CloudOptimizer: Multi-tenancy for I/O-Bound OLAP Workloads

Hatem A. Mahmoud[*]
UC Santa Barbara
hatem@cs.ucsb.edu

Hyun Jin Moon[†]
Google Inc.
hyunm@google.com

Yun Chi
NEC Laboratories America
ychi@nec-labs.com

Hakan Hacıgümüş
NEC Laboratories America
hakan@nec-labs.com

Divyakant Agrawal
UC Santa Barbara
agrawal@cs.ucsb.edu

Amr El-Abbadi
UC Santa Barbara
amr@cs.ucsb.edu

ABSTRACT

Consolidation of multiple databases on the same server allows service providers to save significant resources because many production database servers are often under-utilized. Recent research investigates the problem of minimizing the number of servers required to host a set of tenants when the working sets of tenants are kept in main memory (e.g., in-memory OLAP workloads, or OLTP workloads), thus the memory assigned to each tenant, as well as the I/O bandwidth and CPU time, are all dictated by the working set size of the tenant. Other research investigates the reverse problem when the number of servers is fixed, but the amount of resources allocated to different tenants on the same server needs to be configured to optimize a cost function. In this paper we investigate the problem when neither the number of servers nor the amount of resources allocated to each tenant are fixed. This problem arises when consolidating OLAP workloads of tenants whose service-level agreements (SLAs) allow for queries to be answered from disk. We study the trade-off between the amount of memory and the I/O bandwidth assigned to OLAP workloads, and develop a principled approach for allocating resources to tenants in a manner that minimizes the total number of servers required to host all tenants while satisfying the SLA of each tenant. We then explain how we modified InnoDB, the storage engine of MySQL, to be able to change the amount of resources allocated to each tenant at runtime, so as to account for fluctuations in workloads. Finally, we evaluate our approach experimentally using the TPC-H benchmark to demonstrate its effectiveness and accuracy.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Measurement, Algorithms, Experimentation

[*]Work partially done while the author was at NEC
[†]Work done while the author was at NEC

1. INTRODUCTION

Workload consolidation is an effective method to achieve cost efficiency in cloud computing. The consolidation of multiple small workloads – also known as multi-tenancy – could avoid significant resource waste given that many production servers are often underutilized [3]. Database services in the cloud benefit from the same principle through database multi-tenancy by co-locating multiple databases on the same server. In this paper, we investigate the problem of allocating resources to tenants and assigning tenants to servers in a multi-tenant environment. Each tenant is associated with a service-level agreement (SLA) that defines constraints on throughput, latency, or both. Violation of such constraints results in SLA penalties, and – even worse – client dissatisfaction. Our objective is to develop a principled approach that minimizes the number of servers required to host a set of tenants, thus provides cost efficiency, while satisfying the SLA of each tenant.

Recent research investigates the two problems of resource allocation and server placement separately; that is, either to find an efficient tenant placement when the amount of resources allocated to each tenant is fixed [17, 6], or to find an efficient resource allocation when the tenant placement is fixed [20, 21, 18]. We take a different approach by studying the trade-off between the different resources allocated to tenants, mainly memory versus I/O. Then we develop a principled approach that takes advantage of such a trade-off so as to assign resources to tenants in a manner that makes tenant placement more efficient. We focus on I/O-bound OLAP workloads since previous research has investigated tenant placement for in-memory OLAP workloads [17], as well as OLTP workloads [6]. An example of I/O-bound OLAP workloads is report generating queries over large data sets. Previous research on multi-tenancy that deals with SLAs focuses on either latency SLAs only [17, 21], or throughput SLAs only [6]. To the best of our knowledge, this is the first multi-tenancy solution that allows each user to choose among those two types of SLAs.

The problem of multi-tenancy for I/O-bound OLAP workloads turns out to be closely related to the well-known "5-minute rule" that was proposed by Gray et al. [11, 10] and later brought up-to-date by Graefe [9]. According to the 5-minute rule, under the given costs for memory versus I/O bandwidth, a data item can be served either in-memory (memory-resident) or through I/O access (disk-resident), resulting in different costs; furthermore, there exists a break-even frequency of access that separates the regions where one choice is better than the other. In our work, we extend this principle in the following two directions. First, we demonstrate that in handling an OLAP workload (instead of a single data item [11] or certain sequential SQL operations [10, 9]), there exists a continuous spectrum of configurations which we can exploit, in addition to the choices of 100% memory-resident and 100% disk-resident. For example, by increasing the amount of memory dedicated to a workload, we can trade off a portion of the I/O bandwidth required by the workload. Second, instead of optimizing for a single workload, we study how to place and configure a set of tenants in a manner that minimizes the number of servers required to host all tenants. Here, the challenge for service providers is that all tenants have to be considered together in order to achieve globally-optimum solutions.

Our problem poses many practical challenges. First, in order to minimize the total number of servers we need to configure all tenants in a single optimization problem; however, in order for the solution to be practical, we have to configure tenants in an incremental fashion as this is how tenants would join a cloud service in practice. We deal with this challenge by developing two algorithms for optimizing resource allocation: an offline algorithm named Greedy Resource Optimizer (GRO) that provides an approximation guarantee, and a heuristic named Balanced Resource Optimizer (BRO) that lacks an approximation guarantee but allocates resources to tenants online. By comparing the empirical performance of BRO against GRO, we verify that BRO performs almost the same as GRO for typical OLAP workloads. Another challenge for the solution to be practical is that SLAs must allow a tenant to define different throughput and latency requirements for different time periods (e.g., working hours and evening), perhaps with different costs. This is because the main objective of cloud computing is to avoid over-provisioning. Thus, our solution must account for workload fluctuations. We deal with this challenge by generalizing GRO and BRO to take workload fluctuations into account, and by introducing changes into InnoDB, the storage engine of MySQL, so that we can change the memory size assigned to a database at runtime.

