providing guaranteed capacity. First, it needs to account for independent and correlated failures [44] across various scopes including server,

rack, datacenter, network switch, power row, and cooling systems. Naively increasing the buffer capacity to handle all potential failures would be prohibitively expensive.

Second, the cluster manager needs to uphold a capacity guarantee despite constant infrastructure lifecycle events such as OS kernel upgrades, physical maintenance at various scopes, hardware refresh, and datacenter turn-ups. Each of these can cause different magnitudes of server capacity loss and the cluster manager needs to acquire replacement servers in a timely manner.

Third, a cluster manager should provide capacity that meets the constraints of its workloads and also considers hardware heterogeneity. For instance, some workloads might gain a significant performance boost on a particular CPU generation, others might require affinity with storage systems and databases to meet latency requirements, while yet others might need all their servers situated within a single datacenter so that they would not overwhelm cross-datacenter network links.

Finally, the increased scale of modern infrastructure further exacerbates these challenges as there exists an inherent trade-off between the decision quality and the speed of resource allocation. If the cluster manager prioritizes meeting an allocation-speed service level objective (SLO) by compromising the allocation spread across fault domains, a single large-scale correlated failure might render a significant fraction of capacity unavailable.

It is challenging to allocate a set of servers that simultaneously satisfy guaranteed capacity requirements, support datacenter lifecycle operations, and meet complex workload constraints, while still achieving low-latency container placement. To that end, we propose breaking resource allocation into a two-level problem: 1) assigning servers to logical clusters and 2) assigning containers to servers in a cluster.

While the latter has been widely explored [9, 12, 16, 22, 27–31, 36, 43, 45–47], assigning servers to clusters has received far less attention in the past. These two levels compliment each other. Some problems that are hard or impossible to be solved by the latter can be more easily solved by the former. This paper presents our solution for server assignment, which runs off the critical path of container placement.

1.1 Prior Solutions

The most common approach of assigning servers to clusters is based on static scopes. For example, all servers in a datacenter may belong to one cluster. Servers may be added to or removed from a cluster, but often these changes are manually initiated. A benefit of this approach is that the static cluster membership reduces the candidate servers to be evaluated on the critical path of container placement. Hence, it enables new containers to be quickly deployed within a few seconds by using servers already in the cluster. Unfortunately, this benefit comes with some drawbacks. First, with static assignment of servers to clusters, some clusters may run out of capacity and cannot sustain higher loads, while others are underutilized. Second, server allocation may be suboptimal due to variation in power and network consumption of workloads and their different hardware requirements such as certain CPUs, flash storage, and memory capacity. Finally, service owners have to individually prepare for datacenter-scale failures by themselves.

Our previous approach at Facebook, presented in Twine [39], is to forego the boundary of physical clusters and datacenters, and use a shared mega server pool that comprises all servers from datacenters in a geographical region connected by a low-latency network. Conceptually, Twine organizes servers into logical clusters called *entitlements*. When a new container needs to be started but cannot fit on any existing server in an entitlement, a free server is greedily acquired from a shared region-level free-server pool and added to the entitlement to host the new container. When the last container running on a server is decommissioned, the server is returned to the shared free-server pool. On one hand, this approach has the benefit that a single server pool eliminates server capacity stranded in many smaller physical clusters. On the other hand, it puts a whole region's server-to-entitlement assignment on the critical path of container placement. As a result, previously we had to adopt simple heuristics to make quick server-assignment decisions, which led to sub-optimal server assignment and could not provide guaranteed capacity in the event of correlated failures.

Overall, both approaches, though highly practical, have their limitations. Ideally, a cluster manager should combine their benefits without their drawbacks.

1.2 Our Solution: Continuous Server Reassignment

In this paper, we present Twine's new server-allocation component, called *Resource Allowance System (RAS)*. RAS uses a capacity abstraction called *reservation* to represent a set of servers dynamically assigned to a logical cluster. A reservation provides to its workloads a certain amount of guaranteed capacity that takes into account random and correlated failures, datacenter maintenance events, heterogeneous hardware resources, and compound workload requirements and characteristics.

RAS adopts a two-level approach that first assigns servers to reservations off the critical path and then allocates containers to servers within each reservation. Through the two-level approach, server-assignment constraints are removed from the latency-sensitive container-placement process, and are evaluated at the reservation-creation time and maintained

continuously. Moreover, the two-level approach enables multiple container allocators to run independently for better scalability, by treating each reservation as a separate logical cluster. Finally, each reservation embeds the buffer capacity needed for handling large-scale failures and maintenance, removing server-to-reservation assignments from the critical path of these operations.

RAS exploits the inherent computational slack to deploy a mixed-integer-programming (MIP) solver to optimize server-to-reservation assignments at a regional level, continuously, every few tens of minutes. The generality of MIP allows RAS to incorporate a broad set of factors into the optimization problem, including capacity requirements, server availability, network, datacenter topology, as well as random and correlated failure buffer constraints. Furthermore, RAS introduces stability and optimality objectives that minimize movement of servers and optimize server spread across fault domains of various scopes.

To cater to the increasingly heterogeneous hardware resources within datacenters, RAS abstracts the amount of heterogeneous hardware through relative resource units (*RRUs*). RRUs allow RAS to form reservations based on equivalent heterogeneous resources that take into account a workload's relative performance on different CPU, memory, flash, GPU, and ASIC configurations.

Overall, RAS has several advantages over prior solutions. First, RAS avoids the drawbacks of statically-scoped clusters, including capacity stranded in clusters and the burden for service owners to individually prepare for large-scale failures. RAS resolves them by dynamically adjusting servers assigned to reservations to best match workload characteristics and underlying infrastructure changes, and by embedding and optimizing failure and maintenance buffers as part of reservations. Second, RAS avoids the drawbacks of Twine's previous approach of performing server assignments on the critical path of container placement, by allocating a reservation's full capacity ahead of time. Hence, container placement can immediately use a free server already in the reservation. Finally, RAS offers to users the simple abstraction of workloads running on a reservation that provides guaranteed capacity and supports stacking. Underneath, RAS takes care of hierarchical fault domains in a region, random and correlated failures, datacenter maintenance, heterogeneous hardware, network affinity, power consumption, and myriad of other datacenter constraints and realities.

As a relatively new component of Twine, RAS has been running in production for almost two years. RAS achieves near fully allocated regions, where close to 94% of servers are allocated for workloads as guaranteed capacity, 2% are allocated for random failures, and the remaining 4% are allocated for large-scale correlated failures.

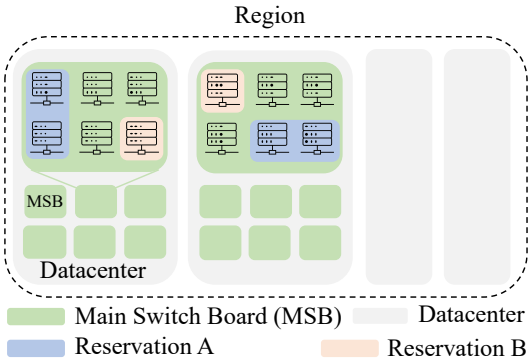

Figure 1: Regional Datacenter Topology.

The contributions of this paper are as follows:

- RAS introduces the concept of reservation, i.e., a guaranteed amount of server capacity that functions as a logical cluster and abstracts away the complexity of datacenters, hardware heterogeneity, and workload characteristics.
- RAS presents a two-level approach that decouples server-to-reservation assignments from container placement.
- RAS formulates server assignments as a MIP problem. We describe our techniques for scaling our MIP solver to allocate up to a million servers in a whole region.
- RAS optimizes server assignments based on a broad set of factors such as capacity requirements, datacenter topology, random and correlated failures, server movement constraints, and fault-domain spread.

