Optimizing Resource Allocation in Hyperscale Datacenters: Scalability, Usability, and Experiences

Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang, *Meta Platforms*

https://www.usenix.org/conference/osdi24/presentation/kumar

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Optimizing Resource Allocation in Hyperscale Datacenters: Scalability, Usability, and Experiences

Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir,
Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang

Meta Platforms

Abstract

Meta's private cloud uses millions of servers to host tens of thousands of services that power multiple products for billions of users. This complex environment has various optimization problems involving resource allocation, including hardware placement, server allocation, ML training & inference placement, traffic routing, database & container migration for load balancing, grouping serverless functions for locality, etc.

The main challenges for a reusable resource-allocation framework are its usability and scalability. Usability is impeded by practitioners struggling to translate real-life policies into precise mathematical formulas required by formal optimization methods, while scalability is hampered by NP-hard problems that cannot be solved efficiently by commercial solvers.

These challenges are addressed by Rebalancer, Meta's resource-allocation framework. It has been applied to dozens of large-scale use cases over the past seven years, demonstrating its usability, scalability, and generality. At the core of Rebalancer is an expression graph that enables its optimization algorithm to run more efficiently than past algorithms. Moreover, Rebalancer offers a high-level specification language to lower the barrier for adoption by systems practitioners.

1 Introduction

In Meta's private cloud, millions of servers are deployed to host tens of thousands of services, powering dozens of products that serve billions of users. In such a complex environment, we routinely encounter a wide variety of resource allocation problems. The following are some real examples:

- *Hardware placement* [30]: Decide when and where to add or remove server racks in a datacenter while balancing competing goals such as staff work schedule, power budget, spread across fault domains, and colocation for proximity, e.g., ML training servers requiring high-bandwidth network.
- *Service placement* [32]: Decide on the allocation of servers to services while spreading each service across fault domains and optimizing the matching between services and server generations, as different services exhibit varying performance across server generations.
- *Service sharding* [25]: For sharded services like databases, determine how to migrate data shards both within and across datacenter regions in response to real-time load changes, while ensuring spread across fault domains and preventing too many concurrent changes that could destabilize the system.
- *Traffic routing* [5]: Route traffic from billions of users to geographically distributed datacenters while optimizing network latency and datacenter load.
- *Locality groups* [35]: Intelligently partition serverless functions into groups, with each server executing functions exclusively within its designated group. The objective is to enhance locality, maximize hits in the JIT code cache, and balance CPU and memory usage across servers.

All these problems have a common pattern where we want to *assign* a set of *objects* to a set of *bins* in a way that optimizes specific *objectives* while meeting certain *constraints*.

Mixed-Integer Programming (MIP) is a well-known technique that can be used to solve such assignment problems. In this approach, assignment variables, denoted as v_{ij}, take the value of 1 if object i is assigned to bin j, and 0 otherwise. A MIP solver determines optimal values for these variables, optimizing the specified objectives while adhering to the given constraints.

While MIP is conceptually straightforward and has been explored in systems research [11, 21, 41], reports of its usage in large-scale production systems are limited. For instance, in the context of load balancing for sharded services, Google's Slicer [2] relies on hand-crafted heuristics, and Azure Service Fabric [21] unsuccessfully experimented with MIP before eventually adopting simulated annealing. Meta's own sharding system, Shard Manager [25], initially used hand-crafted heuristics for several years, but it became too complex to add new features. Eventually, it adopted the framework described in this paper.

The limited adoption of MIP in solving large-scale systems problems is primarily due to its two major limitations: usability and scalability.

Usability. Despite MIP's conceptual simplicity, most systems practitioners lack the training to translate real systems' complex, and sometimes ad hoc, policies into MIP's precise mathematical formulas. To address this usability gap, DCM [38] enables developers to express constraints using familiar SQL statements, while Wrasse [34] employs a domain-specific language for the same purpose. However, industry adoption of these approaches is yet to be reported, and thus, their generality remains unverified. Moreover, they do not sufficiently address the scalability challenge described below.

Scalability. Recall that assignment variables v_{ij} represent whether object i is assigned to bin j. Therefore, an assignment problem has $|\mathbb{O}| \times |\mathbb{B}|$ variables. Here, $\mathbb{O}$ is the set of objects, $\mathbb{B}$ is the set of bins, and $|\mathbb{O}|$ and $|\mathbb{B}|$ are their sizes. Although one can define assignment variables differently to reduce their number, the overall input size of a MIP problem formulation is still $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$; we omit the details here. In our large-scale private cloud, assignment problems involve up to several million objects and 100,000 bins. However, even approximate MIP solvers would struggle with $\mathcal{O}(10^{11})$ assignment variables, not to mention that most assignment problems are NP-hard.

1.1 Overview of Rebalancer

To address the usability and scalability challenges, we have developed Rebalancer, a generic assignment problem solver. Over the past seven years, it has been applied to dozens of diverse use cases, demonstrating usability, scalability, and generality.

There are three core issues in designing a solver: *model specification*, *model representation*, and *model solving*. To address the scalability challenge, Rebalancer represents the model as a directed acyclic graph (DAG), reducing the model size from $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$ to $\mathcal{O}(|\mathbb{O}| + |\mathbb{B}|)$. Moreover, the graph representation is a fundamental reason why Rebalancer's optimized local search can solve the model more efficiently than past local search algorithms [1]. To address the usability challenge, Rebalancer supports declarative model specification through intuitive APIs, automatically translating high-level specifications into the graph representation for efficient processing. We discuss each of these topics below.

Model specification. To address the usability challenge, Rebalancer employs a three-step approach to incrementally elevate the level of abstraction for ease of use. First, it introduces essential modeling constructs, such as *dimensions* (representing objects and bins' attributes like CPU and memory) and bin *scopes* (representing server, rack, datacenter, etc). Next, it provides an *expression API* to expose commonly used expressions (e.g., Max and Sum) for transformations on these constructs, as well as recursively on other expressions. Finally, leveraging these expressions, it exposes a high-level *spec API* implementing dozens of common objectives and constraints.

The high-level spec API enables developers to effortlessly construct assignment problems (see Figure 1). For example, CapacitySpec allows developers to specify constraints such as memory usage on a server with 64GB memory, and GroupCountSpec can be used to specify that each server rack can host at most one replica of a database shard, ensuring spread across fault domains.

If no existing high-level spec meets a developer's needs, they can always utilize the expression API to define a new spec, which can then be exposed for other developers to reuse in the future. In practice, across dozens of use cases supported by Rebalancer, 85% of their constraints and objectives are implemented by directly reusing existing specs, without resorting to the expression API. Moreover, we demonstrate that it is relatively easy to define a new spec using the expression API.

Model representation. After a developer leverages the specs to define an assignment problem, Rebalancer translates it into an *expression graph* $\mathcal{G}$. Each node in $\mathcal{G}$ represents an expression constructed from its child expressions. Due to careful choices in model constructs and common expressions, the size of $\mathcal{G}$ is scalable, $\mathcal{O}(|\text{objects}| + |\text{bins}|)$, as opposed to $\mathcal{O}(|\text{objects}| \times |\text{bins}|)$ in the MIP problem formulation.

Model solving. Since the class of assignment problems are in general NP-hard, our goal is to find a high-quality solution within a reasonable amount of time. For small problems, Rebalancer translates the expression graph into a MIP model and solves it with commercial solvers. However, large problems at Meta are either too large for commercial solvers or have tight deadlines, e.g., due to real-time load balancing requirements. To address these limitations, Rebalancer implements an optimized local search algorithm. The graph representation is the fundamental reason why this algorithm is more efficient than past local search algorithms [1, 29, 33, 34].

Contributions. This paper makes the following contributions.

- **First of a kind.** To our knowledge, Rebalancer is the first framework that solves a wide range of assignment problems and has been extensively validated through production usage in hyperscale infrastructure.
- **Model specification.** Seven years of hands-on experience with dozens of use cases has allowed us to iteratively improve and arrive at the current modeling constructs and high-level specification API. Although other usability-enhancing abstractions have been proposed before, their generality has not been validated through widespread production usage.
- **Model representation.** Due to careful choices in model constructs and expressions, the size of the expression graph is scalable, $\mathcal{O}(|\text{objects}| + |\text{bins}|)$, as opposed to $\mathcal{O}(|\text{objects}| \times |\text{bins}|)$ in the MIP problem formulation.
- **Model solving.** The expression graph is also an important distinction that enables us to design a highly scalable algo-

rithm for model solving, utilizing optimized local search on top of the graph.

In subsequent sections, we will describe model specification, model representation, and model solving, in that order.

2 Model Specification

Rebalancer allows developers to easily specify an assignment problem as a composition of a predefined set of high-level *specs*. Figure 1 shows such an example.

2.1 Modeling Constructs

To ensure reusability of the specs, Rebalancer defines a set of modeling constructs that can flexibly represent user requirements:

- **Dimensions.** A *dimension* is a mapping of each object and bin to a number. For example, the memory dimension of a server (a bin) specifies the server's memory capacity, while the memory dimension of a task (an object) specifies the amount of memory needed to run the task. Dimensions can also represent complex relationships. For example, we can define a prohibitedObjects dimension, where an object takes a value of 1 or 0, depending on its assignability to a bin.
- **Bin hierarchy.** Rebalancer uses *scopes* to represent the hierarchical structure of bins. For example, the *datacenter* and *rack* scopes represent servers in a datacenter or rack. A scope divides bins into sets called *scope items*. For example, under the *rack* scope, the scope items $rack_1$ and $rack_2$ represent the set of servers in those specific racks.
- **Object hierarchy.** Similar to scopes and scope items for bins, an *object partition* is an aggregation of objects, which may not be necessarily disjoint. Each set in the partition is referred to as a *group*. For example, in the context of cluster management, all tasks are *partitioned* into jobs and a job is a *group* of tasks that run the same executable.

As a concrete example of using these constructs, Figure 1 defines two dimensions, CPU and storage, to model resources; a rack scope as a fault domain; and a job partition where each group comprises tasks that run the same executable.

2.2 Definition of Utilization

Next, we describe an important concept called *utilization* of bins or scope items. It encompasses, but is more general than the intuitive concept of a server's CPU or memory utilization. Formally, given an object-to-bin assignment and a dimension D, the *utilization* of a bin b_j with respect to D, denoted $\mathsf{util}(b_j, D)$, is defined as the sum of dimension values of all objects assigned to the bin. That is,

$$\mathsf{util}(b_j, D) = \sum_{o_i \in \mathbb{O}} D(o_i) \cdot v_{ij}, \qquad (1)$$

where $D(o_i)$ is the dimension value of object o_i, and v_{ij} takes value 1 if o_i is assigned to b_j and 0 otherwise.

```
// Do not exceed CPU and storage capacity.
addConstraint(CapacitySpec(
    scope="server", dimension="CPU"))
addConstraint(CapacitySpec(
    scope="server", dimension="storage"))

// A rack hosts no more than one task per job.
addConstraint(GroupCountSpec(
    scope="rack", dimension="ObjectCount",
    partition="job", limit=1))

// Balance CPU and storage usage across servers.
addObjective(BalanceSpec(
    scope="server", dimension="CPU"))
addObjective(BalanceSpec(
    scope="server", dimension="storage"))
```

Figure 1: Using Rebalancer's high-level specs to specify the objectives and constraints for assigning tasks (objects) to servers (bins).

For example, a bin's utilization with respect to the ObjectCount dimension is simply the number of objects assigned to it. Note that utilization can also be defined for a scope item with respect to a group of objects. For example, $\mathsf{util}(\mathsf{rack}_r, \mathsf{job}_j, \mathsf{ObjectCount})$ counts the number of job_j's tasks deployed on servers in rack_r.

Flavors of utilization. The basic definition of utilization in Eqn 1 is inadequate for certain intricate scenarios. For instance, in the process of migrating a data shard from a source server to a destination server, it may be necessary to first load the shard on the destination, ensuring its healthy operation, before removing it from the source. Throughout this transition period, which might be prolonged when involving substantial data copying, the shard consumes resources on both the source and destination servers. Another scenario involves modeling system stability requirements, such as restricting the number of objects moved in and out of a bin.