The CloudOptimizer system that we present in this paper is a component of our comprehensive data management platform for the cloud, CloudDB [12]. We summarize our contributions in this paper as follows.

- We propose a profiling approach to capture the trade-off between the amount of memory and the I/O bandwidth needed by a given tenant in order to meet its SLAs.
- We develop an approximation algorithm, called Greedy Resource Optimizer (GRO), to approximate a globally-optimum resource allocation that assigns resources to tenants in a manner that optimizes the total number of servers needed to host all tenants.
- We develop an online heuristic alternative to GRO, called Balanced Resource Optimizer (BRO). Although BRO lacks an approximation guarantee, by comparing BRO against GRO we show empirically that BRO performs well for typical OLAP workloads.
- We explain how we modified InnoDB, the storage engine of MySQL, so as to allow for the amount of resource allocated to a given tenant to change at runtime as the workload fluctuates.
- We conduct an experimental study to demonstrate the effectiveness of our approach, compared to other baseline heuristics, and to evaluate the accuracy of our profiling approach.

The rest of this paper is organized as follows. In Section 2 we survey related work. In Section 3 we give an overview of our system architecture. In Section 4 we describe how we profile tenants under different resource configurations. In Section 5 we detail the algorithms and heuristics that we develop for optimizing resource allocation. In Section 6 we explain how we control the amount of resources allocated to different tenants on the same server using our modified version of MySQL. Finally, we present experimental studies in Section 7 and give conclusion in Section 8.

2. BACKGROUND AND RELATED WORK

Multi-tenant databases have become an important research area recently as more users move their databases to the cloud [14, 7]. In this section we discuss prior research that is related to the various aspects of our work.

2.1 Problem Definition

The two problems that we deal with in this paper, namely (i) tenant placement and (ii) resource allocation, have been investigated separately in prior research. One direction of prior research investigates the problem of tenant placement, with the objective of minimizing the number of servers, given a fixed amount of resources allocated to each tenant. That type of problem arises when the working sets of tenants are kept in main memory, as in the case with in-memory OLAP workloads [17], and OLTP workloads [6]. When the working set of each tenant has to be maintained in main memory, the amount of resources allocated to each tenant (e.g., RAM, CPU, I/O) is dictated by the working set size as well as the workload of that tenant. So the challenge is to find a packing of tenants that minimizes the number of servers given the amount of resources already allocated to each of the tenants. Another direction of prior research investigates the reverse problem; that is, given a constant number of servers, with tenants already assigned to servers, the objective is to allocate resources to tenants in a manner that optimizes a particular cost function; for example, to maximize database performance [20], to minimize SLA penalties [21], or to minimize energy consumption [18]. In this paper, neither the number of servers, nor the amount of resources allocated to each tenant, are constants. We analyze the trade-off between different resources, mainly RAM versus I/O, to allocate resources to tenants in a manner that minimizes the total number of servers.

Prior research on multi-tenancy that considers service-level agreements (SLAs), or service-level objectives (SLOs), as constraints on the optimization problem, focuses on either throughput constraints or latency constraints. For example, TREX [17] and SmartSLA [21] are two systems that consider latency constraints; that is, the service provider guarantees that the response time of each query (or most of the queries) does not exceed a pre-specified upper bound, and agrees to pay penalties whenever such guarantee is not met. Meanwhile, Kairos [6] is an example of a system that considers throughput constraints; that is, the service provider agrees to handle the queries of a client at a pre-specified rate. That rate fluctuates from one time period to another based on the needs of the client. The client either states their throughput constraints to the service provider for each time period explicitly, or provides the service provider with traces of previous runs of the database in order to clarify the desired throughputs at different periods of time. In this paper, we consider both types of SLAs; that is, the client can specify an upper bound on latency, a lower bound on throughput, or both, and the SLA may specify multiple bounds for different time periods.

Some of the prior research that is related to database multi-tenancy focuses on consolidation of large numbers of almost-inactive tenants by sharing the same schema among tenants (e.g., [4, 13, 15, 19]). The main challenges in that type of systems are (i) scalability [4, 13], due to the limit on the number of tables a DBMS can handle for a given schema, as well as (ii) security [15, 19], because tenants share the same schema. This paper, in comparison, targets workloads with non-trivial throughput requirements.

2.2 Solution Properties

Prior research that tackles the problem of optimizing tenant placement can be classified into offline or online solutions. Offline solutions (e.g., [6]) compute tenant placement for all tenants in a single, large optimization problem. Online solutions (e.g., [17]) make tenant placement decisions, one tenant at a time. Online solutions are more practical because tenants typically register for a cloud service one tenant at a time, not in a single, one-time batch. In this paper, we develop an online approach, namely Balanced Resource Optimizer (BRO), that computes a resource allocation to optimize tenant placement, for one tenant at a time. To verify that BRO minimizes the number of severs required to host all tenants, we also develop an offline approximation algorithm with an approximation guarantee, namely Greedy Resource Optimizer (GRO), that minimizes the total number of servers by computing a resource allocation for all tenants in a single optimization problem. By comparing the empirical performance of BRO against GRO, we demonstrate that BRO achieves almost the same results as GRO for typical OLAP workloads.

Previous research on multi-tenancy can be also classified based on whether the solution relies on database profiling (e.g., [6]) or on predictive models (e.g., [21]). Although predictive models require less pre-processing, they are heavily dependent on the underlying infrastructure, such as the implementation of the database management system, and the underlying hardware. We do introduce changes into the underlying infrastructure to enable our solution, mainly by modifying the storage engine of MySQL. However, we opt to use a profiling approach that treats the underlying infrastructure as a black box to ensure that our solution is portable. To mitigate the cost of pre-processing, we design our profiler in a way that that the profiling does not incur notable disruptions to the normal operations of the database that is being profiled.