2 Resource Management Realities

Efficiently providing guaranteed capacity within a region poses significant challenges to resource allocation due to the capacity scale, intricacies of datacenters, and diverse workload characteristics.

2.1 Region Layout

Facebook's infrastructure is geo-distributed across multiple regions around the globe. Figure 1 shows a high-level overview of a region's layout. A region consists of multiple datacenters connected through high-bandwidth networking. At Facebook, each datecenter is composed of several fault domains defined by the power main switch board (MSB), which are isolated power and network domains that can fail independently. Each MSB consists of thousands of servers.

The network latency within a region is less than a millisecond. This enables resource management to aggregate resources across datacenters in a region beyond the traditional concept of a physical cluster, which is historically

constrained within a datacenter. Still, the network bandwidth across datacenters in a region is only a fraction of the bandwidth between racks in the same datacenter. As a result, server assignments need to consider cross-datacenter bandwidth. Such constraints are important for network-intensive workloads such as distributed machine learning training that can swiftly saturate the available bandwidth.

2.2 Hardware Heterogeneity

Turn-up and decommission of servers combined with the increased hardware heterogeneity leads to vastly different hardware mixtures across MSBs. Figure 2 shows the hardware mixture across a set of representative MSBs within a region. The last bar shows the average mixture across all the MSBs of the region. Hardware resources are broken down into $< C_i - S_i >$ tuples where C represents a hardware category while S represents subtypes within each category. We divide hardware into subtypes only if there is a notable performance difference.

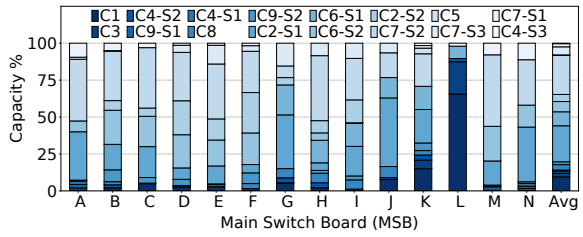

Figure 2: Hardware heterogeneity across MSBs.

Figure 2 shows that there are nine hardware types and a total of 12 subtypes. Furthermore, there is a large variation of the hardware mixture across MSBs. With the emergence of persistent memory and specialized hardware such as GPUs and accelerators for AI and video [19], datacenters are bound to exhibit even higher heterogeneity in the future. As a result, server assignment needs to take into account the hardware heterogeneity across MSBs and abstract away physical hardware characteristics to more uniform metrics.

2.3 Impact of Hardware Heterogeneity on Services

Services are commonly classified based on their resource requirements such as compute, memory, storage, and network. A major challenge of resource allocation is to automatically adapt to different types and generations of hardware available at hand. In order to classify how a service benefits from a given hardware generation, we define a metric called *Relative Value* that accounts for the important performance metrics of each service. To compute the Relative Value, we aggregate the average for each service across the entirety of its instances. We further use this metric to guide our capacity planning for each service.

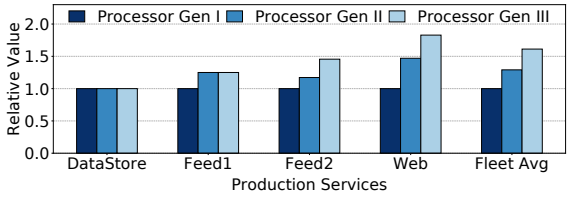

Figure 3: Relative value gained across Facebook' services across three processor generations.

Figure 3 shows how a set of four large services within Facebook gain values from different hardware generations. Each service is normalized individually to the first hardware generation. Some services gain significant values from new hardware generations. For example, Web gains 1.47x and 1.82x, by upgrading to the second and third hardware generation respectively. The fifth group shows the fleet average across all Facebook's services excluding the major four shown in the figure. The result indicates that in general processor generation upgrades lead to significant gains. However, some major services such as the DataStore do not see any gains. Others like Feed gain from one generation but not the next. Clearly, service intrinsics introduce important server assignment constraints and opportunities. In general, every workload scales slightly differently although similar workloads exhibit comparable scaling. Notably, it is important to provide a resource abstraction such that hardware-specific nuances do not burden individual service owners.

2.4 Diverse Capacity Requests

At Facebook we opted to split resource management between capacity and container requests. Capacity requests are usually initiated by engineers and capacity planners. The result is a collection of hardware resources that are then managed by the container scheduler. Requests are commonly in the order of a few thousand per day. Furthermore, they are highly diverse as they exhibit multiple degrees of variation across the number of requested resources and hardware types.

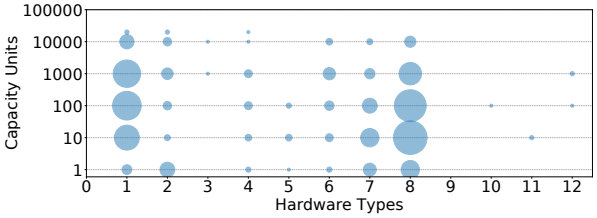

Figure 4: Requested capacity vs. the number of hardware types that can fulfill them.

Figure 4 shows the requested hardware capacity in normalized units across the possible hardware types that can

fulfill them. A capacity unit represents a physical server. We plot the y-axis on log-scale and aggregate the frequency of requests. Each circle represents the amount of servers of a given server type. The size of the circle shows the amount of requests across all services. For example, the circle in hardware type 8 and capacity unit 100, shows that capacity requests for about 100 servers that can be full-filled by 8 hardware types are one of the most common. As we can see from the figure, the requested capacity varies from one to more than 10,000 units. The majority of requests range from a few hundred to a few thousand units. A few requests are close to 30,000, which are from very large Web and Feed services deployed at Facebook.

Furthermore, capacity requests are spread over the possible hardware types that can fulfill them. Many requests can only be served by a single type of hardware which is usually the latest processor generation. A high number of requests can be served by eight hardware types. These requests are from services that require one or two processor generations and are agnostic to the remaining server configuration parameters such as main memory capacity. Finally, a small number of capacity requests can be fulfilled by 10-12 hardware types because some services are able to make good use of any hardware generation and system configuration.

Server assignment needs to handle the unceasing flow and diversity of capacity requests such that it minimizes server movements in the face of varying sizes of requested capacity and takes into account their hardware requirements.

2.5 Server Unavailability Events

Server unavailability is common within datacenters. Maintenance, hardware failures, power capping, kernel updates, and other operations lead to significant downtime of servers.

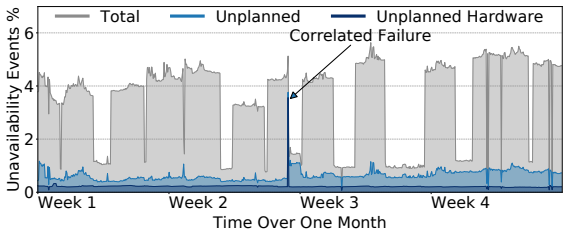

Figure 5: Server unavailability events over one month.

Figure 5 shows the number of planned and unplanned unavailability events active in a region over periods of 60 minutes normalized to the total capacity of the region during a month. In addition the figure shows the percentage of unplanned hardware failures. The remaining unplanned unavailability is due to software events. The duration of unavailability events can be minutes for planned maintenance, while hardware failures can last weeks. Planned unavailability events are controlled by infrastructure owners.