To accommodate these complexities, we introduce additional utilization variants. Utilization of bin b_j is the sum of contributions from a set of objects. In Eqn 1, this set consists of objects *currently* assigned to bin b_j, referred to as AFTER. Additionally, we define sets like INITIAL, representing the objects initially assigned to b_j, and $\mathsf{STAYED} = \mathsf{INITIAL} \cap \mathsf{AFTER}$, denoting the initial objects that remained in bin b_j. Extending the notation to include the *temporal* set of objects contributing to utilization as $\mathsf{util}(b_j, D, \mathsf{TIME})$, we refer to it as $\mathsf{TIME}_{\mathsf{util}}$, where TIME can be AFTER, STAYED, or INITIAL. Concretely, expressions like $\mathsf{INITIAL}_{\mathsf{util}}$, $\mathsf{AFTER}_{\mathsf{util}}$, and $\mathsf{STAYED}_{\mathsf{util}}$ capture the utilization by the sets of INITIAL, AFTER, and STAYED objects, respectively.

Through set operations on these base definitions of util, we can create derived definitions such as:

- $\mathsf{NEW}_{\mathsf{util}} = \mathsf{AFTER}_{\mathsf{util}} - \mathsf{STAYED}_{\mathsf{util}}$ which captures the utilization of *new* objects that moved into bin b_j.
- $\mathsf{OLD}_{\mathsf{util}} = \mathsf{INITIAL}_{\mathsf{util}} - \mathsf{STAYED}_{\mathsf{util}}$ which captures the uti-

lization of *old* objects that moved out of bin b_j.

- $\mathsf{ANY}_{\mathsf{util}} = \mathsf{INITIAL}_{\mathsf{util}} + \mathsf{AFTER}_{\mathsf{util}} - \mathsf{STAYED}_{\mathsf{util}}$ which captures the utilization of objects that were in bin b_j at any point in time. Note that subtracting the STAYED term avoids double counting for objects that stayed in b_j.

These utilization variants help capture complex scenarios. For instance, $\mathsf{ANY}_{\mathsf{util}}$ can model double occupancy, while $\mathsf{NEW}_{\mathsf{util}}$ and $\mathsf{OLD}_{\mathsf{util}}$ can model system stability.

Internally, Rebalancer translates these utilization variants into their mathematical forms. $\mathsf{AFTER}_{\mathsf{util}}$ is simply Eqn 1 using v_{ij} determined by the current assignment. $\mathsf{INITIAL}_{\mathsf{util}}$ is a constant value that can be pre-computed from Eqn 1 using the initial assignment. To implement $\mathsf{STAYED}_{\mathsf{util}} = \mathsf{util}(b_j, D, \mathsf{STAYED})$, a new dimension named D_j^{init} is introduced for each bin b_j. This dimension takes the value $D(o_i)$ for all objects initially assigned to b_j and zero otherwise. We can again use Eqn 1 with the fact that $\mathsf{util}(b_j, D, \mathsf{STAYED}) = \mathsf{util}(b_j, D_j^{\mathsf{init}}, \mathsf{AFTER})$.

2.3 Common Specs

Over the past seven years, through the process of supporting dozens of large-scale use cases, we have iteratively developed the common specs shown in Table 1. On average, an assignment problem uses seven specs, with a maximum of 14.

Table 1 highlights the reuse of many specs, with six used only once, suggesting they are developed when needed initially. The high-level spec API prioritizes ease of use, while the low-level expression API offers extensibility for new spec development. It provides different flavors of util expressions, mathematical operators (Max, Sum) for aggregation, and transformation operators (Step, Ceil, Log, and Power). Besides being user-friendly, this API enables modeling of non-linear properties, providing a more convenient alternative to crafting a MIP problem formulation from scratch. Overall, the expression API facilitates the implementation of simple specs in dozens of lines of code, and even the most complex specs can be implemented in a few hundred lines of code.

Consider, for example, the introduction of UtilIncreaseCostSpec to prioritize moves to servers with CPU utilization less than a specified threshold T_0. When all servers have CPU utilization over T_0, it favors the one with the least utilization. Using the expressions API, this is modeled in just 65 lines of code by adding the penalty expression $\mathsf{Power}(\mathsf{excessUtil}_i,\ 2)$ to the objective for every server i. Here, $\mathsf{excessUtil}_i = \mathsf{Max}(0,\ \mathsf{util}(\mathsf{server}_i, \mathsf{CPU}, \mathsf{AFTER}) - T_0)$.

3 Case Studies of Model Specification

In this section, we describe how to model several real world assignment problems using Rebalancer's spec language.

3.1 Hardware placement

To provide context, we first outline our infrastructure hierarchy: datacenter region→datacenter→suite→main switch board (MSB)→server row→server rack→server. Globally, there are tens of datacenter regions and each region has multiple datacenters within a few miles' radius. Each datacenter consists of four large rooms called suites. Each suite has three MSBs, each supplying power to 10K to 20K servers laid out as rows of server racks. Each rack hosts tens of servers.

A datacenter undergoes continuous evolution with the addition of new server racks and the removal of existing racks for maintenance or decommissioning. The hardware-placement problem involves computing an optimized weekly schedule for these operations, considering the staff's work schedule, ensuring hardware spread across fault domains, and adhering to capacity constraints on power, network, etc.

We model racks as *objects* and *(week, position)* pairs as *bins*, where a *position* is a physical location in the datacenter. We introduce *scopes*, such as *MSB* and *week*, where each *scope item* is a collection of bins associated with the same MSB and week respectively. Similarly, an *object partition* of racks can be defined, where each group consists of racks of the same type. In the following, we outline a small subset of objectives and constraints for hardware placement.

- **New racks.** Initially, all new racks belong to a special bin called *unassigned*. Applying ToFreeSpec on that bin ensures that new racks are assigned to certain *(week, position)*.
- **Power and network constraints.** This is achieved by using CapacitySpec at different *scopes* of the infrastructure hierarchy such as position, MSB, and suite.
- **AI Zone.** AI server racks must be placed in an *AI zone*, which is a special section of the datacenter connected by a high-bandwidth network. To enforce the placement of AI racks in the AI zone, we introduce a new dimension AiRack, which takes the value 1 for AI racks and 0 otherwise. Similarly, this dimension has a limit of 1 for AI zone positions and 0 otherwise. We then apply CapacitySpec with the AiRack dimension over the *position* scope.
- **Place certain racks in the same week.** This is achieved by putting those racks into a *group* and applying ColocateGroupSpec to the group over the *week* scope.

Overall, as rack changes occur incrementally over time and we compute a solution for each datacenter region separately, the hardware-placement problem is relatively small in size, involving hundreds of objects and thousands of bins. For these small problems, Rebalancer translates the expression graph into a MIP problem formulation and employs a MIP solver, instead of a local-search solver, to ensure high-quality results.

3.2 Service Placement

Hardware capacity in datacenters are allocated to teams responsible for different products in the form of quotas called *reservations*. Whole servers or fractions of a server's resources are assigned to reservations while adhering to all kinds of constraints. While reservations can be either global

Spec name	Description	Usage count	Lines of code
CapacitySpec	Enforce that the utilization of a scope item is within specified limits.	19	340
GroupCountSpec	Restrict utilization by objects of a group placed in the same scope item, commonly used to enforce the spread of objects across scope items. For instance, a job can have at most one task in a rack.	17	480
AvoidMovingSpec	Do not move any of the specified list of objects.	16	120
BalanceSpec	Balance utilization across scope items.	12	250
MinimizeMovementSpec	Minimize the number of objects that move into or out of a scope item.	12	90
MovesInProgressSpec	Objects specified as moving from one bin to another by the previous solver run must finish the move.	10	65
NonAcceptingSpec	Specify scope items that are not accepting incoming objects.	9	100
MinimizeBinsSpec	Minimize the number of bins utilized.	8	105
AssignmentAffinitiesSpec	Indicate that specific objects prefer specific bins.	6	90
ToFreeSpec	Free up certain bins. For example, move services out of servers that will be decommissioned.	5	70
ColocateGroupsSpec	Place objects of the same group in the same scope item, e.g., placing a job's tasks in the same rack.	5	100
GroupMoveLimitSpec	Limit how many objects of the same group (e.g., a database's replicas) can move concurrently.	4	95
AvoidAssignmentsSpec	Prevent assignments of certain objects to scope items. For example, in hardware placement, an AI zone in a datacenter only accepts AI server racks.	4	60
GroupDiversitySpec	Every scope item must get objects from at least (or at most) K different groups.	4	80
SingleGroupFailureBufferSpec	Provide additional buffer objects when a group of objects fails together. Used in service placement to ensure that services have enough servers even when a fraction of a datacenter fails.	2	145
DrainCapacitySpec	Allow draining objects from a faulty bin to other bins while respecting capacity constraints.	1	80
MoveGroupSpec	Move objects in the same group together across bins.	1	75
MinimizeNthLargestUtilization	Minimizes the utilization of scope items with the n-th largest utilization	1	85
MaximizeAllocationSpec	Maximize utilization on a set of scope items	1	65
UtilIncreaseCostSpec	Prefer moving objects to underutilized scope items.	1	65
Logical Or/And Specs	Perform a logical OR/AND of certain specs.	1	55

Table 1: List of 21 most frequently used specs out of a total of 28 specs currently supported by Rebalancer. Remaining specs are specific to their respective usecases and their descriptions involve defining concepts beyond the scope of this paper.

or regional, our discussion focuses on regional ones for simplicity. A regional reservation can comprise servers from any datacenter within the same region but not across regions. Our cluster management system treats each reservation as a dynamic virtual cluster and deploys the owner team's jobs on it.

In this service-placement problem [32], we model servers as *objects* and reservations as *bins*. Moreover, we group servers by *MSB*, *rack*, and hardware *type*, which become object partitions. Below, we describe some used objectives and constraints.

- **Capacity sufficiency.** If a reservation specifies a demand of X units for a server type Y, we fulfill it by utilizing CapacitySpec with the count dimension for each server type. The variability in performance among services on various server types can be represented as a dimension. Thus, we can optimize for assigning servers of a specific type to services that can extract optimal performance from them.
- **Spread.** GroupCountSpec ensures that servers allocated to a reservation are spread across *MSB* and *rack* partitions.
- **Stability.** As new reservation requests emerge or existing ones are updated, we run the solver to update both old and new reservations. Recomputing solutions for old reservations is necessary, as it enables the relocation of servers from old reservations to new ones, facilitating global optimization. However, moving many servers out of an old reservation, even if those servers are replaced with new ones, would cause churns to services running on those servers. We use MinimizeMovementSpec to minimize churns.
- **Fault tolerance.** For each reservation, we allocate additional buffer capacity to ensure that in case any single MSB in a datacenter region goes offline due to failure or maintenance, there is still sufficient capacity in the reservation. This requirement is modeled using SingleGroupFailureBufferSpec.

A large service-placement problem involves up to 700K servers (objects) in a datacenter region and 6K reservations (bins). We solve one such problem per region every hour. The solve frequency and associated downstream actions necessitate that Rebalancer must finish solving the problem within 10 minutes. Initially, Rebalancer converted the expression graph to a MIP problem and solved it with a MIP solver. However, recently we switched to using the faster local-search solver due to both the growing problem size and the desire to reduce the solve time to fulfill capacity change requests faster. This experience demonstrates one advantage of Rebalancer—it can take the same problem specification and flexibly decide which solver to use based on the problem size and time limit.

3.3 Service Sharding

Sharded services, such as databases, are prevalent at Meta and account for 68% of the total RPC traffic [25, 36]. They often host about 100 shards per Linux process for improved efficiency, and shards are dynamically migrated across these Linux processes to balance the load. For simplicity, we refer to each such Linux process as a "*server*", assuming one process per server. To ensure redundancy, each shard has multiple replicas, which are grouped together using *object partitions*. This problem assigns *shards* (objects) to *servers* (bins) while meeting various requirements, some of which are described below.