2.3 Level of Multi-tenancy

Multi-tenant database systems can be classified [16] based on the level of resource sharing into (i) shared tables (e.g., [4, 13]), (ii) shared process with independent tables (e.g., [17, 6]), (iii) shared OS with independent processes, and (iv) shared hardware with independent VMs (e.g., [20, 21, 18]). In shared-table systems, the service provider hosts a configurable schema that is designed for a specific domain (e.g., customer relationship management [1]), and allows different tenants to choose tables and attributes based on their application-specific needs. Since we target general-purpose OLAP, rather than a particular domain, we opt not to use shared-table multi-tenancy. Multi-tenant database systems that use shared processes with independent tables are not restricted to a single domain, but are more suitable for the case when the entire working sets of tenants reside in main memory, as the case with TREX [17] and Kairos [6]. In such a case, because each tenant has enough main memory for its entire data set, the tenants do not compete over buffer pages, or greedily evict the pages of each others. We run tenants on independent database processes on the same OS in order to impose hard boundaries on the amount of resources allocated to each tenant, while avoiding the overhead of running multiple virtual machines on the same server. This allows us, for example, to force a tenant that has low throughput SLA to answer queries from disk, by assigning less memory to that tenant, so as to leave more memory space for another tenant that has higher throughput SLA. Curino et al. [6] list some advantages of shared DBMS processes over independent DBMS processes and independent VMs when implementing multi-tenancy for OLTP workloads. However, none of those points apply to our case. First, using a shared database process does not save us memory because we do not use virtual machines, and because the Linux kernel loads only a single copy of a program into main memory even when we run the same program multiple times; thus all DBMS processes on a single server share the same copy of the DBMS in main memory. Second, OLAP workloads do not benefit from group logging optimizations that are implemented in some database systems, because insertions in OLAP databases already happen in large batches. Third, in an I/O-bound OLAP workload, the overhead of context switching between database processes is negligible compared to I/O latencies.

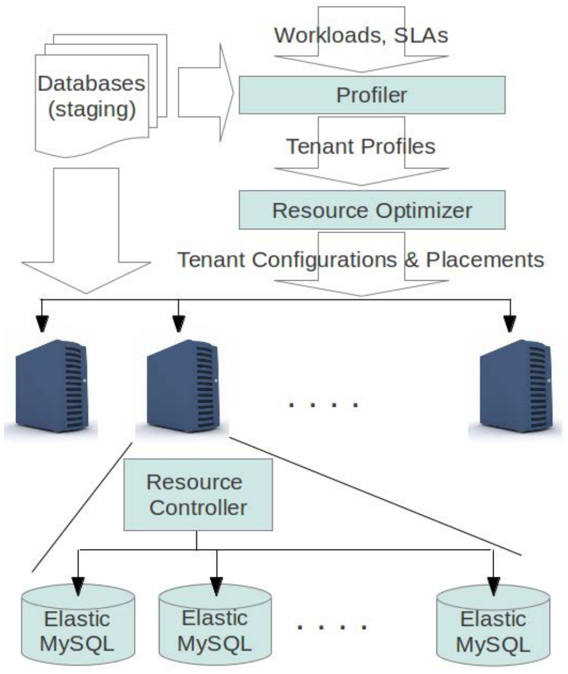

Figure 1: CloudOptimizer overview

3. CLOUD-OPTIMIZER OVERVIEW

We consider the problem of tenant configuration and placement for I/O-bound OLAP workloads. The objective is to allocate resources for I/O-bound OLAP tenants, and pack them into servers in a manner that minimizes the total number of servers needed to host all tenants, while satisfying the SLA requirements of each tenant. The SLA requirements of tenants are expressed as lower bounds on throughput, upper bounds on latency, or both. Latency SLAs are typically specified explicitly by the user, while throughput SLAs are determined either explicitly by the client, or implicitly based on the performance of the database before consolidation. Workloads fluctuate over time in different ways depending on the domain of each tenant. For example, a database that serves ad-hoc analytical queries has higher query arrival rates during working hours, then the query arrival rate drops during the evening, and drops further during night. Such a database is also expected to have low latency SLA requirements. Meanwhile, a database that serves automated report-generation queries may have its highest query arrival rate after working hours, and typically does not have tight latency SLAs.

Initially, when a tenant moves its database to the cloud environment, the tenant database first runs on a private server that is referred to as a *staging* server. A staging server is identical to the servers that host consolidated tenants, with the exception that a staging server hosts only one tenant, and there is a *profiler* installed on each staging server to measure the resource utilization of the tenant database. The profiler utilizes another component of the system, the *resource controller*, to gradually reduce the amount of resources allocated to the tenant database in order to figure out the minimum amount of resources that the tenant can operate with while satisfying its SLAs. In particular, the profiler examines the trade-off between the amount of memory and the I/O bandwidth assigned to the tenant. The profiler ceases to reduce resources, and reverses any changes as soon as the performance of the database starts to drop below its SLA, so as to avoid disrupting the normal operation of the database. The profiler repeats this profiling process for each time period for which the client specifies different SLA requirements.

The output of the profiling phase is a set of profiles that characterize the performance of the tenant database at different time periods of the day. The profile of a particular time period determines, for each possible memory size that can be assigned to the tenant database, the minimum I/O bandwidth that is required to satisfy the SLA of the tenant, as well as the appropriate share of CPU time. These profiles are passed as input to another component of the system, the *resource optimizer*, which determines the optimum amount of each resource that needs to be allocated to the tenant in order to satisfy the SLA of the tenant while minimizing the total number of servers required to host all tenants. The output of the resource optimizer is a resource allocation vector that specifies the amount of each resource that should be allocated to the tenant database for different time periods of the day. This resource allocation vector is used as input to a vector packing solver that determines what other tenants this tenant should be co-located with in order to achieve optimum packing.