Combined planned and unplanned server unavailability can render more than 5% of the regional capacity unavailable. Typically, planned unavailability events, such as maintenance, account for the majority of capacity loss. Maintenance operations including maintenance of servers, network switches and power devices, kernel updates, and other operations lead to significant downtime of servers. However, unplanned events that usually account for less than 0.5% of the total capacity can spike to more than 3%.

Capacity unavailability due to random or correlated failures is a major challenge in datacenters. We consider failures at the server level or the Top-of-Rack (ToR) switch level as *random failures*. For example, at any time 0.1% of Facebook's fleet is classified as unavailable due to hardware repairs, a subcategory of random failures.

Correlated failures occur due to failures of common physical devices such as power breakers, network devices, or cooling fans. The largest fault domain inside a datacenter at Facebook is defined by the power main switch board (MSB), which may consist of thousands of servers. Within a region, we need to prepare for correlated failures as large as a whole MSB. Figure 5 shows an example of a correlated failure that caused a ≈4% capacity loss.

We have measured that roughly one MSB fails per month per region at our current capacity. Moreover, Facebook performs planned maintenance events at the granularity of MSB, which sets consistent expectations for handling planned maintenance events and correlated failures at the MSB scope or lower. Hence it is important to build resource management systems that provide guaranteed capacity to services in the presence of large-scale failures or maintenance.

3 RAS Design

RAS continuously optimizes server assignments and provides to services guaranteed capacity in the face of random and correlated failures, diverse capacity requests, heterogeneous hardware resources, and compound service requirements and characteristics.

3.1 Two-level Architecture

Our guiding principle is to decouple container placement from capacity allocation to enable region-wide optimizations for capacity allocation. Figure 6 shows an overview of our two-level architecture. The first level is RAS, which consists of two main components, the *Async Solver* and the *Online Mover*. The *Solver* uses mixed integer programming (MIP) to continuously reason about the entire region capacity and adapts resource binding to form dynamic capacity *reservations*. The *Mover* executes the *Solver*'s decision. The second

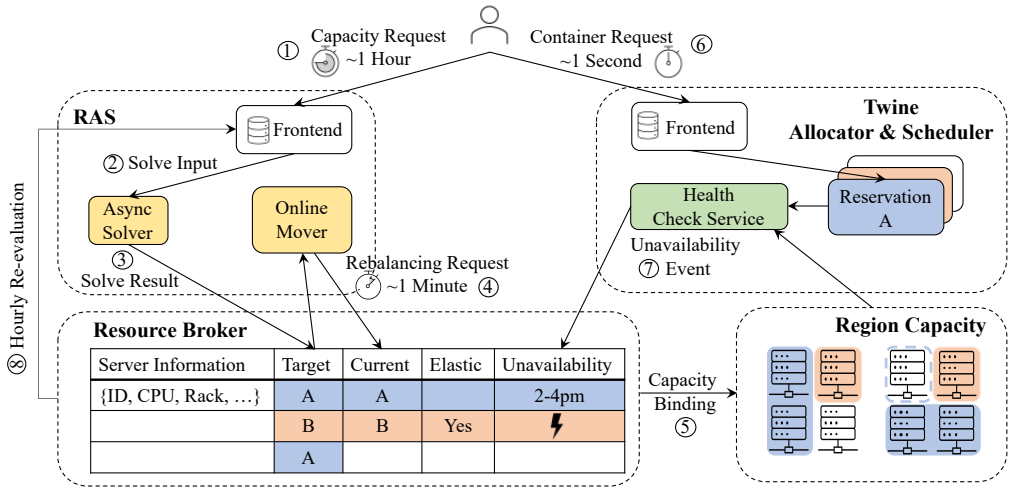

Figure 6: The two-level architecture with the components of RAS on the left and the Twine Allocator & Scheduler on the right. The Resource Broker on the bottom maintains server reservations of the regional capacity.

level is the *Twine Allocator & Scheduler*, which performs real time container placement atop each reservation and manages container lifecycle. The *Health Check Service* monitors all servers in the fleet. *Region Capacity* of hardware resources is virtualized through the *Resource Broker* that maintains server status and other reservation information.

A reservation is a logical cluster that represents a materialized amount of resources that the Twine Allocator can use. Each reservation is owned by a business unit to host their services. Containers in a reservation are stacked on servers assigned to the reservation. Reservations are characterized by the amount of resources, hardware types, placement policies, and operating-system (OS) configuration requirements.

RAS abstracts the amount of heterogeneous hardware resources through relative resource units (*RRUs*). A resource unit amount is defined for each server type reflecting throughput of the particular server type. Then, a total sum amount of resource units are requested, which reflect the aggregate throughput requirement. RRUs ensure services can define the aggregate hardware capacity they require. Additionally, smaller services can use a simple count-based approach where expressed units are equivalent among server types. RRUs avoid the pitfalls of uniform vCPUs as the minimum compute unit and can automatically and fairly adapt to different generations of hardware that co-exist in datacenters. RRUs can be defined for all types of heterogeneous resources such as accelerators, memory, storage, and network resources. RRUs effectively decouple capacity requests from the actual hardware allocated and allow RAS to fulfill each request by equivalent heterogeneous resources that together form a reservation.

Twine's Host Profile mechanism [39] allows each reservation to have custom OS configuration requirements (e.g., kernel version & settings) for its servers based on the needs of its workloads. A reservation's server placement policies are enforced by RAS based on the constraints of workloads. Most workloads within Facebook desire a wide spread across MSBs in a region, whereas some workloads require network and storage affinity.

3.2 Resource Management Flow

Facebook's resource management is separated between capacity requests and container requests. A service owner first performs a capacity request that defines her intent. The end result is hardware capacity that forms a reservation and is managed by the Twine Allocator.

Service owners create, modify, or delete capacity requests ① expressed as RRUs via the Capacity Portal. The state of all requests are stored in a database owned by RAS. The *Async Solver* performs continuous optimizations at each solve ②, and takes into consideration the latest request state and the complete hardware fleet of a region from the *Resource Broker*, a highly available storage that stores the *current* resource binding and hardware *unavailability* events. The async solver has a Service Level Objective (SLO) of completing each solve within one hour. The goal is to provide a solution that fulfills the resource requirements and meets the placement constraints such as sizing of failure buffers, network bandwidth limits, and fault-domain spread, while minimizing preemption to existing allocations, i.e., shuffling servers across reservations. Its output ③ is a mapping between servers and reservation IDs, signaling the binding intent, and is persisted back in the Resource Broker by updating the *target* field.

The design of handling all server assignments via a single solver under a one-hour SLO is ideal to optimize datacenter resources. However, despite efforts taken by service owners to forecast capacity demands, there will always be situations

where emergency capacity is needed immediately and cannot wait for one hour. Our solution is to offer a separate emergency path to quickly allocate capacity in these situations, and later the Async Solver can improve the potentially suboptimal allocation done by the emergency path.

The Online Mover is responsible for changing the ownership of a server following updates from RAS solve outputs ④. It also performs two efficiency optimizations: shared buffers and opportunistic capacity, as explained later. Each server move includes steps such as preempting containers off the current reservation and performing necessary host clean up and OS reconfiguration based on the target reservation specification. The outcome of this process is capacity binding that ties hardware resources to reservations ⑤.

After a capacity request is complete, the service owner can submit container requests ⑥. Twine's two-level architecture enables the Twine Allocator to provide swift response times of seconds on the critical path of container allocation. The Twine Allocator leverages the Resource Broker to get a list of candidate servers by referencing the reservation ID and perform further filtering and optimizations based on the constraints of the job at hand. Within a reservation, containers from different jobs can be placed on one server.