- **Capacity limit.** We use CapacitySpec to ensure that servers are not overloaded. Given that cross-server shard moves are not instantaneous, we use ANY_{util} to account for double occupancy (§2.2).
- **Limit churns.** To cap the number of moves per server or per shard, we use CapacitySpec with *ObjectCount* as the dimension and NEW_{util} and OLD_{util} as the utilization (§2.2).
- **Region preference.** Certain shards prefer servers in specific datacenter regions because the users accessing those shards are close to those regions. This preference is modeled using AssignmentAffinitiesSpec with the *region* scope.
- **Load balancing.** To ensure that the load is balanced across servers, we use BalanceSpec with the *bin* and *region* scopes for regional and global load balancing, respectively.
- **Fault Tolerance.** To ensure that a shard's replicas are spread across various fault domains, such as rack and MSB, we use GroupCountSpec with *replica* as the partition and *rack* or *MSB* as the scope.

The largest sharding problems involve 1.8M objects and 27K bins and have a solve time limit of five minutes. Rebalancer's local search algorithm has scaled well to produce high-quality solutions for such large problems.

3.4 Message Queue Placement

Many people use Meta's messaging products. On the server side, a message queue is created for each user to store messages intended for delivery. Placing the message queue in a datacenter region close to the user reduces latency. This problem is to assign user queues (objects) to datacenter regions (bins). However, treating each user as an individual object is inefficient, so we aggregate users objects with common properties into a bundle and treat each bundle as an object. The bundles are computed offline based on properties such as proximity and connectivity of users within a bundle. The following are some supported requirements:

- **Minimize latency.** Every user bundle has a numerical affinity to each datacenter region. The affinity is equal to the negative of the average network latency to a region for users in the bundle. Utilizing AssignmentAffinitiesSpec as an objective with these affinities minimizes the total latency.
- **Colocate related user bundles.** If two bundles' users frequently communicate with each other, colocating them in the same region would reduce both latency and cross-region traffic. ColocateGroupSpec achieves this purpose, where each *group* is a set of related bundles.
- **Buffer capacity for disaster recovery.** One service level objective is that any single datacenter region can go offline without causing disruption to users. For this purpose, a traffic matrix with elements t_{ij} specifies that in the event of region i failure, a fraction t_{ij} of region i's traffic will be redistributed to region j. Rebalancer must ensure that each region has enough spare capacity to absorb the incoming redistributed traffic. This is achieved by using DrainCapacitySpec to enforce that the peak utilization of a bin with the worst case spillover traffic must be within its capacity limit.

The message-queue problem typically involves tens of thousands of objects and tens of bins, and is solved only once a week. Because of the small scale and lenient solving deadline, Rebalancer translates it into a MIP problem. The traffic-routing problem described in §1 shares some commonality with this problem, but it needs to update the global edge-to-datacenter traffic matrix every few minutes. Hence, it utilizes local search to achieve a low average runtime of 5 seconds.

3.5 Kubernetes Scheduler

When evaluating the flexibility of Rebalancer's specs using the use-case examples above, a question naturally arises: do the specs inherently cover these examples because they are designed to support them? To showcase Rebalancer's flexibility, we implement Kubernetes' scheduling policies using Rebalancer's existing specs. While Rebalancer handles load balancing of containers across machines in production, it does not handle the Kubernetes-like initial container placement in our fleet. This function was implemented in Meta's cluster manager using heuristics similar to those in Kubernetes years before introducing Rebalancer and is still in active use.

Specifically, to prepare for a direct comparison with DCM [38] in performance evaluation (§6), we implemented Kubernetes' scheduling policies listed in Figure 2 of the DCM paper. To improve usability, DCM uses SQL statements to express allocation policies and internally translates these SQL statements into a constraint satisfaction problem, which is then solved using the Google OR-tools CP-SAT solver.

In Kubernetes, container *pods* are scheduled on *nodes* (machines). We represent pods as objects and nodes as bins. Similar to DCM, our implementation schedules a batch of pods together. All unscheduled pods of the current batch are placed in a special *unassigned* bin, and we impose a ToFreeSpec constraint on that bin, forcing Rebalancer to place them on certain nodes. Resource limits, such as CPU and memory, are enforced using CapacitySpec. Affinity of pods for specific nodes is specified using CapacitySpec with a custom dimension representing affinity. Inter-pod anti-affinity for replica groups is modeled using GroupCountSpec. Fixing certain pods to nodes is achieved with AvoidMovingSpec, and so forth.

Overall, we are able to model Kubernetes' scheduling constraints using Rebalancer's existing specs in about 500 lines of code, which is comparable to 550 lines of SQL-based specification in DCM. This demonstrates that Rebalancer is flexible and its usability is comparable to DCM. We will compare the performance of DCM and Rebalancer in §6.

3.6 Other Use Cases

In addition to the examples described above, Rebalancer supports dozens of use cases at Meta, including Linux container rebalancing across servers [39]; routing traffic from globally distributed edge datacenters to main datacenters [5]; grouping serverless functions to improve locality [35]; balancing online ML training workloads across regions while considering the priority of ML workloads; minimizing the number of replicas for ML inference models or databases deployed across geo-distributed datacenters, while adhering to latency SLO and meeting varying user request rates; various sharding systems that have requirements different from the one in §3.3; assigning work tickets to engineers; and so forth.

4 Model Representation

Once an optimization problem is specified using specs, Rebalancer materializes them into an expression graph. Recall that Rebalancer's expression API supports operators such as Max and Sum for aggregation, and Step, Ceil, Log, and Power for transformation. For example, $\mathsf{Step}(x)$ evaluates to 1 if x is positive and 0 otherwise. We translate the specs into a recursive composition of expressions that reuses common expressions to obtain a compact expression graph.

4.1 Translating Specs into Expressions

We use several examples to illustrate how to translate specs into expressions. *CapacitySpec* enforces that the utilization of a resource is within specified limits. For instance, with a CapacitySpec applied to the scope server and dimension CPU, Rebalancer creates $|\mathbb{B}|$ constraints (one per server) in the form of $\mathsf{util}(\mathsf{server}_i, \mathsf{CPU}, \mathsf{AFTER}) \leq L_i$, where L_i represents the CPU limit of server_i. To model double occupancy, CapacitySpec would use ANY instead of AFTER.

MinimizeMovementSpec minimizes the movement of objects into or out of a scope item. Rebalancer adds the expression $\mathsf{util}(S_{\text{out}}, \mathsf{count}, \mathsf{NEW}) + \sum_j \mathsf{util}(S_j, \mathsf{count}, \mathsf{NEW})$ to the objective, where S_{out} represents the set of bins that do not belong to any scope items, such as the *unassigned* bin in the hardware-placement example (§3.1). Note that each moving object is only counted once. Specifically, the first term captures the objects that move out of all the scope items, while the second term captures objects that move within the scope items.

GroupDiversitySpec ensures diversity in the set of objects assigned to a bin. For example, the servers (objects) assigned to a service's global reservation (bin) must come from at least k datacenter regions so that in the event of a region failure, there is at least some capacity available for the service. In this case, assuming objects are partitioned into groups G_i based on their region, Rebalancer adds $|\mathbb{B}|$ constraints (one per service) in the form of $\sum_i \mathsf{Step}(\mathsf{util}(b_j, G_i, \mathsf{count})) \geq k$. Note that the inner Step expression evaluates to 1 if bin b_j contains objects from group G_i and 0 otherwise.

4.2 Reducing Model Size

In the previous section, we discussed how to translate specs into mathematical formulas using expressions such as util. However, the direct representation of util as shown in Equation 1 is inefficient as it would lead to a problem representation size of $\Theta(|\mathbb{O}| \times |\mathbb{B}|)$. To address this scalability challenge, Rebalancer implements several efficient custom expressions that help reduce the problem's model representation to $\Theta(|\mathbb{O}| + |\mathbb{B}|)$. Below, we describe one such expression called Lookup, which is used to efficiently implement util.

Object Lookup. The insight behind Lookup is that, in most cases, an object's dimension value remains unaffected by the bin to which it is assigned. For instance, a task consumes the same amount of memory irrespective of the server on which it runs. This allows representations of utilizations for different bins and scope items to share and reuse these dimension values, reducing the problem input size by a factor of $\Theta(|\mathbb{B}|)$, resulting in an overall input size of $\Theta(|\mathbb{O}| + |\mathbb{B}|)$.

Specifically, for each such static dimension D, we establish an *object-vector*, denoted as V_D, representing a mapping from objects to their dimension values. Given an *object-vector* V_D and a scope item S_i, a Lookup represents an efficient aggregation operation over the (object, bin) pairs for bins in S_i. For example, $util(S_i, D) = \mathsf{Lookup}(S_i, V_D)$ simply aggregates the utilization across all bins in S_i with respect to D through lookup. Note that the memory usage of Lookup itself is of constant size since it only keeps references to S_i and V_D, while the representations of S_i and V_D, with sizes $\mathcal{O}(|\mathbb{B}|)$ and $\mathcal{O}(|\mathbb{O}|)$, respectively, are shared and reused across all expressions. This leads to an overall problem size of $\Theta(|\mathbb{O}| + |\mathbb{B}|)$.

Expression graph. Since constraints in Rebalancer are inequalities in the form of $f(\cdot) \leq 0$, they can be combined using the Max expression. For example, although a CapacitySpec results in $|\mathbb{B}|$ constraints (one per bin), it can be simplified into a single constraint by rewriting it as

$$\mathsf{Max}_i(\mathsf{Lookup}(b_i, V_D) - L_i) \leq 0.$$

As each constraint and objective can be written as a recursive composition of expressions, we can encapsulate an assignment problem's all constraints and objectives in a DAG $\mathcal{G}$. The nodes of $\mathcal{G}$ correspond to expressions, and each objective or constraint in the problem is a subgraph of $\mathcal{G}$; see Figure 2 for an example. For a node $v \in \mathcal{G}$, we use $\mathsf{children}(v)$ to denote the set of all nodes w such that $v \to w$ is an outgoing edge from v. The $\mathsf{type}(v)$ of a node (e.g., Max) is its mathematical operator, and $\mathsf{children}(v)$ represent its inputs. The DAG $\mathcal{G}$ is obtained from a recursive composition of these operators.

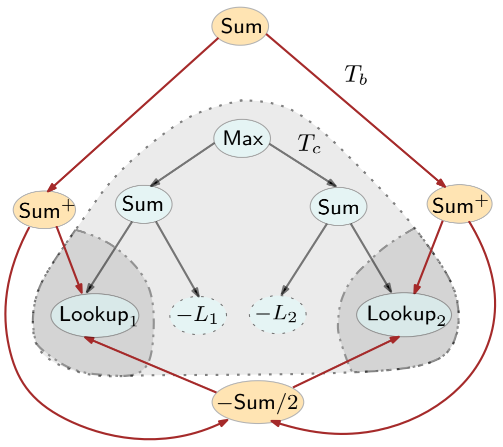


Figure 2: This *expression graph* represents a simplified version of the task-allocation problem shown in Figure 1. It aims to balance the CPU utilization across two servers, server$_1$ and server$_2$. It uses BalanceSpec on the CPU dimension as the objective (subgraph T_b) and CapacitySpec on the CPU dimension as the constraint (tree T_c). Nodes in the intersection of T_b and T_c namely Lookup$_1$ and Lookup$_2$ are reused. Constant nodes are shown in dashed circles. L_i is the CPU limit of server$_i$, Lookup$_i$ represents the lookup for the CPU dimension on server$_i$, meaning server$_i$'s CPU utilization. Sum$^+$ is a shorthand for $\max(0, \text{Sum})$

Algorithm 1 Local search using expression graph $\mathcal{G}$

```
1:  while exit-conditions are not met do
2:      $\mathcal{L} \leftarrow$ generate_candidate_moves($\mathcal{G}$)
3:      for local change $\delta$ in $\mathcal{L}$ do
4:          obj$_\delta \leftarrow$ evaluate_moves($\mathcal{G}, \delta$)
5:          if obj$_\delta > 0$ then
6:              discard $\delta$    ▷ violates constraint or worsens objective
7:          end if
8:      end for
9:      $\delta^* \leftarrow \min_{\delta \in \mathcal{L}}$ obj$_\delta$    ▷ best local change
10:     apply_moves($\mathcal{G}, \delta^*$)
11: end while
```

5 Model Solving

After representing an assignment problem as an expression graph $\mathcal{G}$, the next step is to solve the problem. For small problems, Rebalancer translates it into a MIP problem and solves it with a MIP solver. For large problems, Rebalancer implements its own optimized local search. The existence of the expression graph is a fundamental reason why Rebalancer's local search is more efficient than existing local search algorithms.