Once the tenant database is assigned to a particular server, a *resource controller* takes responsibility of adjusting the amount of server resources allocated to that tenant at different times of the day based on the resource allocation vector provided by the resource optimizer. The resource controller makes use of a version of MySQL that we modified to allow for the buffer pool size of the database to be changed at runtime without having to restart the database process. Besides, the resource controller makes use of standard Linux administrative tools, like `nice` and `ionice`, to divide CPU time and I/O bandwidth among tenants on the same server.

Figure 1 illustrates the design of our system. The main components of the system can be summarized as follows.

- A **Profiler:** monitors the performance of the tenant database under different resource configurations and at different times of the day, and generates a profile that captures the trade-off between different resources at different time periods of the day.
- A **Resource Optimizer:** takes as input the set of profiles generated by the profiler for each tenant database, and computes the optimum amount of each resource that needs to be allocated to the tenant database. The objective is to satisfy the SLA of the tenant while minimizing the total number of servers that is required to host all tenants.
- A **Resource Controller:** adjusts the amount of resources allocated to the tenant database at different times of the day. The main purpose is to account for workload fluctuations, based on the resource allocation plan that is generated by the resource optimizer, and relying on our modified version of MySQL.

The following three sections explain each of these three components in details.

4. PROFILER

The profiler uses the resource controller (explained in Section 6) to gradually decrease the amount of resources allocated to a database, one resource at a time, while the database operates on a staging server, to examine the trade-off between different resources. Mainly, the profiler captures the trade-off between the memory size and the I/O bandwidth allocated to the database. The CPU time needed by an I/O-bound OLAP workload is seldom the main bottleneck, compared to the I/O bandwidth that is needed by that workload. We observed this by profiling several OLAP workloads that we construct from the TPC-H benchmark, with different query mixtures, and different query arrival rates. Thus, the profiler, as well as other components of the system, set the percentage of CPU time that is allocated to a tenant to the percentage of I/O bandwidth that is allocated to that tenant, as a proper upper bound. The profiler starts with the most generous configurations, by allocating 100% of the I/O bandwidth and CPU time of the staging server to the tenant database, and by setting the buffer pool size of the database to the size of the entire data set. The profiler reduces the I/O bandwidth and CPU time gradually while measuring throughput and latency of the workload. Once the database performance starts to fall short of the SLA requirements, the profiler immediately resets the I/O bandwidth and CPU time to 100%, then repeats the profiling process for a smaller memory size, and so on. The profiler stops once the memory size allocated to the tenant does not satisfy the SLA even with 100% I/O bandwidth and CPU time. If the SLA of the tenant specifies different throughput or latency requirements for different time periods, the profiler repeats the profiling process for each of those time periods.

Figure 2 shows an example of a tenant whose throughput is measured under different memory sizes and different I/O bandwidths, for a particular time period. The workload consists of a stream of TPC-H queries targeting a data set of size 1 GB. The dashed line in Figure 2 (top) indicates a lower bound on throughput, as specified by the SLA of the tenant. In this example, the profiler starts with a memory size of 1000 MB, which is the size of the data set. The profiler keeps decreasing the memory size, 100 MB at a time, down to 500 MB, beyond which the throughput goes below the SLA lower bound even with 100% I/O bandwidth. For each memory size, the profiler starts with 100% I/O bandwidth and 100% CPU time, and keeps decreasing this percentage, 20% at a time, while measuring the throughput of the tenant for different I/O bandwidths and CPU times, until the throughput goes below the SLA lower bound. The granularities of resource allocation (in this example, 100 MB of

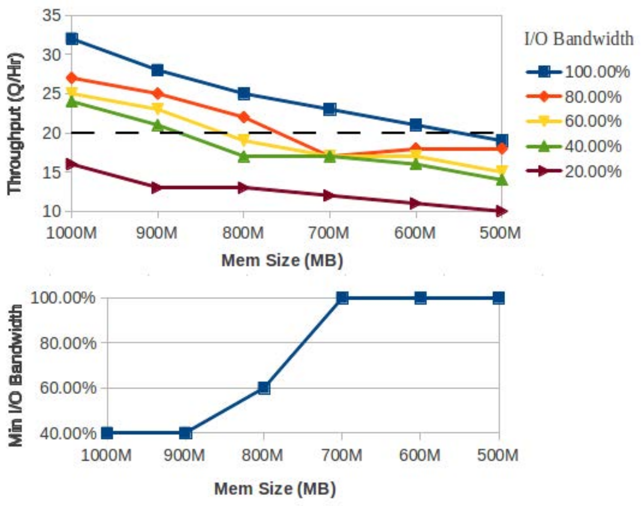

Figure 2: Example profile.

RAM, and 20% of I/O bandwidth and CPU time) are configurable parameters of the profiler.

When the profiler sets the I/O bandwidth and CPU time of a tenant to a percentage less than 100%, the profiler runs *dummy* workloads in the background, and allocates the remaining I/O bandwidth and CPU time to those dummy workloads, to emulate the state when other tenants run concurrently on the same server. The profiler runs a dummy workload in the background for each available granularity of I/O bandwidth and CPU time. For example, when the tenant that is being profiled is assigned 40% of the I/O bandwidth and CPU time, the profiler runs three dummy workloads in the background, each has 20% of the I/O bandwidth and CPU time. Each dummy workload consists of a stream of single-table scans running continuously on a separate data set, to ensure that the I/O bandwidth allocated to the dummy workload is fully utilized. Each dummy workload also performs calculations on each row that it scans, to consume all the CPU time that is available for it. The profiler sets the buffer pool sizes of the database processes that run dummy workloads to negligible sizes, and periodically cleans the I/O buffers of the Linux kernel, to ensure that the dummy workloads answer their queries from disk rather than from main memory. Moreover, the profiler disables I/O prefetching for the databases that run dummy workloads, because prefetching is typically implemented as asynchronous reads that the Linux kernel schedules with lower priority, so they do not compete properly against the tenant over the available I/O bandwidth.