The *Health Check Service* monitors all servers in a region and updates the unavailability field in the Resource Broker ⑦. Both the Twine Allocator and Online Mover subscribe to unavailability events via call back. Unplanned events prompt the Online Mover to provide replacement servers within one minute, and then the Twine Allocator & Scheduler move containers to those servers. Replacement servers for planned events are already baked into the reservation and do not need the Online Mover to take actions on the critical path.

Finally, RAS re-evaluates its past server-assignment decisions every hour ⑧ in order to take into account capacity fluctuations and continuously optimize resource bindings.

3.3 Failure Buffers

At Facebook, we have observed that the p97.5 server unavailability over 90 days is ≈1% of the global fleet capacity. Correlated failures are observed at the frequency of ≈0.5% of power rows and ≈2% of MSBs impacted over a year. RAS manages random failures and correlated failures differently in order to minimize the buffer capacity.

3.3.1 Handling Random and Correlated Failures

All reservations use a common pool of *shared buffers* to handle random failures. Upon a server failure, the Online Mover reassigns a server from the shared buffer to the impacted reservation in less than a minute, as shown in ④ of Figure 6. This quick decision may not be optimal and the *Async Solver* may generate a better assignment later. Using a shared random-failure buffer saves capacity and is feasible because of the low random-failure rate and the short server-replacement time. RAS uses long-term trends and forecasting to predict how many extra servers are needed for random-failure events. Currently, this is 2% of the total region capacity.

Within a region, our infrastructure is designed to handle the failure of a whole MSB without causing noticeable impacts on services. RAS handles correlated failures through *embedded buffers*, i.e., the buffer servers are already preallocated into reservations. In the event of a correlated failure, the Twine Allocator can immediately use the buffer servers to host containers without the Online Mover taking any action. We do not use shared buffers for correlated failures because a correlated failure may eliminate thousands of servers in one MSB. Finding and moving replacement servers at that scale in near real time would add significant complexity on the critical path of failure mitigation, leading again, to a trade-off between server assignment quality and response time.

In addition to handling correlated failures, embedded buffers are also used to handle large-scale planned maintenance events at the granularity of an MSB. In other words, planned maintenance and correlated failures share the same buffer, which leads to significant cost savings. In the event of a correlated failure, maintenance events are paused and embedded buffers are returned to deal with the failure in two phases. 75% of embedded buffers are returned within seconds while the remaining 25% within 30 minutes. This is possible as our maintenance scheduling system limits concurrent maintenance operations to 25% of an MSB.

The size of a reservation's embedded buffer needs to be as large as the reservation's largest capacity in any MSB, because it has to anticipate the failure of any MSB. RAS optimizes the placement to minimize the buffer size. For example, in a production region with 36 MSBs, RAS required 4.2% of capacity for embedded failure buffers. Taking into account imbalances between capacity requests and spread of heterogeneous hardware across MSBs, the minimal required buffer capacity is 4.06%. If hardware was perfectly spread across MSBs, the lower bound would be 100 / 36 = 2.8%.

3.3.2 Storage Services

Storage services have some unique requirements. They utilize all capacity in a reservation, including embedded buffers, to deploy redundant data copies. RAS enables replication-based storage services by ensuring enough server spread to maintain a quorum during an MSB failure. Furthermore, it enables erasure-coding-based storage services by ensuring enough server spread to minimize data shard re-construction costs across alive servers during an MSB failure. Finally, to ensure the locality between compute and storage, RAS can

enforce that the compute capacity allocated to each datacenter matches the ratio of storage.

3.4 Elastic Reservation

When buffers are not in active use for handling failures or maintenance events, they are handed out in the form of *Elastic Reservations* to services that can utilize opportunistic capacity, e.g., asynchronous computing platform and offline machine learning training. Specifically, the Online Mover monitors the usage of a server and changes its ownership to an elastic reservation when it is idle. Elastic service owners, similar to guaranteed service owners, submit container requests by referencing the elastic reservation ID. Whenever failure handling needs the buffer capacity, servers are revoked from elastic reservations and returned back to the original guaranteed reservations.

3.5 Async Solver

The core of RAS placement decisions is a MIP solver. RAS formulates server-to-reservation assignment as an optimization problem and lets commercial solvers reason about the combinatorial complexities. In the following sections, we first summarize the intuition behind our optimization goals, and then discuss how to practically solve a large MIP problem in production. Finally, we describe the details of our MIP model.

3.5.1 Solver Intuition

RAS balances the capacity of an entire region continuously. At each solve, RAS considers the previous solve result, the capacity requests, and the server pool. The server pool information includes the hardware specification of each server, their topologies in the region, unavailability events, and their current reservation assignment. The shared random-failure buffer is treated as a standalone special reservation.

Each placement goal can be treated as a constraint or an objective. If a goal is important enough to block capacity fulfillment, then we make it a constraint. Constraints are always more dominant and thus fixing them can come at high objective costs. When competing constraints cannot be met, we soften constraints such that no constraint can regress from its initial value and there are high-priority objectives associated with fixing as many constraints as possible. Otherwise, we make it an objective and strive to set parameters appropriately, e.g., whether to introduce preemption (i.e., moving running containers) to fix such an objective.

Constraints. RAS determines the group of servers assigned to each reservation subject to capacity requirements, server availability, network and correlated failure buffer constraints. Specifically, the capacity constraint enforces that the allocated capacity for a reservation meets its requested capacity in RRUs. The availability constraint filters out servers that are unavailable due to unplanned failures, whereas unavailability due to planned maintenance is treated as usable capacity as discussed in Section 3.3.1. The network constraint aims to minimize unnecessary cross-datacenter communication traffic by enforcing compute capacity allocated to each datacenter matches the ratio of storage. The correlated-failure-buffer constraint ensures that after losing any MSB, the reservation can still meet its capacity requirement.

Objectives. RAS optimizes for multiple objectives. First, to minimize churns, RAS aims to move unused servers instead of those with running containers, and strives to maintain the same move in the current solve if a move was generated in a previous solve. Second, RAS spreads reservations across MSBs to minimize correlated-failure buffers. Finally, RAS aims to reduce hotspots that may overload rack switch uplinks.

3.5.2 Scaling the Solver

Although RAS removes server-allocation decisions from the critical path of container placement, it still needs to meet the SLO of solving within one hour, which is challenging due to the large number of servers in a region and a MIP solver's limited scalability. As RAS must ensure capacity remains fungible within a region, trivial partitioning of the problem would cause fragmentation. Our scaling strategy revolves around exploiting the symmetry of servers in the MIP. Furthermore, via phased solving, we cautiously trade solution optimality for solving speed.

Exploit symmetry. RAS exploits the natural symmetry in servers to reduce the size of the MIP problem. In Section 3.5.3, we express the basic MIP model with assignment variables $x_{s,r}$ which are 1 when server s is given to reservation r and 0 otherwise. In the MIP formulation, there are large groups of servers where all of their assignment variables $x_{s,r}$ have the same coefficients in all constraints and objectives. In practice, these groups of servers are identical in terms of our modeling. Thus, we merge them into a single integer variable representing how many of that equivalence class are assigned to a particular reservation.