5.1 Using MIP Solver

To translate an expression graph into a MIP problem, each expression implements a recursive mipTranslate operation. This operation, based on the expression's type, converts it into a linear combination of binary assignment variables v_{ij} that indicates whether object i is assigned to bin j. Invoking mipTranslate on the root nodes of an expression graph yields a MIP model, which can subsequently be solved using a commercial solver. Note that the MIP model's input size is $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$, which is not scalable. Therefore, we only use the MIP solver for relatively small problems.

5.2 Graph-Assisted Local Search

In contrast to the MIP solver's all-or-nothing approach to finding the *optimal* solution, local search [1] incrementally generates a set of object *moves* that improve upon the initial assignment but without guaranteeing optimality. Each move (o_i, b_s, b_d) corresponds to reassigning object o_i from its source bin b_s to its destination bin b_d.

Although local search has been applied to assignment problems before [19, 33], the uniqueness of our approach, as highlighted in Algorithm 1, lies in its exploration of the expression graph for all its main steps: (1) generating candidate moves, (2) evaluating them, and (3) applying the best moves. In the rest of this section, we describe the main ideas that enable our algorithm to scale to millions of objects and bins.

5.2.1 Generating Candidate Moves

Because each object can potentially be moved from its current bin to any other bin, there are a total of $|\mathbb{O}| \times (|\mathbb{B}| - 1)$ candidate moves to consider. Obviously, it would be too expensive to evaluate all of them. There are two natural ways to reduce the candidate set: (1) restricting the search space to one bin at a time and finding the best moves involving that bin, or (2) restricting the search space to one object at a time and finding the best moves for that object. Rebalancer takes approach (1) because the number of bins is usually much smaller than the number of objects. With this settled, we still need to decide in which order to evaluate bins and, given a bin, how to propose candidate moves. We discuss these topics below.

Bin selection. Our insight here is to first evaluate *hot bins* that potentially can have the biggest impact on the overall objectives by moving objects into or out of these bins. The structure of the expression graph already captures what objectives are affected by which bins and by what amount. Intuitively, for example, if there is a directed path from a node v to a Lookup on bins b_1 and b_2, moving objects in and out of these bins will improve node v's value. The idea is to process the leaf nodes of $\mathcal{G}$ (such as Lookup) in a *greedy order* of their contribution to the objective. This ordering of leaf nodes gives us a sequence of sets of bins $S_v, S_w, \ldots, S_z$ and we can use these sets to infer the *hottest bin*.

Move strategies. After identifying a hot bin, Rebalancer explores different *move strategies* to move objects into and out of it. For example, the SINGLE move strategy considers moving every object in b_s to every other bin b_d exhaustively and accepts the best move. There are also variants of SINGLE, such as SINGLE_GREEDY, which accepts the first improving move, and SINGLE_RANDOM, where b_d belongs to a small sample of randomly chosen bins. A commonly used effective strategy is to first use SINGLE_RANDOM for some period of time when opportunities for improvement are abundant and later switch to using SINGLE_GREEDY when opportunities for improvement become scarce.

In addition to variants of the SINGLE strategy, Rebalancer also supports more complex strategies such as swapping objects between two bins, and using the Kernighan–Lin algorithm to identify the move-destination bin; see details in the Appendix. Finally, Rebalancer also supports custom strategies that exploit domain knowledge. For example, Shard Manager [25] uses Rebalancer to move shards (objects) across servers (bins) to balance the load. If a hot server has many small shards and a few large shards, going through the shards sequentially may spend most of the time evaluating moving small shards that have little impact on the objective. Instead, Rebalancer evaluates large shards earlier, which not only accelerates the search but also reduces the number of shard moves.

5.2.2 Evaluating and Applying Moves

The remaining components of Rebalancer's algorithm are *evaluating* and *applying* moves. When given a candidate move (o_i, b_s, b_d), a naive way to evaluate its impact is to apply the move to the initial assignment to obtain the new assignment. Then, we compute the new assignment's objectives from scratch through a full graph traversal. However, since we already have the value of every graph node under the initial assignment, few nodes might be affected by applying the candidate move. We can significantly speed up the computation by only recomputing the values for these affected nodes.

Bottom-up change propagation. To only recompute the changed nodes, we preprocess the leaf nodes in the expression graph to build a map from objects to the leaf nodes that reference them. Similarly, we build a map from bins to the leaf nodes that they affect. Given a move candidate, we use the two maps to identify a set of leaves affected by the change. We then traverse from those leaves to the roots, and the reached nodes along the way are the set of nodes whose values need to be recomputed.

Minimal computation during a node update. When recomputing the value of a changed node, iterating over all its child nodes is often unnecessary, as only a small fraction of them likely have changed. Depending on the type of the node, we can store additional information to speed up the recomputation. Below, we provide an example for the Max node, while similar optimizations exist for other node types.

For the Max node, we separately compute the maximum value of its changed child nodes (denoted as z_1) and the maximum value of its unchanged child nodes (denoted as z_2). Then, the new value of the node is simply $\max(z_1, z_2)$. To compute z_2 efficiently, we maintain a sorted list of child nodes ordered by their decreasing node value. We iterate over this list and stop at the first child node that is unchanged. This child node's value is z_2. Note that the runtime of this algorithm is proportional to the number of changed child nodes c instead of the total number ℓ of child nodes. This algorithm incurs the overhead of $\mathcal{O}(c \log \ell)$ to update the sorted child list, but it only occurs when a move is accepted and applied. In practice, the number of evaluated but rejected candidate moves often dominates.

Parallelizing Move Evaluations Since move evaluations do not affect the current state, we can use multiple threads to evaluate candidate moves in parallel before we pick the best candidate to *apply*. This improves the number of evaluations per second (evals/s) by an order of magnitude enabling local search to make faster progress. Although the exact number depends upon the problem instance, we are able to obtain roughly 150k evals/s for most large instances. For example, for a large sharding problem, parallel move evaluations resulted in 170k evals/s and 12k applied moves within the time limit of 300s. In contrast, sequential move evaluations result in 25k evals/s and 2k applied moves in the same time limit. In this case, local search was able to progress six times faster due to parallelization of move evaluations.

5.3 Identifying Equivalent Objects

In addition to focusing on hot bins to reduce the search space, for certain commonly used objectives and constraints, we can identify objects that are equivalent from a modeling perspective, thereby effectively reducing the number of objects. This optimization can be applied to both local search and MIP solvers. For example, in the problem of placing tasks on servers, all tasks that belong to the same job are equivalent since they all affect the constraints and objectives in the same way. For a set of equivalent objects, we only need to explore moves with at most one of those equivalent objects, which reduces the search space. In Rebalancer, we employ a recursive algorithm that exploits the expression graph to compute sets of equivalent objects. Various additional details about our solver algorithm can be found in the Appendix.

6 Evaluation

In this section, we evaluate Rebalancer's scalability and solution quality, and compare it with alternative approaches, such as DCM [38] and MIP partitioning techniques similar to POP [31]. Additionally, we assess the efficacy of local search techniques such as hot bin ordering.

Figure 3 shows the statistics of the problems solved by Rebalancer in production during a typical week. These tens of millions of solves span diverse use cases outlined in §3. The largest cases involve service sharding (1.8M shards, 27k servers) and service placement (700k servers, 6k reservations). We will use these application scenarios in evaluation.

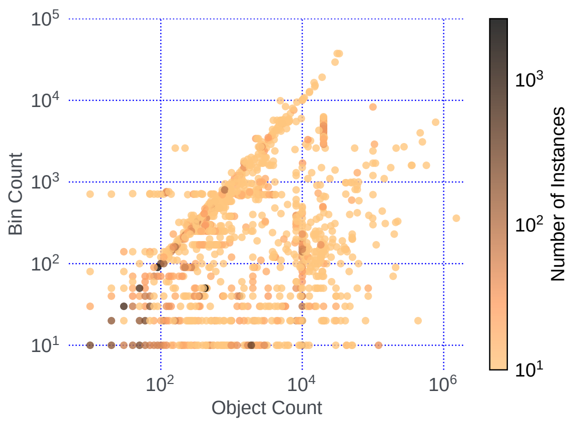


Figure 3: Samples of production problems solved by Rebalancer. The P99 (99th percentile) solve time is 16s with a problem size of 65k objects and 5k bins.

6.1 Comparison with DCM

In the context of cluster management, DCM [38] uses SQL to express policies for placing *pods* on *nodes*. It translates these SQL statements into a constraint satisfaction problem, which is then solved using the Google OR-tools CP-SAT solver. DCM's scalability bottleneck is the CP-SAT solver, which has computational intractability similar to MIP solvers.

We replicated DCM's implementation of Kubernetes scheduler in Rebalancer (§3.5). To stress-test our implementation, we impose inter-pod anti-affinities for all replica groups, a condition known to increase scheduling difficulty. Similar to evaluations in the DCM paper, we use the Azure dataset [27]. All experiments are conducted on a machine with 40 CPU cores and 256 GB of RAM.

We will quote numbers from the DCM paper instead of conducting measurements on our machines because DCM's open source code cannot run on our production machines due to the security setup of those machines. We also cannot run Rebalancer on third-party machines due to its dependencies on tools only available in our production environment. Despite the inability to directly compare their absolute performance on the same machine, their scalability trends are evident from their respective evaluations using the same dataset and implementing the same Kubernetes scheduling algorithm.

We also compare local search's SINGLE_GREEDY move type, which evaluates placing an unscheduled pod on every node, with the SINGLE_RANDOM move type, which randomly selects a fraction f of nodes as targets. If unsuccessful, it repeats the process with the remaining unexplored nodes.

Scalability. We evaluate Rebalancer under two scales: the *DCM-scale*, scheduling 50 pods per batch on clusters with 1k, 5k, and 10k nodes (similar to that in the DCM paper); and the *hyperscale*, scheduling 500 to 5k pods per batch on clusters with 10k, 50k, and 100k nodes. Using larger pod batches in the hyperscale setup is motivated by the fact that the arrival rate of pods typically increases with the cluster size. As shown in Figure 4, the P99 (99th percentile) per-pod scheduling latency of SINGLE_GREEDY is less than 35 ms for instances up to 10k nodes. However, beyond 10k nodes, SINGLE_GREEDY becomes progressively slower, while SINGLE_RANDOM continues to scale well. We have found a sample size of $f = 10\%$ (capped at 1k bins) to offer a good trade-off between solution quality and runtime.

Overall, Figure 4 demonstrates Rebalancer's significant scalability advantage over DCM. Due to inherent scalability limitations in DCM's CP-SAT solver, the largest problem tackled in the DCM paper involved scheduling 50 pods in a 10k node cluster, with close to 30 ms scheduling latency per pod. In contrast, even with a cluster size of 100k nodes (10 times that of the DCM experiment) and a batch size of 5k pods (100 times that of the DCM experiment), Rebalancer with SINGLE_RANDOM achieves a per-pod latency of 14ms.

Solution quality. We compare Rebalancer's local search with an optimal MIP solver in an experiment that places 10k pods from the Azure dataset on 500 nodes, with the objective of maximizing the number of placed pods. As shown in Table 2, neither MIP nor local search can place all pods due to their aggregate resource demand surpassing the capacity of 500 nodes. While SINGLE_RANDOM places 1.4% fewer pods than MIP when nodes are full, its runtime is 5.2 times faster.

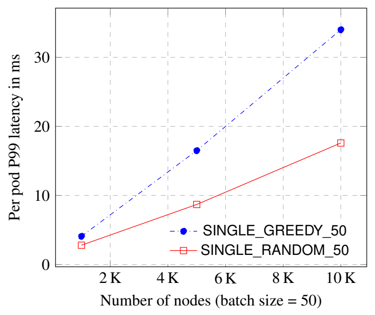
Test case	Optimal MIP solver	SINGLE GREEDY	SINGLE RANDOM
Azure dataset	94.6% (31ms)	92.8% (8ms)	93.2% (6ms)
Pathological N=10	100% (209ms)	97.1% (0.6ms)	97.8% (0.9ms)
Pathological N=100	N/A (timed out after 600s)	97.4% (26.5ms)	97.7% (22.4ms)

Table 2: Percentage of placed pods and the scheduling latency.