Algorithm 1 summarizes the profiling procedure. The goal of profiling is to compute for each possible memory size, the minimum I/O bandwidth that is required by the tenant to satisfy its SLA, as shown in Figure 2 (bottom). Alternatively, we may compute for each I/O bandwidth the minimum memory size required to satisfy the SLA; both approaches lead to the same results when we run resource optimization on tenant profiles, thus for the rest of this paper we assume without loss of generality that the x-axis corresponds to memory size. The profiler generates a profile for each time period for which the tenant has different SLA requirements.

5. RESOURCE OPTIMIZER

Algorithm 1 Profiler

```
 1: for each time period for which the tenant has different SLAs
 2:     Set tenant buffer pool size to data set size
 3:     Allocate 100% of I/O and CPU to tenant
 4:     do
 5:         do
 6:             Deduct a granularity of I/O and CPU from tenant
 7:             Run a dummy workload in the background
 8:             Measure latency and throughput
 9:         while SLA constraints are satisfied
10:         Allocate 100% of I/O and CPU to tenant
11:         Kill all dummy workloads
12:         Clean the I/O buffers of OS
13:         Deduct a granularity of memory from tenant
14:         Measure latency and throughput
15:     while SLA constraints are satisfied
16: end for
```

In this section we explain the resource optimizer that is the core component of our system. We begin by formulating the problem that this component solves, then present our algorithmic and heuristic solutions to this problem.

5.1 Problem Formulation

After profiling a tenant t_i, the profiler outputs a finite set of mappings, $P_i = \{P_i^1, P_i^2, ...\}$, one mapping for each time period for which the tenant specifies different SLA requirements. For example, if the tenant t_i specifies two lower bounds on throughput, one for working hours and one for the rest of the day, the profiler outputs two mappings P_i^1 and P_i^2. Each mapping $P_i^j(m)$ returns the minimum I/O bandwidth that needs to be allocated to t_i in order to satisfy its SLA for the time period j when the memory size of t_i equals m. The value of m must be divisible by the granularity of memory allocation, μ, which we set to 100 MB in our experiments. The value of $P_i^j(m)$ indicates the CPU time as well, because we tie CPU time to I/O bandwidth. The mappings that are generated by the profiler are passed to the resource optimizer, which picks for each time period j a single memory size, and consequently a single I/O bandwidth and CPU time, in a manner that ensures that the total number of servers that are needed to host all tenants is minimized.

The resource optimizer divides the day into Z standard time periods. A reasonable value for Z is 12, thus each standard time period is two hours. The resource optimizer then transforms the set of profiles P_i of the tenant t_i into a new set P_i' that contains exactly Z mappings, one mapping for each standard time period. The mapping $P_i'^j$ is computed by finding out which of the original time periods that are provided by the tenant t_i contains the j^{th} standard time period; if the j^{th} standard time period overlaps with more than one of the original time periods provided by the tenant, we pick the original time period that has tighter SLA requirements. (Alternatively, we may profile tenants for the standard time periods instead of their original time periods to avoid this overlapping case). With this transformation, the output of the resource optimizer for any tenant t_i is a $2Z$-dimensional resource allocation vector A_i that specifies, for each of the Z standard time periods, the amount of memory and the I/O bandwidth allocated to t_i. That is, let m_i^j be the optimum memory size of the tenant t_i at the j^{th} standard time period, then $A_i = [m_i^1, P'^1_i(m_i^1), ..., m_i^Z, P'^Z_i(m_i^Z)]$.

The resource optimizer needs to pick A_i such that A_i packs well with the resource allocation vectors of other tenants, so as to minimize the total number of servers. Consider the offline case when the profiles of all tenants are given to the resource optimizer as one-time input. Let the set of resource allocation vectors of all tenants be $\mathcal{A} = \{A_1, A_2, ..., A_n\}$, where n is the number of tenants. To pack tenants into servers, we pass $\mathcal{A}$ as input to multi-dimensional vector packing [8]. Since our objective is to minimize the total number of servers, the resource optimizer should ouput a set of resource allocation vectors $\mathcal{A}^*$ that minimizes the optimum output of multi-dimensional vector packing. That is, let $Opt\_kDVP(\mathcal{A})$ be an optimum solution to multi-dimensional vector packing, then $\mathcal{A}^* = arg\_min_{\mathcal{A}}\{Opt\_kDVP(\mathcal{A})\}$. Since multi-dimensional vector packing is an NP-hard problem in the strong sense (i.e., the worst case running time is exponential in the value of the input), computing $\mathcal{A}^*$ is NP-hard as well. However, there are polynomial-time approximation algorithms for multi-dimensional vector packing with guarantees on the worst case ratio between the approximate solution value and the optimum solution value; such ratios are called approximation ratios. For approximation algorithms of NP-hard problems, worst case approximation ratios are typically guaranteed by proving a worst case ratio between the approximate solution value and a lower bound on the optimum solution value (rather than the optimum solution value itself, which is hard to characterize). The lower bound that is used in this kind of guarantees is usually an infeasible solution value that can be computed efficiently for each instance of the hard problem, and is guaranteed to be less than the optimum solution value.