Phased solving. By itself, the symmetry strategy above is insufficient because it cannot be applied to servers with different location properties such as rack, MSB and datacenter. Consider a region with 1000 reservations, 5 datacenters, 10 MSBs per datacenter, 200 racks per MSB, and 20 server types. The product of all of those values leads to a scale of 200 million assignment variables even after exploiting symmetry, which is beyond what MIP can solve within our SLO of one hour. Empirically, we find that ≈10 million assignment variables is the upper limit. Thus, we chose to cautiously trade solution optimality for reduced solving time via two-phase solving.

In the first phase, RAS solves the problem without any

rack-related goals, which allows grouping more symmetric servers into one assignment variable and reduces the problem to less than ten million variables. In the second phase, RAS solves the problem with all goals in phase one plus rack goals, but limits the problem to a subset of reservations in order to keep the number of variables under a limit. The reservations that have the worst rack-level objectives are prioritized to be selected in this second phase. This is a compromise as we cannot guarantee that rack-related objectives are immediately met for all reservations after one run of the Async Solver.

3.5.3 MIP Model

We now present the detailed MIP model of RAS. The presentation is simpler on the raw problem without leveraging symmetry. In practice, we express the problem using the raw form, and then perform an automatic translation to group symmetric assignment variables before constructing the MIP. Table 1 describes the notation used for the following problem.

Minimize:

$$\sum_{s \in S, r \in R} M_s * \max(0, X_{s,r} - x_{s,r}) \tag{1}$$

$$+\beta * \sum_{r \in R, G \in \Psi^K} \max\left(0, \sum_{s \in G} (V_{s,r} * x_{s,r}) - \alpha^K * C_r\right) \tag{2}$$

$$+\beta * \sum_{r \in R, G \in \Psi^F} \max\left(0, \sum_{s \in G} (V_{s,r} * x_{s,r}) - \alpha^F * C_r\right) \tag{3}$$

$$+\tau * \sum_{r \in R} \max_{G \in \Psi^F} \left(\sum_{s \in G} V_{s,r} * x_{s,r}\right) \tag{4}$$

Subject to:

$$\sum_{r \in R} x_{s,r} \leq 1, \qquad \forall s \in S \tag{5}$$

$$\sum_{s \in S} (V_{s,r} * x_{s,r}) - \max_{G \in \Psi^F} \left(\sum_{s \in G} V_{s,r} * x_{s,r}\right) \geq C_r, \qquad \forall r \in R \tag{6}$$

$$\left| \frac{\sum_{s \in G} (V_{s,r} * x_{s,r})}{C_r} - A_{r,G} \right| \leq \theta, \quad \forall r \in R, G \in \Psi^D \tag{7}$$

Stability objective. Expression 1 imposes a cost of M_s for each server s that is moved out of a reservation. We impose a large M_s on servers with active running containers, and a lower value if not.

Spread-wide objective. Expressions 2 and 3 promote the spread of capacity at the rack and MSB levels, respectively. α^K and α^F are the tunable proportional spread parameters that sets the desired threshold for proportion of capacity allowed within a single physical scope. Then, β is the parameter of objective penalty associated with each server exceeding the desired threshold. The spread wide objective also mitigates power and network hotspots.

Table 1: Notation for MIP

Notation	Description
S	Set of all servers
R	Set of all reservations
$x_{s,r}$	Assignment variable which is 1 if server s is assigned to reservation r and 0 otherwise
$X_{s,r}$	Constant initial assignment value
M_s	Movement cost of server s
τ	Cost of each correlated-failure-buffer server
β	Cost of each server outside spread goals
$\alpha^{K,F}$	Proportional limit of reservation for spread in K (rack) or F (MSB fault domain)
$V_{s,r}$	RRU value of server s for reservation r
C_r	Capacity desired for reservation r
$\Psi^{K,F,D}$	Partition of servers based on K (rack), D (datacenter), or F (MSB fault domain)
$A_{r,G}$	Affinity of reservation r to a partition group G

Assignment variables. Expression 5 represents our basic assignment constraints which we utilize as a primitive for the rest of the problem.

Embedded correlated-failure buffer. Expression 6 ensures that a reservation has sufficient remaining capacity after the failure of any MSB, whereas Expression 4 minimizes the correlated-failure buffer.

Shared random-failure buffer. RAS constructs a special reservation for each hardware type to represent the random-failure buffer shared by all reservations. The reservation's capacity is proportional to the expected random-failure rate.

Network affinity constraints. Expression 7 enforces a reservation's preference for physical datacenters. For example, if a service's data resides in a datacenter, its compute servers should also come from that datacenter in order to minimize cross-datacenter traffic. Systems outside RAS determine the values of $A_{r,G}$, which dictates the amount of capacity that should be allocated from different datacenters for a reservation.

4 Evaluation

RAS has been running in production for almost two years and achieves near fully allocated regions. Currently, the capacity guarantee of RAS to a reservation is to tolerate the failure of one MSB, and 2% of random failures within a region. Our evaluation sheds light on different aspects of RAS. Specifically, it answers the following questions:

(1) How does RAS perform and scale in production?
(2) What is the effect of RAS on correlated-failure buffers?
(3) How does RAS spread services across MSBs?
(4) What is the effect of RAS on cross-datacenter networks?
(5) What is the effect of RAS on power spread?
(6) Does RAS cause server-reassignment churns?

4.1 RAS Performance and Scalability

To characterize the performance and scalability of RAS, we first present the allocation time in a production region, breakdowns by phases, and solution quality of MIP. Finally, we show the allocation time in several production regions to demonstrate how RAS scales as a function of the number of assignment variables.

4.1.1 *Allocation Time Distribution*

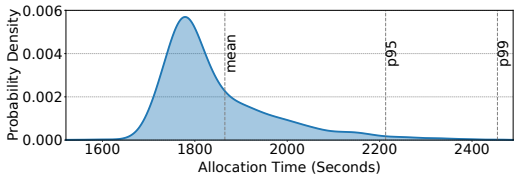

Figure 7: RAS regional allocation time distribution.

Figure 7 shows the distribution of the production allocation time over three months for a region that hosts several hundreds of thousands of servers. The mean allocation time is 1.8K seconds. The distribution further reveals that the 95th percentile is at 2.2K seconds while the 99th percentile is at 2.45K seconds, within the one-hour SLO. One reason for the tight distribution is due to moderate hardware pool changes between solves.

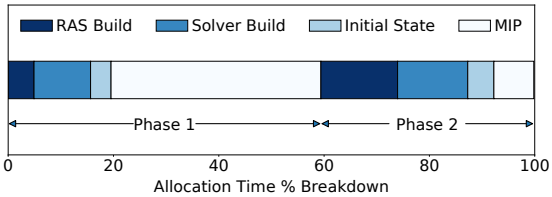

Figure 8: RAS allocation time breakdown.

Figure 8 shows a breakdown of the allocation time. *Phase 1* accounts for 60% of the total allocation time. Each phase is broken down into four steps. The *RAS Build* step builds the objectives and constraints required by RAS. The *Solver Build* step builds the constraints and objectives based on the requirements and applies the symmetric-server optimization. The *Initial State* step provides to the solver the initial assignment and perform the initial LP solve. Finally, the *MIP* step does the actual MIP solving.

Phase 1 spends 67% of its time in the MIP step. By contrast, Phase 2 spends only 19% of its time in the MIP step, whereas almost 70% of its time is split equally between the two build steps. These differences are due to differences in complexity between the two phases. Phase 1 performs a coarse-grained solve and takes into account the region's entire capacity and ensures basic capacity for reservations and failures buffers. Phase 2 further refines the server assignments done by Phase 1.

4.1.2 *Allocation Quality*

To meet the one-hour allocation-time SLO, we impose a timeout on phase 1, which may interrupt the MIP solver before a true optimal solution is found.