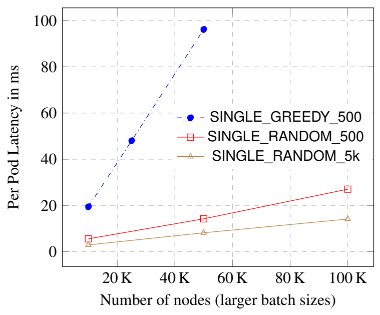
To evaluate local search in a scenario where it is unlikely to perform well, we design a pathological case parameterized by N, with $31N$ pods and $50N$ nodes, each with 32GB memory. Pod memory requirements follow an exponential distribution over 5 groups: the first $16N$ pods with 2GB memory, the next $8N$ with 4GB, the next $4N$ with 8GB, the next $2N$ with 16GB, and the last N pods with 32GB. The only way to schedule all pods is when the total used memory on every node is precisely 32GB. Due to the nature of local search, finding this uniquely optimal solution is unlikely. As shown in Table 2, local search can place more than 97% of the pods, while being over 200 times faster than MIP. Additionally, as depicted in Figure 5, Rebalancer finds a solution where the memory usage across nodes is well balanced.

6.2 Comparison with Partitioned MIP

POP [31] improves scalability by partitioning a large MIP problem into smaller ones and solving them independently, coordinating solutions as needed. We have implemented a



(a) DCM-scale problems.



(b) Hyperscale problems.

Figure 4: Per-pod scheduling latency of our Kubernetes implementation.

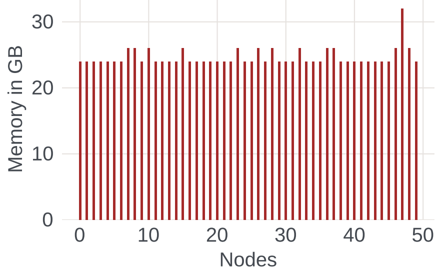


Figure 5: Memory usage across 50 nodes for the pathological case.

POP-like partitioned-MIP solver in Rebalancer in addition to its existing local-search and MIP solvers.

In Table 3, we compare local search with partitioned MIP on the service placement problem (§3.2) using our production data. Here, local search uses a combination of move types, including SINGLE_RANDOM, SINGLE_GREEDY, and SWAP. Local search is up to four times faster, meeting all the service-level requirements such as spread and capacity sufficiency, while its solution quality is less than 0.6% worse than that of partitioned MIP. Due to local search's scalability, even though we used partitioned MIP in production for a period of time, we eventually switched to local search (§7).

Problem Size (objects × bins)	Local Search	Partitioned MIP	Relative Gap
700k × 5.7k	184s	376s	0.43%
710k × 4.8k	214s	350s	0.56%
568k × 4.7k	151s	455s	0.21%
645k × 4.5k	146s	557s	0.11%

Table 3: Problem sizes and runtimes of service placement. The "*relative gap*" denotes the difference in objective values between local search and partitioned MIP.

6.3 Local Search's Individual Techniques

Next, we evaluate local search's individual techniques.

Expression graph $\mathcal{G}$ scales linearly. We validate this using three production instances of the service sharding problem, with 71k, 152k, 289k objects each and roughly 1.5-2k bins. The expression graph's memory usage grows almost linearly from 1.7GB to 3.2GB and 6.2GB.

Move evaluations are fast. As can be seen from the pod scheduling example (Figure 4), evaluating all possible 50k moves for a pod takes only 100ms. This indicates that Rebalancer can evaluate up to 500k moves per second. The techniques detailed in §5, such as bottom-up traversal of changed nodes, help achieve this speed; without them, move evaluations would be an order of magnitude slower.

Hot bin ordering is effective. A *hot bin* is one that contributes the most to the objective, and Rebalancer processes

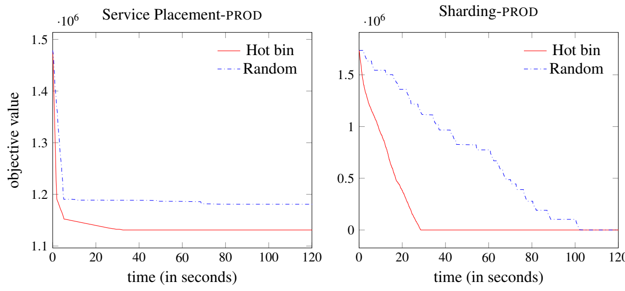


Figure 6: Comparing *hot bin* selection and *random bin* selection using service placement and service sharding.

bins in the order of their hotness. As shown in Figure 6, processing bins in the order of their hotness reduces the objective value more quickly. This results in higher solution quality when the search time is capped for large problems.

7 Experiences and Limitations

In this section, we share some learnings from using Rebalancer and discuss its limitations.

7.1 Alternative Approaches for Optimization

We summarize three categories of approaches to solving optimization problems and make our recommendations.

Approach 1: *ad hoc heuristics*. This approach directly implements heuristics to support resource-allocation policies as code. It is the most widely used approach as it is easy to start with. However, as policies grow in complexity over time, adding new ones becomes increasingly difficult, and the solution quality tends to decrease due to the intricate balance required between different policies.

Approach 2: *formal problem specification solved by a formal solver*. Systems utilizing MIP, such as Flux [10], belong to this category, but there are other formal methods as well, such as network flow optimization. Rebalancer with a MIP solver in the backend also fits into this category. To address the challenge that most systems practitioners are unfamiliar with formal problem specification, a higher level of abstractions can be introduced, for example, DCM [38] using SQL and Rebalancer using a high-level specification language. However, a formal method's solver often lacks scalability, posing a hard barrier to its application at hyperscale.

Approach 3: *formal problem specification solved by a systematic-heuristics solver*. This category includes systems that formulate problems using MIP but solve them using simulated annealing or local search. Rebalancer using a local-search solver in the backend falls under this category. The usability issue can be solved by raising the level of abstraction, similar to that for Approach 2. While good scalability is a strength of this method, its weakness lies in producing suboptimal solutions compared to formal solvers.

Recommendation: Based on our experiences, we make recommendations as follows. First, systems requiring short and predictable latency in resource allocation decisions should use ad hoc heuristics, even though they are hard to maintain and evolve. Schedulers for high-throughput, short-lived batch jobs fall under this category. Second, systems needing high-quality solutions for small to medium-sized problems should leverage formal solvers for their optimal solutions. Lastly, for hyperscale systems not requiring real-time decisions, we recommend approach 3 over approach 1, because it is much easier to add or evolve resource-allocation policies with approach 3. One advantage of Rebalancer is its ability to support both Approaches 2 and 3 using the same high-level specification, and seamlessly switch the backend solver as needed, depending on the problem scale and solve time constraint.

7.2 Experiences with Alternative Approaches

Among the three alternatives described in the previous section, our choice for service placement (3.2) evolved from Approach 2 to Approach 3, while our choice for service sharding (3.3) evolved from Approach 1 to Approach 3. We discuss these experiences below.

Service placement. The service-placement problem [32] was initially modeled using Rebalancer's high-level specification language and solved with a MIP solver. This is because initially the problem size was still manageable for MIP, the 20-minute solve time service-level objective (SLO) was sufficient, and we were (overly) worried about the solution quality.

However, as more services and machines were added to the fleet, MIP's solve time became problematic. Meanwhile, the solve time SLO was reduced from 20 minutes to 10 minutes due to the need for faster reactions to capacity change requests. We continued optimizing the MIP solver, for example, by grouping equivalent objects to reduce the number of decision variables. These incremental optimizations bought us some time, but MIP still constantly fell behind on scalability. Other issues included the MIP solver having not only unpredictable execution times but also occasionally running into infeasibility due to numerical precision issues. Debugging and fixing these elusive problems in production under time pressure were recurring pain points for engineers.

To scale MIP, we implemented a POP-like [31] partitioned MIP solver (§6.2). Initially, it performed well in both solve time and reliability. Managing hundreds of smaller subproblems meant that a few failing MIP solves would not have a fleet-wide impact. However, new requirements, such as stacking more services on bigger machines, increased the problem scale by an order of magnitude. At this point, building a partitioned MIP model, which in the worst case uses $|\text{objects}| \times |\text{bins}|$ assignment variables, was no longer practical.

This finally forced us to explore local search. With some tuning, the local search solver achieved both good solution quality and fast solve times. As shown in Table 3, the solution-quality difference between local search and partitioned MIP is less than 0.6%. The lesson for us is that, despite having the local search technology, our unwavering faith in MIP's optimality led us on a lengthy detour to reach our current state.

Service sharding. While service placement started with Approach 2 and converged to Approach 3, the service sharding system Shard Manager, went through the opposite direction, switching from Approach 1 to Approach 3.

When Shard Manager started with ad hoc heuristics more than a decade ago, Rebalancer did not exist at that time. As Shard Manager became widely adopted by many applications, its load-balancing algorithm became overly complicated. For example, it supported multiple-dimension balancing across CPU, memory, and storage, enforced rate limiting, and considered regional and global locality. Unsurprisingly, this complexity led to frequent issues where some servers were overloaded while others were underutilized. Shard Manager struggled to balance the load because the heuristics implementing sometimes conflicting policy requirements were not robust.

The team started rewriting Shard Manager with yet another heuristic implementation. It represented the topology (region, datacenter, power domain, rack, server) as a tree and enforced resource constraints across all levels of the topology. However, achieving good load balancing proved challenging even with a clean implementation. Iterative tuning of the heuristics required constant code changes that might not lead to positive outcomes and often became dead code later.

Eventually, the heuristic-based new prototype was abandoned, and the team switched to exploring Rebalancer with local search. The usability benefits were immediate, as it was much easier to experiment with different load-balancing algorithms by changing a few lines of high-level specification in Rebalancer. The main challenges were solution quality and scalability when solving problems with millions of objects and tens of thousands of bins within the five-minute solve time SLO. These requirements are well met, and in production, 90% of the solves actually finish within 10 seconds.

7.3 Experiments with Simulated Annealing

Recall that Rebalancer's internal architecture decouples problem representation from problem solving. There is a common abstraction all solvers (e.g. local search, MIP) inherit from which can be extended to support experimentation with new kinds of solvers. In the past, we experimented with a solver based on the simulated annealing algorithm. One common problem faced by local search is finding a locally optimal solution that does not lead to a globally optimal solution. This happens when there is no sequence of moves which strictly improve the objective at each step. In order to get out of a local optimum, the sequence of moves would have to temporarily decrease the objective quality. This is not contemplated in local search, but is one of the features of simulated annealing.

However, we found our implementation of simulated annealing to not be practical at all for production-scale problems. It did not beat the performance of local search in any experiments, neither in terms of quality nor run time. At our scale, the number of possible object moves at any point is large (e.g. $\mathcal{O}(10^{11})$ combinations for a problem with several million objects and 100k bins), and the vast majority of these moves are not helpful. We found that it is important to exploit the structure of the objective function and aggressively prioritize which moves to evaluate. This is primarily done by the hot bins optimization in local search. Our implementation of simulated annealing did not exploit this structure, and blindly evaluated moves with equal probability, regardless of the shape of the objective. For this reason, we believe that further research into heuristics to reduce the search space would be needed to make simulated annealing practical. This is challenging because the heuristic would not only have to select *good* moves, but also the right *bad* moves which decrease the quality but are likely to lead to an improved quality later on.

Recall that to ensure allocation stability, it is often desired to limit or minimize the number of object moves needed to improve the assignment. We found this challenging to enforce in an algorithm such as simulated annealing, which greedily makes moves that barely improve or even decrease the quality of the objective. In contrast, local search is able to maximize the objective improvement at each individual step, finding a shorter and more optimized sequence of moves.