Let $Aprx\_kDVP$ be an approximation algorithm for multi-dimensional vector packing with a worst case approximation ratio of ρ, and let Lwr_kDVP be the lower bound that is used to prove the approximation ratio of $Aprx\_kDVP$, thus $Aprx\_kDVP(\mathcal{A})/Lwr\_kDVP(\mathcal{A}) \leq \rho$, for any $\mathcal{A}$. Since $Opt\_kDVP(\mathcal{A}) \leq Aprx\_kDVP(\mathcal{A})$, therefore $Opt\_kDVP(\mathcal{A})/Lwr\_kDVP(\mathcal{A}) \leq \rho$, for any $\mathcal{A}$. Let $\hat{\mathcal{A}}$ be a memory assignment that minimizes $Lwr\_kDVP()$; that is, $\hat{\mathcal{A}} = arg\_min_{\mathcal{A}}\{Lwr\_kDVP(\mathcal{A})\}$. Thus $Lwr\_kDVP(\hat{\mathcal{A}}) \leq Lwr\_kDVP(\mathcal{A}^*)$. Consequently,

$$\begin{aligned} Opt\_kDVP(\hat{\mathcal{A}}) &\leq \rho \cdot Lwr\_kDVP(\hat{\mathcal{A}}) \\ &\leq \rho \cdot Lwr\_kDVP(\mathcal{A}^*) \\ &\leq \rho \cdot Opt\_kDVP(\mathcal{A}^*) \qquad (1) \end{aligned}$$

For our resource allocation problem, a possible lower bound can be computed by allowing tenants to span multiple servers. To demonstrate this, let us first formulate an integer linear program (ILP) for our resource allocation problem as follows. We refer to the following ILP formulation as ILP-1.

$$\begin{array}{lll} & min \sum_j z_l & \\ s.t. & \sum_l x_{il} = 1 & \forall i \\ & x_{il} \leq z_l & \forall i, l \\ & \sum_i x_{il} \cdot A_i[j] \leq z_l & \forall l, j \\ & x_{il} \in \{0, 1\} & \forall i, l \\ & z_l \in \{0, 1\} & \forall l \end{array}$$

$x_{il} = 1$ if and only if tenant t_i is hosted by server l, and $z_l = 1$ if and only if server l hosts at least one tenant. Since the number of servers needed to host all tenants is no more than the number of tenants, therefore $1 \leq i, l \leq n$, where n is the number of tenants. Also, $A_i[j]$ is the value of the j^{th} dimension of the vector of tenant t_i, so $1 \leq j \leq 2Z$. Solving ILP-1 gives an optimum solution to our tenant placement multi-dimensional vector packing problem, however it is NP-hard. Relaxing ILP-1 by replacing the integer constraints (i.e., the last two constraints) with non-negativity constraints (i.e., $x_{il}, z_l \geq 0, \forall i, l$) turns ILP-1 into a linear program (LP) that can be solved in polynomial time. We refer to the linear program that results from this relaxation as LP-1. The solution to LP-1 allows a tenant to span multiple servers, thus it is infeasible for our tenant placement problem. However, the solution value of LP-1 is a lower bound of the solution value of ILP-1. From [5], the maximum ratio between the solution value of ILP-1 and the solution value of LP-1 is $lg(2Z)$. We define $Opt\_kDVP(\mathcal{A})$ as the solution value of ILP-1, and $Lwr\_kDVP(\mathcal{A})$ as the solution value of LP-1, thus $Opt\_kDVP(\mathcal{A}) \leq lg(2Z) \cdot Lwr\_kDVP(\mathcal{A})$. By substituting $\rho = lg(2Z)$ in Equation 1 we get

$$Opt\_kDVP(\hat{\mathcal{A}}) \leq lg(2Z) \cdot Opt\_kDVP(\mathcal{A}^*) \qquad (2)$$

We focus on finding a resource allocation $\hat{\mathcal{A}}$ that minimizes $Lwr\_kDVP()$. From [5], the solution value of LP-1 equals $\max_j\{\sum_i A_i[j]\}$. Therefore $\hat{\mathcal{A}} = arg\_min_{\mathcal{A}}\{\max_j\{\sum_i A_i[j]\}\}$ Finding $\hat{\mathcal{A}}$ provides us with an input to multi-dimensional vector packing, such that when multi-dimensional vector packing is solved optimally we get an output number of servers that is no more than $lg(2Z)$ times the minimum number of servers required to host the tenants. In the next section we present an offline greedy algorithm that approximates $\hat{\mathcal{A}}$ with an absolute error of no more than 1 in pseudo-polynomial running time (i.e., the running time is a polynomial function in the number of tenants), then we show in Section 5.3 how we derive a heuristic alternative that solves the same problem in an online manner (i.e., allocates resources to tenants, one tenant at a time).

5.2 Greedy Resource Optimization (GRO)

In this section we present an offline greedy algorithm, namely Greedy Resource Optimizer (GRO), that approximates $\hat{\mathcal{A}}$ with an absolute error of no more than 1; that is, GRO generates a set of resource allocation vectors $\dot{\mathcal{A}}$ such that $Lwr\_kDVP(\dot{\mathcal{A}}) \leq Lwr\_kDVP(\hat{\mathcal{A}}) + 1$. The high-level idea of GRO is as follows. Consider each standard time period as a separate problem. For each standard time period, begin by assigning maximum memory size to each tenant, then proceed in iterations. Each iteration picks a tenant whose memory size can be reduced while incurring a minimum increase in I/O. Continue until the number of servers needed to provide the total memory size of all tenants, and that needed to handle the total I/O of all tenants, reach a *balanced* state.