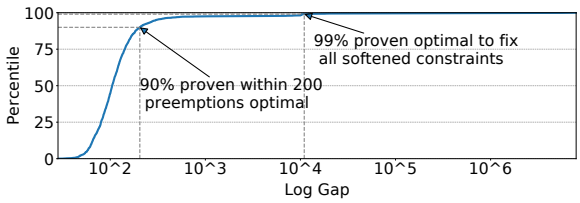

Figure 9: Phase 1 MIP quality gap.

Figure 9 evaluates the solution quality by showing the gap with respect to an optimal solution. We evaluate the quality of a solution by comparing the value of its objective function (i.e., Expressions 1–4) with a) the coefficient cost of a server reassignment across reservations requiring preemption (i.e., M_s in Table 1), and with b) the cost of not fixing an initially broken constraint (i.e., β and τ in Table 1). 90% of the solutions are optimal within 200 server preemptions and 99% of the solutions are optimal in that all initially broken constraints are fixed. Our evaluation shows that running the solver with a longer timeout often tightens the above bounds but produces no new solution, indicating that early timeout is practical. Moreover, we performed many investigations into production issues related to missed placement goals or excessive server-move churns. We found that they were almost always due to unrealistic and competing placement goals rather than low-quality solutions caused by early termination.

4.1.3 *Allocation Scalability*

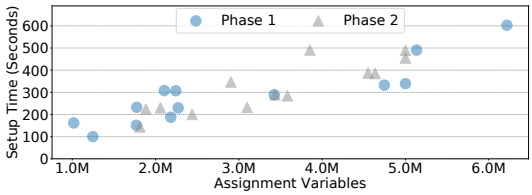

Figure 10: The time of Phase 1 & 2 spent in RAS build + solver build + initial state, measured from Facebook's different production regions and shown as a function of the number of assignment variables.

In Figure 8, even if we can reduce the MIP step's time via early timeout, the first three steps—RAS build, solver build, and initial state—provide a lower bound on the allocation time, which is reported in Figure 10. Additionally, Figure 11 shows the memory usage of the solver. Both time and memory grow linearly as a function of the number of assignment variables.

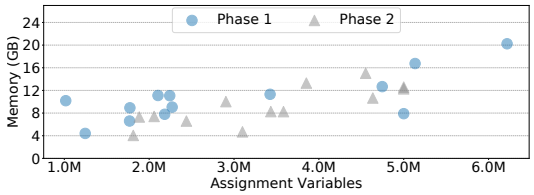

Figure 11: The memory usage of Phase 1 & 2 used by RAS, measured from Facebook's different production regions and shown as a function of the number of assignment variables.

Figure 10 motivates the design of two-phase solving. Note that Phase 2 is configured to add assignment variables until it either covers 10% of reservations or reaches a maximum of five million assignment variables. Without phasing, the full problems would be at least 10x larger than the Phase 2 problems represented in Figure 10. Extrapolating linearly, solving would require 75GB of memory but the time to just set up the largest full MIP problem would take ≈4000 seconds, exceeding our one-hour SLO. Currently, using two phases is sufficient for our problem. In the future, we will likely add more phases when we introduce additional placement goals that significantly break server symmetry.

4.2 Reduce Correlated-Failure Buffers

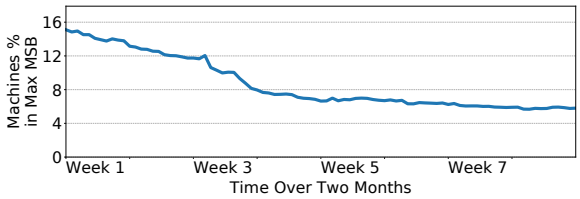

Figure 12: RAS helps reduce correlated-failure buffers over a period of two months.

Figure 12 shows the effect on spare server capacity needed to tolerate correlated failures as a region gradually enables RAS for more reservations over time. The starting point is Facebook's previous production solution that was based on greedy server assignment as described in Twine [39]. Specifically, for each service there is one MSB that hosts the largest percentage of servers used by that service. Reducing this percentage helps reduce the number of spare servers needed to tolerate the loss of one MSB. By minimizing expressions 3 & 4 of the MIP model in Section 3.5.3, RAS reduces the percentage of servers used by a service within one MSB from 15.1% down to 5.8%. As additional MSBs were added to datacenters later, RAS further reduces this percentage down to 4.2%, very close to the optimal lower-bound of 4.06%. If hardware installation in datacenters was perfectly spread across all MSBs, the theoretical lower bound would be 2.8%.

4.3 Spread of Services Across MSBs

Figure 13 shows the spread achieved by RAS for the top 30 services across all the MSBs in a region. We order the MSBs based on time of deployment—the oldest MSBs have a lower number, whereas the newest MSBs have a higher number and host the newest hardware. Each vertical bar represents the spread of a service's capacity across MSBs. The color of a cell, e.g., service 4 at MSB 24, represents the fraction of service 4's capacity allocated at MSB 24. The color code is shown on the right side of the figure. We color an MSB white if it is not utilized by a service. A service is well spread across MSBs if its vertical bar shows a relatively uniform color. Observe that RAS spreads the majority of services across all MSBs, with a few exceptions, based on other required placement constraints.

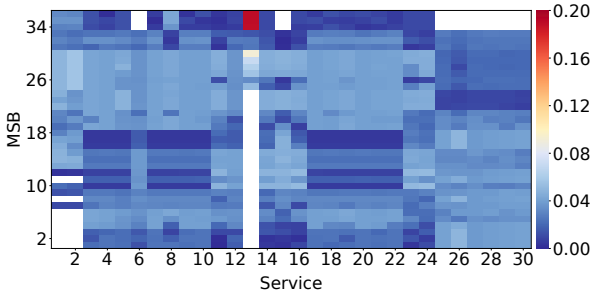

Figure 13: Spread of services across MSBs.

The 1st and 2nd services do not use some of the oldest MSBs as they require hardware that is not present in those MSBs. The 25-30th services do not use the newest MSBs, because they prefer certain hardware that is discontinued and not available in the new MSBs. The 13th service is ML-focused which is constrained to a single datacenter of the latest 12 MSBs due to its high bandwidth requirements with storage. Furthermore, this ML service desires a high amount of the newest hardware that only exists in the latest 3 MSBs, and is forced to have a high proportion of its capacity in those MSBs. Services 6 and 15 are not yet ready to utilize the newest hardware, so they have no capacity in the latest MSBs. Overall, RAS achieves a near uniform distribution of services by enforcing spread-wide objectives while catering to heterogeneous hardware, datacenter topology constraints, and constant capacity resizing performed by service owners.

4.4 Power Spread

Power is an important limiting resource within Facebook's datacenters. The same rules imposed to improve failure domain spread also improve power balance to avoid hotspots and power capping.

Figure 14 shows the normalized power consumption variance across MSBs as a region gradually enables RAS. The

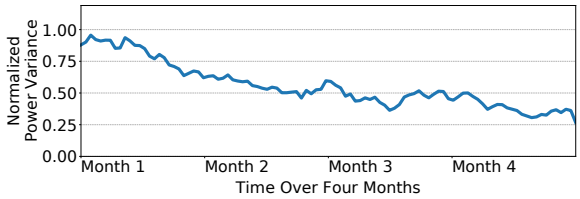

Figure 14: RAS helps reduce power usage variance across MSBs over a period of four months.

starting point is Facebook's Twine [39] greedy server assignment solution that was running in production. Overtime, RAS' placement optimizations reduce the power imbalance across MSBs. Specifically, the variance is reduced from close to 0.9 down to 0.2. This significant reduction brings datacenters much closer to a uniform power distribution across MSBs. Furthermore, RAS' placement optimization reduced the peak power usage of the most loaded MSBs and improved their power headroom from near zero to 11%. Overall, RAS leads to a higher predictability of power consumption, which enables further optimizations for job placement, capacity planning, and maintenance operations.