7.4 Evolution of Rebalancer as a Library

Originally, Rebalancer was designed as a standalone executable that took an input file describing the model in a custom format. This initial design quickly became hard to maintain as it required the service invoking Rebalancer to carefully craft the input. At that point, we decided to create a strongly typed API for programmatically defining models. As a result, with the help of the compiler and runtime sanity checks, the interface-related maintenance overhead drastically reduced. The implementation of this interface was an inflection point for the adoption of the project, as it became intuitive enough to be used by many teams at Meta.

There still remained a question of whether to make Rebalancer a service or a library. We decided to make Rebalancer a library for two main reasons. First, services invoking Rebalancer need to collect and feed Rebalancer with potentially a large amount of input data, which can be done more efficiently via a library API. Second, providing a multi-tenant service is difficult, as different Rebalancer use cases can heavily contend with each other due to their high memory and CPU usage. Currently, if a particular use case still prefers to operate Rebalancer as a single-tenant service, they could do so by wrapping the library with an RPC interface, but we have not seen that in practice.

Therefore, today Rebalancer is implemented as a library that gets compiled into each project that depends on it. The resulting binary has predictable behavior, which does not change unless it is recompiled with a newer version of the code. Different usecases have their custom logic to collect the input and setup an assignment problem using Rebalancer's specification language (See Figure 1 for an example). The usecases then specify the choice of solver (MIP or Local Search) and invoke the core solver which builds the expression graph, solves the problem and returns a solution.

7.5 Handling Multiple Objectives

Each objective specifies a weight and a priority. Rebalancer combines all objectives with the same priority using a weighted sum. Finding appropriate weights for competing objectives (e.g., objA, objB) is done by first normalizing them so that their values are comparable and then selecting multiplicative weights based on their relative importance in the problem domain. Some use cases provide strict priorities for objectives, and Rebalancer ensures that it does not regress on higher-priority objectives when solving for lower-priority objectives.

Recall that local search only makes moves that strictly improve the objective value and is generally more stable. However, when using MIP, if there are two solutions with exactly the same objective value, Rebalancer may choose one of them arbitrarily, causing instability across multiple solves. In such cases, we typically add a MinimizeMovement goal that disincentivizes moves which do not strictly improve the objective.

7.6 Debugging Solver Behavior

The majority of engineering time in setting up a model goes into debugging the behavior of the solver, which includes ranking the objectives by their importance, identifying conflicting constraints, etc. Without proper tools, this process requires a deep understanding of the internals of the solver, which only engineers working on the core of Rebalancer have. Over time we identified the common questions that helped modelers understand the solver's behavior and we built a specialized UI tool for answering them: Rebalancer-explorer. At a high level, Rebalancer-explorer can be compared to debugging tools such as Whyline [24] for post-facto analysis.

Under the hood, Rebalancer-explorer loads the expression graph representation in memory and reuses many of the algorithms mentioned earlier making it extremely fast to evaluate formulas on demand. At a high level, the UI consists of:

- *Table view* to inspect objects, bins, their dimensions and their membership in partitions and scopes.

- *Tree view* to inspect the underlying expression (sub)-graph corresponding to various user provided specs.
- *Solution comparison view* to inspect the goal and constraint values for different solutions. By default, it shows a comparison of initial and final solutions.
- *Timeline navigation view* to inspect the sequence of moves performed by local search at each step and the associated objective improvements.

One example scenario is to answer questions such as *why did solver not move an object to/from a given bin*. We can use the solution comparison view to compare the goal/constraint values between the solver-generated solution, and an ad hoc solution where a specific object is moved to/from the given bin. In this case, the UI gives real-time feedback of why that move is not good – perhaps it breaks a constraint, or makes the goal value worse.

7.7 Limitations of Rebalancer

Some problems can be modeled with MIP but cannot be modeled with Rebalancer because they do not fit the abstraction of assigning objects to bins. One such example is to assign network traffic flow to links, where Rebalancer cannot model a sequence of dependencies in the link topology. However, Rebalancer 's low-level expression API and expression graph are generic enough to support these problems while providing a significant boost for usability. Consequently, we extended the expression API and expression graph to create a more flexible framework, enabling the modeling of MIP problems beyond assignment problems.

8 Related work

There is a rich body of work in the systems research community that uses optimization problem formulations for different resource allocation settings (e.g. [2–9, 11, 13, 14, 16–18, 20, 22, 25, 28, 32, 34, 37–42]). Among these, Rebalancer shares design goals with Wrasse [34] and DCM [38] domain-specific language (DSLs) for resource allocation problems.

Comparison with existing DSLs. Wrasse [34] also uses an object-bin abstraction but their specification language is limited to a small set of properties such as resource capacity but for example does not support many other important properties listed in Table 1 such as spread, balance, affinities etc. Moreover, their GPU-based solver is tightly coupled with these properties making it hard to extend. On the other hand, DCM [38] allows a user to specify constraints and goals using SQL-like queries which are then fed into off-the-shelf solvers. As discussed before, DCM's ability to scale is limited by the intractability of underlying constraint solvers.

Other relevant systems. These systems fall in one of two categories. They either use some hand-crafted heuristics or use a commercial MIP solver. Examples which use heuristics include cluster management [4, 7, 43], application sharding [2, 25], and container reallocation [33]. The paper [33] uses variation of local search to move containers across physical machines but its scalability is limited by the fact that it uses $|\mathbb{O}| \times |\mathbb{B}|$ decision variables. Examples where commercial MIP solvers are used include capacity reservation [32] and in cluster managers [11, 41]. Rebalancer, through its specification language and its ability to translate to MIP models can help set up and solve (using Xpress and Gurobi) a version of many of the above problems at a comparable scale.

There also has been some recent progress on solving hyperscale allocation problems using MIP models. POP [31] presented a technique to decompose large scale assignment problems into small ones and combine their solutions to solve the bigger problem. However, POP requires that resources (objects) are *fungible* and clients (bins) should not prefer one object over others by a large amount. These do not always hold for assignments problems we encounter, for instance, in the case of service placement, it is common for services to request specific server types. This is the reason why our POP-like partitioned MIP solver required some additional techniques.

Work on the machine reassignment problem. A set of relevant work are the papers [12, 19, 26] from the 2012 ROADEF/EURO Challenge [29]. This challenge was designed to solve a *machine reassignment problem*, which is about reassigning processes to machines to satisfy some goals and constraints such as load balancing (see [29] for details). These papers design heuristics to solve such problems and there are some broad similarities with the ideas used in Rebalancer's local search. However, it is unclear if their techniques generalize beyond the relatively small scale and type of problems that were used in the contest.

9 Conclusion

In this paper, we discuss the design and implementation of Rebalancer, a generic framework that solves a diverse set of hyperscale assignment problems. Rebalancer decouples problem specification from optimization by defining a compact graph representation, simplifies problem specification with a high-level language, and designs a scalable optimization algorithm. Finally, we shared our experiences and lessons learned from evolving solutions for service placement and service sharding.

10 Acknowledgements

We acknowledge the contributions made by all past and present members of Algorithmic Optimization, RAS and Shard Manager teams at Meta towards evolution of Rebalancer over the last seven years. In particular, we would like to thank Aravind Narayanan, Raj Rajendran, Sahil Deshpande, Apurva Samudra and Pavan Kumar for their contributions to Rebalancer. We also thank all the reviewers and our shepherd Philip Levis for their insightful comments.

References

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 739–753, 2016.

[3] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. *ACM SIGARCH Computer Architecture News*, 40(1):61–74, 2012.

[4] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

[5] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. 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)*, pages 430–446, 2019.

[6] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[7] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni Matteo Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, pages 177–192, 2019.

[8] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX Annual Technical Conference*, pages 1–14, 2019.

[10] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[11] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the thirteenth EuroSys conference*, pages 1–13, 2018.

[12] Haris Gavranović, Mirsad Buljubašić, and Emir Demirović. Variable neighborhood search for google machine reassignment problem. *Electronic Notes in Discrete Mathematics*, 39:209–216, 2012.

[13] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378, 2013.

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[15] Gurobi. Gurobi Optimizer [online]. 2023. URL: https://www.gurobi.com/resources/chapter-2-resource-assignment-problem.

[16] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: Vm allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 845–861, 2020.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[19] W Jaśkowski, Marcin Szubert, and Piotr Gawron. A hybrid mip-based large neighborhood search heuristic for solving the machine reassignment problem. *Annals of Operations Research*, 242:33–62, 2016.

[20] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[21] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Karthik Kambatla, Vamsee Yarlagadda, Íñigo Goiri, and Ananth Grama. Ubis: Utilization-aware cluster scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 358–367. IEEE, 2018.

[23] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[24] Amy J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, 2004.

[25] 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. Shard Manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles(SOSP'21)*, pages 553–569, 2021.

[26] Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Comparing solution methods for the machine reassignment problem. In *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 782–797. Springer, 2012.

[27] Microsoft. Azure public dataset. https://github.com/Azure/AzurePublicDataset, 2017.

[28] Pulkit A Misra, Íñigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *USENIX Annual Technical Conference*, pages 799–811, 2017.

[29] H Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: the roadef/euro challenge 2012, 2016.

[30] Aravind Narayanan, Elisa Shibley, and Mayank Pundir. https://engineering.fb.com/2020/09/08/data-center-engineering/fault-tolerance-through-optimal-workload-placement/, 2020.

[31] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[32] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. RAS: continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, pages 505–520, 2021.

[33] Bo Qiao, Fangkai Yang, Chuan Luo, Yanan Wang, Johnny Li, Qingwei Lin, Hongyu Zhang, Mohit Datta, Andrew Zhou, Thomas Moscibroda, et al. Intelligent container reallocation at microsoft 365. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'21)*, pages 1438–1443, 2021.

[34] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–12, 2012.

[35] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew McFague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 231–246, 2023.

[36] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios

Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[37] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[38] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 827–844, 2020.

[39] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matt 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. Twine: A unified cluster management system for shared infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 787–803, 2020.

[40] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[41] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[42] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[44] Juan Pablo Vielma. Mixed integer linear programming formulation techniques. *SIAM Review*, 57(1):3–57, 2015. doi:10.1137/130915303.

[45] FICO Xpress. Xpress Optimizer [online]. 2023. URL: https://examples.xpress.fico.com/example.pl?id=assignmentgr.

Appendix A Additional Details on Rebalancer's Local Search

The input to the local search algorithm that powers Rebalancer is an initial assignment $\mathcal{A}_0$ and the compact representation of goals and constraints as the expression graph $\mathcal{G}$. Furthermore, in the rest of this section, we assume that the following preprocessing steps have been done on $\mathcal{G}$.

- *Compute and store current values of each node*: This can be done by a simple recursive algorithm on all the roots $r_i \in \mathcal{G}$, where the value of a node Z_v is computed using the values of its children.
- *Compute and store the current upper and lower bounds for each node:* This can again be done by a simple recursive algorithm. For a node v, we use Z_v^{ub} and Z_v^{lb} to denote its upper and lower bounds, respectively.
- *Compute and store the potential of each node:* The *potential* of a node v is the difference between its current value and its lower bound (i.e., $Z_v - Z_v^{\text{lb}}$). Additionally, for each node $v \in \mathcal{G}$, we maintain a sorted order of its $\mathsf{children}(v)$, sorted by their potentials. We say that a subgraph rooted at v is *optimal* if v's potential is zero.
- *Compute and store the* affected bins *of each node:* Recall that a leaf node $\ell \in \mathcal{G}$ (such as ObjectLookup) is parameterized by a set of bins S_ℓ and changes to contents of S_ℓ will potentially change the value at node ℓ. We refer to the set S_ℓ as *affected bins* which can be recursively computed for all $v \in \mathcal{G}$ as $S_v = \{\bigcup S_w \mid w \in \mathsf{children}(v)\}$. (See also Figure 2).

A.1 Restricting the search space

One important step in any local search algorithm is to generate a neighboring set of candidate solutions. To get to a candidate solution, Rebalancer employs the notion of a *local change*, denoted δ, which is a set of ordered tuples (o_i, b_s, b_d) and where each tuple denotes the change in some o_i's assignment from some b_s to some other b_d (or alternatively, the "movement" of o_i from b_s to b_d). We refer to each tuple in δ as a *move*. So, δ is simply a set of moves. We will use the term *applying the local change* to describe the process of updating $\mathcal{A}$ with the moves in δ and denote it as $\mathcal{A} \oplus \delta$.