Algorithm 2 lists our Greedy Resource Optimizer (GRO) algorithm. GRO solves a separate optimization problem for each standard time period j, since the profile of each standard time period is completely independent of the profiles of other time periods. Let μ denote the granularity of memory allocation. Thus, server memory size, and tenant memory sizes are all multiples of μ. A reasonable order of magnitude for μ is 100MB. Let m_i^{min} denote the smallest memory size with which the tenant t_i can satisfy its SLA for a particular time period. Also, let m_i^{max} denote the data size of t_i if the data size of t_i fits in the RAM of a single server, or the entire server memory size otherwise. For each standard time period, GRO begins by assigning to each tenant t_i its maximum memory size, m_i^{max}. Since the initial total memory size is maximum, and since P_i^j is a monotonically non-increasing function in memory size, therefore the initial total I/O of all tenants is minimum. After the initialization phase, the algorithm proceeds in iterations such that each iteration decreases the memory size of a single tenant. More specifically, in each iteration l, GRO picks a

Algorithm 2 Greedy Resource Optimization (GRO)

```
1: for $j = 1$ to $Z$ do
2:     input: Profiles of the $j^{th}$ time period, $\{P_i^j(m) : \forall t_i, j, \forall m, m_i^{min} \leq m \leq m_i^{max}, m/\mu \in \mathbb{Z}\}$
3:     Set $l \leftarrow 0$, and assign maximum memory to each tenant $m_i^j(l) \leftarrow m_i^{max}\ \forall t_i$
4:     while $\sum_i m_i^j(l) > \sum_i P_i^j(m_i^j(l))$ and $\exists i : m_i^j(l) > m_i^{min}$ do
5:         Define the average I/O increase incurred by a memory deduction as $\bar{P}_i^j(l,\delta) = (P_i^j(m_i^j(l) - \delta) - P_i^j(m_i^j(l)))/\delta$
6:         Cheapest chunk to deduct from $m_i^j$ is $\delta_i(l) = arg\_min_\delta\{\bar{P}_i^j(l,\delta) : m_i^j(l) - \delta \geq m_i^{min}, \delta/\mu \in \mathbb{Z}^+\}$, if tied pick largest $\delta$
7:         Pick the tenant with the cheapest, cheapest deductable memory chunk $i(l) = arg\_min_i\{\bar{P}_i^j(l, \delta_i(l))\}$, break ties arbitrarily
8:         Deduct that memory chunk from the tenant, $m_{i(l)}(l+1) \leftarrow m_{i(l)}(l) - \delta_{i(l)}(l), m_j(l+1) \leftarrow m_j(l)\ \forall j \neq i(l),\ l \leftarrow l+1$
9:     end while
10:    if $\max\{\sum_i m_i^j(l), \sum_i P_i^j(m_i^j(l))\} > \max\{\sum_i m_i^j(l-1), \sum_i P_i^j(m_i^j(l-1))\}$ then
11:        Revert last interation by setting $m_{i(l)}(l) \leftarrow m_{i(l)}(l-1)$
12:    end if
13: end for
14: Let $\dot{\mathcal{A}} = \{[m_i^j(l), P_i^j(m_i^j(l))] : \forall i, j\}$
15: return $\dot{\mathcal{A}}$
```

tenant $t_{i(l)}$ whose current memory size $m_{i(l)}(l)$ can be decreased by some amount of memory $\delta_i(l)$ while incurring a minimum average increase in I/O per unit memory decreased; that is, the cheapest available chunk of memory that can be removed from a single tenant. GRO finishes a standard time period either after the total memory size needed by all tenants for that period becomes no more than the total I/O bandwidth needed, or after each tenant t_i is assigned its minimum feasible memory size for that period; i.e., m_i^{min}. Before finishing a time period, GRO makes one last check to see if it is better to rollback the last iteration. Then GRO moves to another standard time period, and so on. We denote the resource allocation returned by GRO as $\dot{\mathcal{A}}$.

To analyze the running time of GRO, let us use a priority queue to pick $i(l)$ at each iteration. The running time of GRO depends on the value of $1/\mu$; that is, the granularity at which memory is assigned to tenants. If $1/\mu$ is a constant then the absolute approximation error equals 1, and the running time is $O(n\ log(n))$, where n is the number of tenants. However, μ can be used as a parameter to control the running time as well as the approximation error of GRO. If μ is a parameter the running time of GRO is $O(n\frac{1}{\mu}(lg(n) + \frac{1}{\mu}))$, and the worst case absolute error equals $1 + n \cdot \mu$. The extra error stems from the fact that each tenant may be assigned $\mu - \epsilon$ memory units more than its optimum memory assignment, where ϵ is a negligible amount of memory. The following theorem states the approximation guarantee of GRO when $1/\mu$ is constant.

THEOREM 5.1. $Lwr\_kDVP(\dot{\mathcal{A}}) \leq Lwr\_kDVP(\hat{\mathcal{A}}) + 1$

The proof of Theorem 5.1 is in the appendix. From Equation 2 and Theorem 5.1, we get

$$\begin{aligned} Opt\_kDVP(\dot{\mathcal{A}}) &\leq lg(2Z) \cdot Lwr\_kDVP(\dot{\mathcal{A}}) \\ &\leq lg(2Z) \cdot Lwr\_kDVP(\hat{\mathcal{A}}) + lg(2Z) \\ &\leq lg(2Z) \cdot Lwr\_kDVP(\mathcal{A}^*) + lg(2Z) \\ &\leq lg(2Z) \cdot Opt\_kDVP(\mathcal{A}^*) + lg(2Z) \end{aligned} \tag{3}$$

5.3 Balanced Resource Optimization (BRO)

In this section we present an online heuristic alternative to GRO, namely Balanced Resource Optimizer (BRO). The high-level idea of BRO is as follows. Given a single tenant, consider each standard time period as a separate problem. For each standard time period, allocated to the tenant a memory size that makes the I/O bandwidth allocated to the tenant balanced with the memory size. Note the similarity with GRO, except that BRO processes one tenant at a time.