4.5 Cross-Datacenter Network Traffic

An exception to spread-wide rules is affinity of services with high bandwidth needs. At Facebook the main network bottleneck within a region is cross-datacenter bandwidth. Furthermore, networking requirements of services exhibit a heavy tail. Most services have minimal networking requirements but the last few percentiles of services can swiftly saturate cross-datacenter bandwidth. Optimizing placement of the few such services within a region is important in order to minimize cross-datacenter traffic.

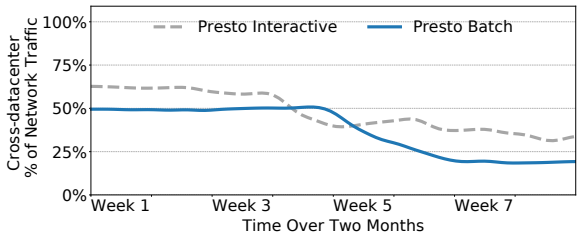

Figure 15: RAS helps reduce cross-datacenter network traffic over a period of two months.

Figure 15 shows the percentage of cross-datacenter traffic for two services as RAS gradually enables cross-datacenter networking affinity constraints in expression 7 of the MIP in Section 3.5.3. They are interactive and batch SQL query services based on Presto [37]. On one hand, placing a service entirely within a single datacenter would eliminate cross-datacenter traffic. On the other hand, spreading a service across datacenters reduces the buffer capacity for handling correlated failures. RAS strikes a balance among different optimization constraints. It reduced cross-datacenter traffic by more than 2.3x and 1.6x for batch and interactive Presto services respectively, by enforcing network constraints while meeting other constraints.

4.6 Server Movement Churns

As RAS' continuous optimization reassigns servers across reservations, it incurs the overhead of migrating containers running on those affected servers, which is particularly harmful for stateful services.

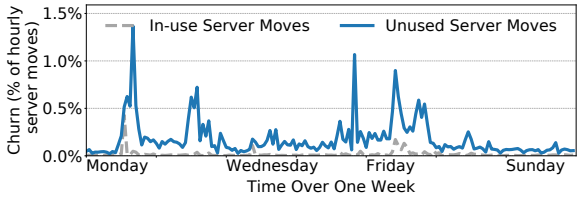

Figure 16: Weekly in-use vs. unused server moves.

Figure 16 shows the percentage of server movement per hour. RAS enforces via expression 1 of the MIP in Section 3.5.3 to minimize movement and use a 10x smaller penalty for servers without active running containers since their moves are virtually free. The average hourly rate of unused server moves is 10.6x greater than that of in-use. Overall, ≈80% of servers run containers for guaranteed reservations and RAS is able to meet most placement objectives by selecting moves from the remaining 20% of servers that are either idle or temporarily assigned to elastic reservations. The spikes in the figure align with working hours, during which server moves are mainly driven by capacity requests submitted by engineers. During off-hours, there are few capacity requests and server moves are mostly driven by random failures.

5 Discussion

5.1 How to Apply RAS to Other Systems

RAS has been running in production for almost two years and manages the entire Facebook fleet. We believe that the following key ideas of RAS can be selectively integrated into other systems: 1) Present to users the abstraction of dynamic reservations as opposed to static clusters, 2) Decouple server-to-cluster assignment from container placement, 3) Formulate server-to-reservation assignment as a general optimization problem, 4) Scale the solver by exploiting server symmetry to create equivalence sets and orchestrate solves into a hierarchy of phases based on optimization and infrastructure-level scopes (e.g., availability zones and racks).

The key ideas of RAS can potentially be applied incrementally without being adopted all at once. Suppose your infrastructure has 100 logical clusters (i.e., 100 reservations)

and ≈1,000 servers per cluster. You may apply MIP directly to optimization problems of this scale without using the idea 4 above for scalability. Furthermore, if your infrastructure cares less about complex constraints such as hardware heterogeneity or is less strict in enforcing expectations such as spread across fault domains, you may use simpler heuristics that tailor to your needs to dynamically assign servers to logical clusters, without using MIP. Still, this approach provides the benefits of flexible logical clusters, as opposed to static clusters that may strand unused capacity.

If your infrastructure operates multiple physical clusters and already has a quota system that admits jobs at the time of job submission, you can apply RAS to convert each static physical cluster to a dynamic logical cluster (i.e., reservation) and obtain immediate benefits without changing your quota system. The quota system can apply to resources in a reservation similar to how it applies to a physical cluster. Instead of using static clusters, which have many limitations described in Section 1.1, RAS allows you to easily grow or shrink reservations to avoid stranded capacity. Moreover, a reservation can enlist servers from different fault domains, which improves fault tolerance of jobs running in a reservation.

We believe that RAS can be retrofitted into existing cluster managers because the integration point is narrow, i.e., around how a cluster manager maintains a list of servers in a cluster. This interface should exist in every system. For example, Kubernetes' tool "`kubeadm join`" adds a node to a cluster. At Facebook, Twine is a 10-year old cluster manager and we were able to retrofit RAS into it successfully.

5.2 Additional Placement Goals

RAS is relatively new at Facebook, and we described the current placement goals in Section 3.5.2. In the future, we plan to leverage RAS to further increase the efficiency of our datacenters. Specifically, we plan to incorporate additional placement goals to further minimize network hotspots at the rack-switch level, power hot spots across various power domains, as well additional hardware constraints such SSD burnout reduction through IO-aware server assignments. The expressiveness of MIP allows us to easily introduce new optimization goals.

5.3 Lessons Learned

Prioritize buffer capacity. When there is a capacity crunch for a particular service and extra capacity is required, it can be convenient to dip into currently unused buffer capacity. However, depleting buffers reserved for failures can run a real risk of harming the entire region and causing site outages. Thus, RAS treats buffers just like a large, important service that cannot be downsized. This forces pool operators to find a safer solution by downsizing services that are less important.

Visibility into optimization decisions. Having granular visibility into the optimization decisions and the reasons behind those decisions made by the solver is important to operate a capacity management system at scale. Specifically, it is important that we are able to describe to service owners why they received a certain composition of hardware generations or particular spread across fault domains. Similarly, when a capacity request gets rejected due to some requirements not being met, the rejection message needs to explain the reason; otherwise, it is not actionable.

Running MIP on the critical path. While MIP provides flexibility and expressibility, our use of a third-party MIP solver introduces runtime risk from rare bugs. We manage this risk by gradually rolling out new placement policies, monitoring runtime and memory metrics to detect anomalies, and thoroughly root-causing production issues.

Stacking reservations. RAS provides capacity guarantees at the granularity of individual servers. Stacking containers on a server is left for the container allocator to handle. However, this puts burdens of capacity management and finding efficient stacking onto the reservation owner. To improve efficiency, we are actively extending RAS so that a single server can be shared by multiple stackable reservations.

5.4 Challenges

Although RAS is able to make guarantees on capacity, it also introduces a few challenges.