It is easy to see that given any two assignments $\mathcal{A}$ and $\mathcal{A}'$, there exists a set of moves δ such that $\mathcal{A}' = \mathcal{A} \oplus \delta$. So, now, the question remains as to how we can systematically generate candidate sets of moves. First, observe that even if we restrict ourselves to *single moves*—i.e., local changes where we explore moving a single object from some source to some destination bin—there are $\mathcal{O}(|\mathbb{O}| \times |\mathbb{B}|)$ choices. As discussed before, this is unacceptable for large problems, and so we must find another way to restrict the search space.

There are two natural ways to do this, *i)* restrict the search space to *one bin at a time* and find the best moves involving that bin, *ii)* restrict the search space to *one object at a time* and find the best moves for that object. Note that in both these cases the order in which we explore the bins (resp. objects) is extremely important or we may spend too much time exploring moves that yield little improvement. In Rebalancer, we restrict the search space at the *bin* scope. This decision is primarily motivated by the fact that the number of bins is usually much smaller than the number of objects, so restricting the search by bins makes faster local progress. Now, even with the choice of exploring one bin at a time, there still remains two main questions: *i)* What is the order of bins to consider, *ii)* Given a choice of bin, how to generate the set of local changes. We will answer the second question below and return to the first one in the next section.

A.1.1 Move Types

Given a bin b_s, we can generate local changes that move objects to and from b_s. To methodically generate them, we use the notion of a *move type* that describes the semantics of these changes. For example, a SINGLE move type considers moving every object in b_s to every other bin b_d. There can be other variants of SINGLE such as SINGLE_GREEDY which accepts the first improving single move, and SINGLE_RANDOM where b_d belongs to a small sample of randomly chosen bins. Although in our experience just using *single move types* often suffices, it is worth noting that the notion of a move type is highly customizable and can exploit problem-specific domain knowledge. In the following, we describe this in a greater detail.

Overall, each move type generates a set of local changes $\mathcal{L}$, evaluates each of the resulting candidate solutions (i.e., for each $\delta \in \mathcal{L}$, evaluates $\mathcal{A}' = \mathcal{A} \oplus \delta$), and if it exists, returns the δ that improves the objective the most. The logic to generate $\mathcal{L}$ varies based on the move type and the following are some commonly used ones.

- **SINGLE:** Given a source bin b_s, it tries moving every object in b_s to every other bin b_d. That is, $\mathcal{L}$ is the set of all $\delta_{i,d}$, where $\delta_{i,d} = \{(o_i, b_s, b_d)\}$ is a move set with exactly one move.
- **SINGLE_GREEDY:** Similar to SINGLE, but instead of evaluating moving every object in b_s to every other b_d, it considers the objects in some order and only considers the moves with the subsequent object if no improving move was found with the previous one.
- **SWAP:** For a source bin b_s and all other destination bins b_d, it tries swapping every object in b_s with every object in b_d .
- **KL_SEARCH:** inspired by Kernighan–Lin algorithm [23]. Given a source bin b_s, and for every possible destination bin b_d, construct the KL-move set δ_k iteratively as follows. Let $\delta_0 = \emptyset$ be an initially empty move set. The move sets at the end of i-th ieration δ_i is best of $\delta_{i-1} \cup m_i$ where m_i is a single move from b_s to b_d or from b_d to b_s. The iteration stops once moves involving all objects in b_s and b_d have been tried. The KL-move set δ_k is the best of all δ_i.

In fact, there are more complex move types in Rebalancer, but we do not go into its details here due to space constraints.

A.1.2 Identifying equivalent objects

In addition to restricting the search space to explorations from a bin, depending on the set of objectives and constraints, it might be possible to identify objects that are equivalent from a modeling perspective. For example, in the TASKS-ON-SERVERS example, all tasks that belong to the same job are equivalent, since they all affect the constraints and objectives in the same way. Observe that if we identify the sets of equivalent objects, then we can cut down the search space even further by only exploring moves with at most one object from an equivalence class. In Rebalancer, we employ a recursive algorithm that exploits the expression graph $\mathcal{G}$ to compute sets of equivalent objects.

Consider again the TASKS-ON-SERVERS example. There we would ideally want a solver to automatically identify that all tasks that belong to the same job are *equivalent*, since they all affect the objectives and constraints identically. In fact, such a feature can be quite powerful in further reducing the search space, since it allows us to discard moves that are equivalent while exploration (two moves are equivalent if they both move equivalent objects from a source bin to destination bin).

In Rebalancer, the intuition described above is formalized using the notion of *equivalent objects*. Formally, let $\mathcal{A}$ be any feasible solution of the given problem, and for two objects o_i and o_j, let $\mathcal{A}'$ be the assignment obtained by swapping their bins, i.e., $\mathcal{A}' = \mathcal{A} \setminus \{(o_i, \mathcal{A}(o_i)), (o_j, \mathcal{A}(o_j))\} \cup \{(o_i, \mathcal{A}(o_j)), (o_j, \mathcal{A}(o_i))\}$. Then, o_i and o_j are deemed *equivalent* if all constraints and objective expressions evaluate to the same value for both $\mathcal{A}$ and $\mathcal{A}'$. Alternatively, one can also, slightly informally, think of o_i and o_j as equivalent if, for every bin b and for a problem expressed in native form (i.e., using assignment variables), the modified problem that results from replacing every variable of the form $v_{i,c}$ with variable $v_{j,c}$ is mathematically equivalent to the original problem.

Ideally, we want compute an optimal set of equivalent objects (i.e., a set $P = \{I_1, \cdots, I_k\}$ of minimum size and where each object is part of one of the I_js and each I_j is a set of equivalent objects), however this is computationally hard. Hence, we use a greedy recursive algorithm which once again exploits the expression graph $\mathcal{G}$. The main component in our algorithm is for every node in $\mathcal{G}$ to maintain some information about what sets of objects are equivalent with respect to it. For example, in the case of ObjectLookup, two objects are equivalent w.r.t. it if they have the same value in the corre-

Algorithm 2 Finding sets of equivalent objects

```
1: $P \leftarrow \{\mathbb{O}\}$    ▷ initially, all objects are considered equivalent
2: repeat for each node $v \in \mathcal{G}$
3:     if children($v$) is empty then,
4:         return set of equivalent objects w.r.t. $v$
5:     end if    ▷ every leaf stores sets of objects eq. w.r.t. it
6:     for each child $w \in$ children($v$) do
7:         $P_w \leftarrow$ set of equivalent objects w.r.t. $\mathcal{G}_w$
8:         for each set $I \in P_w$, where $I \notin P$ do
9:             $\mathcal{E}_1, \ldots, \mathcal{E}_k \leftarrow$ sets in $P$ that intersect $I$
10:            Create $2k$ sets $O_1, \ldots, O_k$ and $N_1, \ldots, N_k$,
               where $O_i = \mathcal{E}_i \setminus I$ and $N_i = \mathcal{E}_i \cap I$
11:            $P \leftarrow \{O_1, \ldots, O_k, N_1, \ldots, N_k, \mathcal{E}_{k+1}, \ldots, \mathcal{E}_{|P|}\}$
12:        end for
13:    end for
14: until all nodes in $\mathcal{G}$ are explored
15: return $P$
```

sponding V. Once we have this information for every node in $\mathcal{G}$, we can then have an algorithm that starts by initially considering all objects as equivalent, and then recursively splits this set while ensuring that no two objects that are deemed non-equivalent by a node is part of the same set. Algorithm 2 describes how to do this. While it is possible to prove that this algorithm does produce a set of equivalent sets (although not necessarily of minimum size), a formal proof is beyond the scope of this paper.

A.2 Computing the hottest bin

While move types help in generating local changes, the more important question is: *what order of bins should one look for moves from?* To answer this, we introduce the notion of *hottest bin*. A bin is considered *hottest* when, given an objective and a current assignment, moving objects to or from this bin reduces the objective value the most. At a high-level, during each iteration we find the hottest (a.k.a. most broken) bin and attempt to fix it by making local changes as dictated by the *move types*, and continue the search until no progress can be made. Algorithm 3 describes the high-level local search algorithm used in Rebalancer. (Timeout handling has been omitted for simplicity.) Observe that there are three performance sensitive components in our algorithm, *i)* finding the hottest bin (line 7), *ii)* given a local change, evaluating the objective value (line 12), and *iii)* applying a local change (line 17). In the rest of this section, we describe each of these components in more detail.

Intuitively, the hottest bin is one that can potentially improve the most from local moves, but it is not obvious how to find such a bin. Prior work explored the concept of *bin potential* which, for a bin b, is the difference in the current objective value and the value of the objective after removing all objects in b [19]. Although a reasonable metric, it only works for objectives that can be improved by moving objects out of a bin, but not when objects need to be moved in, such as what is re-

Algorithm 3 Local Search Algorithm

Input: Objects $\mathbb{O}$, bins $\mathbb{B}$, expression graph $\mathcal{G}$, initial assignment $\mathcal{A}_0$

```
1: $\mathcal{A} \leftarrow \mathcal{A}_0$    ▷ set current assignment
2: $anyProgress \leftarrow$ true
3: while $anyProgress$ do
4:     $anyProgress \leftarrow$ false
5:     $explored \leftarrow \emptyset$
6:     while $explored \neq \mathbb{B}$ do
7:         $b_{hot} \leftarrow$ find_hottest_bin($\mathcal{G}$) $\notin explored$
8:         $currProgress \leftarrow$ false
9:         for $moveType$ in MoveTypes do
10:            $\mathcal{L} \leftarrow$ local changes using $b_{hot}$ and $moveType$
11:            for local change $\delta \in \mathcal{L}$ do
12:                obj$_\delta \leftarrow$ evaluate_changes($\mathcal{G}, \delta$)
13:                remove $\delta$ from $\mathcal{L}$ if it *violates any constraint* or *worsens objective*, i.e., obj$_\delta > 0$
14:            end for
15:            if $\mathcal{L}$ is not empty then
16:                $\delta^* \leftarrow \min_{\delta \in \mathcal{L}}$ obj$_\delta$    ▷ best local change
17:                apply_changes($\mathcal{G}, \delta^*$)
18:                $\mathcal{A} \leftarrow \mathcal{A} \oplus \delta^*$    ▷ update assignment
19:                $currProgress \leftarrow$ true
20:                $anyProgress \leftarrow$ true
21:                break
22:            end if
23:        end for
24:        if $currProgress$ is false, then add $b_{hot}$ to $explored$
25:    end while
26: end while
```

quired when enforcing minimum capacity. Moreover, finding a candidate bin can be expensive as it requires computing the contribution of every bin and taking the maximum.

The bin ranking algorithm used in Rebalancer exploits the expression graph $\mathcal{G}$ and works regardless of whether the objectives improve by moving objects in or out of them. Moreover, it does not need to compute the contribution of every bin and terminates as soon as the hottest bin has been established. Recall that each node $v \in \mathcal{G}$ directly (leaf nodes such as Lookup) or indirectly (internal nodes such as Max) affects a set S_v of bins. The idea is to process the leaf nodes of $\mathcal{G}$ in a *greedy order* of their contribution to the objective. This ordering of leaf nodes gives us a sequence of sets of bins $S_v, S_w, \ldots, S_z$ and we can use these sets to infer the *hottest bin*. Indeed, if each leaf node affected exactly one bin, the hottest bin would be the one corresponding to the first leaf in this ordering. However, leaf nodes such as ObjectLookup may affect many bins, so we need a way to break ties.

We do this by maintaining an initially empty list of *hottest candidates* in a data structure H called *incremental priority*

Algorithm 4 **find_hottest_bin**

```
1:  Incremental priority queue $H \leftarrow \emptyset$
2:  $iter \leftarrow$ objective root
3:  **if** valid cache exists, **then** restore $H$ and $iter$ from cache
4:  **while** $iter \neq end$ **do**
5:      $v \leftarrow$ node at $iter$
6:      **if** $H$ has a unique best element **then**
7:          **return** $\mathsf{top}(H)$ as the hottest bin
8:      **end if**
9:      **if** $v$ is a leaf or *does not need expansion* **then**
10:         $S_v \leftarrow$ affected bins at $v$
11:         $\mathsf{update}(H, S_v)$                    ▷ update queue $H$
12:     **end if**
13:     advance $iter$ to the next node in pre-order
14: **end while**
15: **return** $\mathsf{top}(H)$ breaking ties at random
```

queue. The items in H are bins whose priority is defined using a *series of sets* of descending priority. Given two bins $b, b' \in H$ and the series of sets associated with each, b has a higher priority than b' if it appears in a set before b' does. If both b, b' appear for the first time in the same set, then the second set breaks the tie, and if the second is also the same, then the third is used, and so on. The series of sets can be fed into the data structure one set S at a time with $\mathsf{update}(H,S)$. One implementation of this data structure is to maintain a map of bins with the indices of the sets in which they appear. This map is sorted by a custom *compare* function which orders the bins in the right way. For example, if we had three sets $\{1\colon \{b_p, b_q\}, 2\colon \{b_p, b_q, b_r\}, 3\colon \{b_p, b_r\}\}$, the sorted map will be $b_p\colon \{1,2,3\}, b_q\colon \{1,2\}, b_r\colon \{2,3\}$. In this case, once all the sets have been processed, b_p will be the hottest bin.

Algorithm 4 describes how we compute the hottest bin by traversing the expression graph $\mathcal{G}$ in *pre-order*. We start with the objective root and process children recursively in the order of decreasing potentials and exit as soon as the hottest bin is found (line 7). We also perform some natural optimizations to reduce the number of nodes traversed. For example, we do not recursively expand nodes that have achieved their bound values, and also do not expand if every node in the subgraph rooted at it affects the same set of bins (line 9). Observe that the algorithm saves the progress each time and if possible resumes from a valid cached state. If we invalidate the cache after applying each local progress, then the ordering of hot bins is *dynamic*, otherwise it is *static*. Indeed dynamic ordering often leads to a better solution quality (at the cost of recomputing the ordering every time), but there are cases when a static ordering is sufficient.

A.3 Evaluating and applying candidate solutions

The remaining two important components of our local search algorithm are *evaluating* and *applying* a set of moves δ. A naive way of evaluating or applying a change δ would be to just recompute the modified assignment $\mathcal{A}' = \mathcal{A} \oplus \delta$ and recompute the value of all nodes by a full recursive traversal. Indeed this is quite inefficient as it requires traversing and recomputing values for all nodes of the graph $\mathcal{G}$ when the number of nodes affected by the change δ is likely quite small. Since every node of the graph already stores its value w.r.t. to the current assignment, if we can find a way to identify the child nodes whose values need to be recomputed and combine them with values that were not modified, we can significantly speed up evaluate and apply operations. Observe that *applying* of moves updates the internal state of graph $\mathcal{G}$ (namely node values, bounds, ordering of children by their potential), whereas *evaluating moves* does not modify the graph. This distinction is important as it allows us to achieve even faster running times for *evaluate* operations. Below, we describe some ideas that make this possible.

Bottom-up propagation of changes.

Once the expression graph $\mathcal{G}$ is built, we can also preprocess the leaf nodes to build a map $\mathcal{M}_o$ from objects to the leaf nodes that reference them. Similarly, we build a map $\mathcal{M}_b$ from bins to the leaf nodes that they affect. Recall that each local change δ is a set of moves (o, b_s, b_d) that denotes moving object o from b_s to b_d. Given this, we can iterate over the set of moves in δ, and use the maps $\mathcal{M}_o, \mathcal{M}_b$ to compute a set of leaves L affected by the change δ. Starting from the leaves in L, we traverse the graph bottom-up (from leaves to the roots) using the *incoming edges* at every node. The set of nodes reachable in this way is precisely the minimal set of nodes whose values need to be recomputed. (See also Figure 2.)

Minimal computation at a node v.

While recomputing value of a changed node v, iterating over all the nodes in $\mathsf{children}(v)$ can be unnecessarily expensive especially when only a small number of child nodes may have been updated. Depending on the type of the expression node, we can store some additional information that makes these updates significantly faster. Here we give an example for the Max node; similar optimizations exist for other node types. For a Max node, we maintain a *sorted list of children* by their value. We first iterate over all the *updated child nodes* and take the maximum of their new values; suppose that this value is z_1. Next, we iterate over the list of sorted children nodes and stop at the first node that was not updated. Let the value of that node be z_2. Now, it is not hard to establish that the updated value of this Max node is $\max(z_1, z_2)$. Observe that as a result of this process, updating the value of this node took $O(|Z|)$ time, where $Z \subseteq \mathsf{children}$ is the set of updated children, as opposed to $\mathcal{O}(|\mathsf{children}(v)|)$. However, this comes at the cost of a more expensive apply operation where we will need to update the list of sorted children. This trade-off is acceptable because the number of evaluations is typically several orders of magnitude larger than the number of apply operations.

A.4 Local search exit conditions

Recall that our local search algorithm terminates when it can no longer make any progress. In some cases, depending on the values of the objectives, it might be possible to determine if any future moves can result in an improvement, and if not end the search early. Below, we briefly describe the notions of *global* and *local* optimality that are used for this purpose.

Global Optimality

Recall that we had recursively computed the lower bounds for each node in $\mathcal{G}$. If the current value of the objective root note has reached its estimated lower bound, we say that the current assignment is *globally optimal* and we can terminate our local search algorithm.

Local Optimailty

Observe that in Algorithm 3, we maintain a list of *explored* bins, which are the set of bins for which no improving move was found. We leverage this information to compute a new *constrained lower bound* for each expression. To do this, assume that all the bins in *explored* are frozen (can neither move objects in or out) and compute the new value of each expression. For example, consider a Lookup node v such that all its affected bins S_v are explored. Since future moves cannot improve the value of this node, we can establish that its *constrained lower bound* is its current value. Once the bound of all leaf nodes are updated, we can recursively compute the updated constrained bounds of all the expressions. If we establish that the value of the objective root node is the same as its constrained lower bound, then we say that the current assignment is *locally optimal*, and if this happens, then we can terminate the inner loop of our local search algorithm (lines 6-25 of Algorithm 3).

A.5 Numerical stability of incremental apply

Please refer to the discussion in the main text around incremental application of small changes in local search. Observe that another challenge that we need to tackle is numerical stability of *apply* operations. This is more important for nodes of the graph $\mathcal{G}$ such as *Sum* that can accumulate numerical errors by performing arithmetic operations on values of children nodes. For example, let ε be the numerical tolerance for satisfying a constraint. That is, two values are considered equal if they are within ε of each other. Now suppose applying every change incurred an error of say $\varepsilon' = 10^{-3}\varepsilon$, we would accumulate an error of $K\varepsilon'$ after applying K moves. Say if $K = 10^4$, would incur an error of 10ε which is enough to invalidate a valid constraint. Although some amount of precision loss is unavoidable, apply operation for expressions in Rebalancer are designed to minimize precision loss as much as possible. In some cases, as shown below, there is a trade off between running time and precision loss and we need to use a slightly advanced data structure to achieve both.

For example, consider the Sum, where we have two alternatives for recomputing its value for a given change δ. Note that any floating point arithmetic often results in some loss of precision.

- *Case 1 : Slow with small precision loss and numerical stability.* Compute the value dynamically after every change by summing the values of children. This takes $O(|\mathsf{children}|)$ time. Even though there is some precision loss in this process, the resulting value is the same as the values are added in the same order.

- *Case 2: Fast with high precision loss and numerical instability.* Let z_0 be the current value of the total sum, and z_p, z_n respectively be the sum of prior and new values of the *changed children*. Then the updated value of this node after applying the change is $z_0 + z_n - z_p$. Note that this takes $O(|\delta|)$ time but we would likely accumulate some additional precision loss by subtracting two approximate numbers z_n and z_p. Moreover, this problem is encountered per update, and it adds up across many incremental updates. Finally, this also results in numerical instability because the result depends on the order of applying updates.

We can address this problem by building and maintaining *segment tree* structure over the children values that supports computing sum of values in a given range as well as value updates in logarithmic time. Therefore applying the change δ takes $O(|\delta| \log |\mathsf{children}|)$ time but incurs smaller precision loss and better numerical stability.

Appendix B Additional Details on Rebalancer's MIP Based Solver

In the previous section, we described a local search based algorithm for solving assignment problems. However, for problems that are not too big or where the solution quality is extremely important, we can use commercial Mixed Integer Programming (MIP) solvers such as Gurobi [15] and Xpress [45]. As we will soon describe, here again we use the flexibility of the expression graph $\mathcal{G}$ to translate all or part of the problem to a MIP model, which in turn enables us to combine the strengths of MIP solvers with our local search algorithm for applications that need them.

B.1 On-the-fly translation to a MIP model

Recall that the standard MIP model for an assignment problem consists of *binary* assignment variables v_{ij} for every object bin pair (o_i, b_j). In Rebalancer, we reuse the notion of *equivalent objects* briefly described in Section A.1 to succinctly combine assignment variables of objects that belong to the same equivalence class. To see how, first, let $\mathbb{O}_d$ be the collection of equivalent sets of objects; each element $\mathbb{O}_d^i \in \mathbb{O}_d$ is a set of equivalent objects. Next, we introduce the notion of *dynamic bins*, denoted $\mathbb{B}_d$. A bin is dynamic if objects can move in or out of it. Note that depending on the constraints, some bins may not be able to do either (i.e., they are frozen). Given this, the assignment variables in our MIP model are defined as follows.

1. For each eq. object set $\mathbb{O}_d^i$ and dynamic bin $b_j \in \mathbb{B}_d$, create an *integer* variable v_{ij} that represents the number of objects of type i that are assigned to b_j. Set the lower bound of v_{ij} as 0 and upper bound as $|\mathbb{O}_d^i|$.
2. For each $\mathbb{O}_d^i$, add an *object integrality constraint*: $\sum_j v_{ij} = |\mathbb{O}_d^i|$ which ensures that all the equivalent objects in a set are assigned.

Note how the above reduces the number of variables from $\mathcal{O}(|\mathbb{O}| \cdot |\mathbb{B}|)$ to $\mathcal{O}(|\mathbb{O}_d| \cdot |\mathbb{B}_d|)$. In our experience, this optimization can sometimes be the difference between being able to solve the problem using a MIP solver and otherwise.

Given the assignment variables above, similar to *evaluate*, and *apply* operations, every Rebalancer expression implements a mipTranslate operation, which knows how to correctly translate the expression based on its type into a linear combination of assignment variables. Below we show a couple of examples (see [44] for details on translating many other expressions including some non-linear types).

- For a leaf node v of type Lookup, we can implement the mipTranslate similar to the native representation of util defined in Equation 1.
- For a node v of type Max, we can use standard techniques of translating a max function to MIP model. For simplicity, consider the easy case when v is minimized by some objective. In that case, we add a new variable z to the model, for all $w \in \mathsf{children}(v)$, add constraints $z \geq \mathsf{mipTranslate}(w)$, and return the expression z.

With mipTranslate operation of each node in place, we can build the MIP model M for the entire problem by recursively calling mipTranslate on the objective root obj_r and all the constraint roots, ctr_r^i. Once we have the MIP translation, the objective in the MIP model is to minimize $\mathsf{mipTranslate}(\mathsf{obj}_0)$ and the constraints are $\mathsf{mipTranslate}(\mathsf{ctr}_r^i) \leq 0$ for all ctr_r^i and the ones that are added during the mipTranslate calls on a node (like in the case of Max node described above).

B.2 Solving the model

We can use the aforementioned translation algorithm parameterized by dynamic bin set $\mathbb{B}_d$ to generate the MIP model and solve using commercial solvers. If the problem is small enough, we can solve the full problem with $\mathbb{B}_d = \mathbb{B}$. Otherwise, we can use the hottest bin ranking from Algorithm 4 to select an appropriately sized subset $\mathbb{B}_d \subset \mathbb{B}$ of dynamic bins and solve only part of the problem. This technique can in turn be useful if we want to combine our local search algorithm with MIP solvers.