Capacity-request delays. The RAS solver may take up to one hour to grant a new capacity request, which is too slow if the capacity is needed to handle an urgent site outage. For emergencies, RAS provides an out-of-band mechanism to directly write server assignments to the Resource Broker to grant immediate capacity without obeying all placement guarantees. Then, future solves will correct any placement guarantees that were broken by this process. For normal operations, the delay is a worthwhile trade-off for the benefits of high-quality resource allocation. This out-of-band mechanism is also used as a back-up in a case where the RAS solver may be unavailable.

Extra service preemption. RAS continuously optimizes server assignments to achieve better resource allocation, which may result in a higher preemption rate for services. This requires service owners to build more flexible services that can quickly adapt to a new composition of servers. This in particular imposes more burden on sharded stateful services. Sharding layers have to re-evaluate how to spread shards across the new composition of capacity. RAS strives to minimize these preemptions and carefully vet any new placement goal that may significantly increase preemptions.

Rigid capacity boundaries and random failures. To clearly divide responsibilities, the container allocator works

exclusively within a single reservation. This rigid capacity boundary can be problematic for scenarios where random failures exceed planned failure limits. In such situations, service owners will not have replacement capacity until a slow RAS solve provides healthy capacity. Before RAS was integrated with Twine, there were less strict claims to capacity, so any server without a container could be leveraged as a replacement in this scenario at the risk of poor service placement.

Outages larger than a single MSB. Despite best efforts to keep correlated failures down to at most a single MSB of capacity, there are still unexpected events which can exceed this amount. The operational response today considers the region not operational and re-routes traffic to other regions as our capacity planning ensures we can handle peak load with a single region down. However, that is not the ideal long-term solution as the probability increases of multiple regions being down simultaneously. Today, in such an outage scenario, RAS considers all reservations equally and does not revoke granted capacity. RAS will need to adapt to make the best trade-offs in capacity among reservations to keep a region as functional as possible.

6 Related Work

Cluster scheduling and solvers. Prior research [10, 12, 16, 17, 21, 30, 40, 43, 45] has proposed several heuristic-based solutions for scheduling and resource management. A body of research has adopted solvers [13, 20, 24–26, 38, 41, 42], usually within a small cluster. Medea [20] uses MIP to place containers for long-running applications only within a cluster of a few thousand nodes. DCM [38] allows a developer to use SQL to express placement constraints, which is then translated into an optimization problem. DCM is evaluated via simulation up to only 10K nodes. Other works have used solvers to improve network flows [11, 15, 22, 29]. Moreover, multiple projects focused on job placement to provide high availability of services [2, 7, 8, 44]. At Facebook, both RAS and Shard Manager [33] use a common library called ReBalancer to formulate constrained optimization problems. Internally, ReBalancer can choose different backend solvers to solve an optimization problem. ReBalancer uses a MIP solver for RAS, but uses a local-search-based solver [1] for Shard Manager because Shard Manager needs to perform near-realtime shard-to-container allocation in seconds. Previous approaches on scheduling can be supplemental to RAS, as they can schedule containers within the logical cluster created by RAS. Furthermore, RAS presents a MIP formulation of server assignment and scales it to handle the resources of a multi-datacenter region for the first time.

Cluster scalability and server assignments. Scaling cluster managers has received a significant amount of attention in the past. Kubernetes [31] and Hydra [14] approach scalability through federations while Twine [39] uses sharding. RAS can be integrated with either approach to provide continuously optimized server assignments. The problem of assigning servers to clusters has received little attention in the past. Kubernetes' cluster autoscaler [32] can respond to workload growth by provisioning virtual machines in a public cloud and adding them to a node pool. This is similar to RAS growing a reservation, but RAS does much more complex optimizations. Cloudlab [18] presents a quota system within time windows that perform late-binding of containers to servers. It supports a single hardware-type, does not provide capacity guarantees in the presence of faults and does not perform continuous optimizations of resources. HarvestVMs [5] provide flexible VM instances that aim to exploit underutilized resources in a similar vein to AWS Spot instances [4], with the ability to grow or shrink based on underlying hardware. RAS can directly support this use case through elastic reservations, by building HarvestVMs on top of elastic reservation in order to maximize efficiency and provide strict SLAs to services.

Capacity reservations. Cloud providers commonly provide resource reservation offerings [3, 6, 23, 34, 35]. By reserving resources users can have significant cost savings compared to the pay-as-you-go model. Despite the cost-benefits of these approaches, there is little information of how reservations are materialized, how or if they provide guarantees against large-scale failures, how heterogeneous hardware resources and datacenter realities are handled, and if cloud providers perform one time allocations versus continuous optimizations like RAS.

7 Conclusion

We identified the need to provide guaranteed capacity to services. We presented RAS, Facebook's region-scale resource allocator, that addresses this need in the face of datacenter realities such as random and large-scale correlated failures, heterogeneous hardware, and datacenter maintenance. RAS introduces the concept of reservations and adopts a two-level architecture that dynamically assigns servers to reservations and decouples it from container allocation. We formulated server assignments as a MIP problem and discussed our techniques to scale RAS to handle the resources of a multi-datacenter region. Finally, we shared our experience with RAS and our strategy to deploy further region-scale optimizations.

References

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. 2003. Local search in combinatorial optimization. Princeton University Press.

[2] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association.

[3] Alibaba. 2021. How to Use Alibaba Cloud Reserved Instances to Reduce Costs. https://www.alibabacloud.com/blog/how-to-use-alibaba-cloud-reserved-instances-to-reduce-costs_595237.

[4] Amazon. 2021. EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/ambati

[6] AWS. 2021. EC2 Capacity Reservations. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-capacity-reservations.html.

[7] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. Turaga, and C. Venkatramani. 2008. Towards Optimal Resource Allocation in Partial-Fault Tolerant Applications. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*.

[8] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. 2012. Surviving Failures in Bandwidth-Constrained Datacenters. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*.

[9] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*.

[10] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association.

[11] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, Kevin Cole, and Dmitri Perelman. 2019. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*.

[13] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-Based Scheduling: If You're Late Don't Blame Us!. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*.

[14] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. 2019. Hydra: A Federated Resource Manager for Data-Center Scale Analytics. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association.

[15] Emilie Danna, Subhasree Mandal, and Arjun Singh. 2012. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*.

[16] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, USA).

[17] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGARCH Comput. Archit. News* (2014).

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[19] Facebook. 2019. Accelerating FacebookâĂŹs infrastructure with application-specific hardware. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[20] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[21] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*.

[22] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*.

[23] Google. 2021. Reserving Compute Engine Zonal Resources. https://cloud.google.com/compute/docs/instances/reserving-zonal-resources.

[24] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*.

[25] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[26] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[27] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 845–861.

[28] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center . In *Proceedings of the 8th USENIX Symposium on Networked*

Systems Design and Implementation.

[29] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles.*

[30] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.*

[31] Kubernetes. 2020. https://kubernetes.io/.

[32] Kubernetes cluster autoscaler. 2020. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[33] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. 2021. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles.*

[34] Microsoft. 2021. Azure Reservations. https://azure.microsoft.com/en-us/reservations/.

[35] Oracle Cloud. 2021. Ensure Business Continuity With Capacity Reservations. https://blogs.oracle.com/cloud-infrastructure/ensure-business-continuity-with-capacity-reservations.

[36] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles.*

[37] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 1802–1813.

[38] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association.

[39] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 787–803.

[40] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *EuroSys'20.*

[41] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. 2012. Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12).*

[42] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16).*

[43] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing.*

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).*

[45] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys).*

[46] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.*

[47] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. In *Proceedings of the VLDB Endowment.*