To see the rationale behind BRO, let us define $\mathfrak{D}(\mathcal{A})$ for any resource allocation $\mathcal{A}$ as $\mathfrak{D}(\mathcal{A}) = \max_{i,j}\{A_i[j]\}\}$. That is, $\mathfrak{D}(\mathcal{A})$ is the maximum dimension of all vectors in $\mathcal{A}$. From [5], if $\mathcal{A}$ is used as input to ILP-1 and LP-1, the worst case ratio between the solution value of ILP-1 and the solution value of LP-1 is $1+2Z \cdot \mathfrak{D}(\mathcal{A})$. That is,

$$Opt\_kDVP(\mathcal{A}) \leq (1 + 2Z \cdot \mathfrak{D}(\mathcal{A})) \cdot Lwr\_kDVP(\mathcal{A}) \tag{4}$$

To ensure that the number of servers needed by our system is not much larger than the optimum number of servers, we need to minimize the right hand-side of Inequality 4. GRO focuses on finding a resource allocation that minimizes $Lwr\_kDVP(\mathcal{A})$. Meanwhile, BRO minimizes $\mathfrak{D}(\mathcal{A})$ by ensuring that, for each tenant t_i and for each standard time period j, the memory size allocated to t_i minimizes $\max\{m_i^j, P_i^j(m_i^j)\}$.

Unlike GRO, BRO does not have an approximation guarantee. In fact, it is possible to come up with corner cases where BRO performs very badly. For example, consider the case when we have n tenants, all of them have the profile $\{P_i^j(1-\epsilon) = 0, P_i^j(0) = 1\}$. BRO sets the buffer size of each tenant to $1 - \epsilon$ so as to minimize $\mathfrak{D}(\mathcal{A})$. However, this resource allocation requires n servers to host the n tenants, which is the maximum number of servers any resource allocation algorithm would require. When the same input is given to GRO, GRO sets the buffer sizes of half the tenants to $1 - \epsilon$, and the buffer sizes of the other half to 0, thus requires $\lceil n/2 \rceil$ servers to host the given tenants. Nevertheless, as we show in our experiments, BRO performs almost the same as GRO on typical OLAP workloads. Moreover, BRO has several practical advantages over GRO. First, BRO assigns a memory size to each tenant independently of other tenants, thus it can be used as an online algorithm. Second, even in an offline mode, BRO is more efficient in terms of running time compared to GRO. If μ is not a parameter, the running time of BRO is linear in the number of tenants. Otherwise, if μ is a parameter, the running time of BRO is $O(n/\mu)$, where n is the number of tenants. Third, BRO can be generalized easily to handle the trade-off between more than two resources. For example, if we consider co-locating CPU-bound workloads such as OLTP with I/O-bound OLAP workloads, we can extend our resource allocation vectors to include, for each time period, the CPU time allocated to tenants besides memory size and I/O bandwidth. Using an analysis similar to that we do for the I/O-bound case in Equation 4, we can use BRO to find a resource allocation that is close to optimum for a mixture of I/O-bound and CPU-bound workloads by balancing the three resources (i.e., RAM, I/O, and CPU). Fourth, BRO does not assume that the profiles of the tenants are monotonic functions. Fifth, BRO is much simpler and easier to implement compared to GRO.

We also examine a hybrid approach that attempts to balance the memory size and I/O bandwidth of each tenant, while taking advantage of the approximation guarantee of GRO. We refer to this approach as GRO+BRO. In GRO+BRO, we begin by running GRO till its end to obtain $\dot{\mathcal{A}}$, then we iteratively reduce $\mathfrak{D}(\dot{\mathcal{A}})$ as long as this reduction does not degrade the approximation guarantee of $\dot{\mathcal{A}}$. Our experiments show that, while GRO+BRO improves the approximation guarantee of GRO significantly, it rarely decreases the actual number of servers needed by GRO, since the actual number of servers needed by GRO is usually much less than the upper bound provided by its approximation guarantee.

6. RESOURCE CONTROLLER

Once a tenant is deployed on a shared server with other tenants, a resource controller is responsible for allocating the appropriate amount of each resource to the tenant. The controller is also responsible for changing the amount of resources allocated to the tenant at different time periods based on the resource allocation vector that is generated by the resource optimizer. We use MySQL as the default database management system for our cloud environment. MySQL does not allow for the buffer pool size of a database to change at runtime, so we had to modify InnoDB, the storage engine of MySQL, to be able to change the amount of memory allocated to a tenant at runtime without restarting the database process. The resource controller uses standard Linux commands such as `ionice` and `nice` to partition the I/O bandwidth and CPU time among tenants. Other tools like `ioband` and `cpuset` can be used for that purpose as well. In this section we explain how the resource controller controls each of these three resources.

6.1 Memory Size

We modify the code of InnoDB, the storage engine of MySQL, in order to be able to change the amount of memory allocated to a tenant at runtime without having to restart the database process. MySQL allows the user to define more than one buffer pool *instance* for a single database process. MySQL evenly divides the buffer pool size that the user specifies at launch time, for a single database process, among all buffer pool instances of that process. Whenever MySQL needs to load a data page from disk, MySQL picks one of the buffer pool instances arbitrarily, using a hash funcion, and assigns the new data page to that buffer pool. Similarly, whenever MySQL searches for a data page in the buffer pool, MySQL uses the same hash function to determine which buffer pool instance to search in. The purpose of having multiple buffer pool instance is to reduce contention on the mutexes that guard the data structures that MySQL uses to manage each buffer pool instance.

We make use of the fact that each buffer pool instance has its own data structures so as to make our memory adjustments safe and seamless. To implement our memory adjustments into MySQL, we begin by making the startup parameter

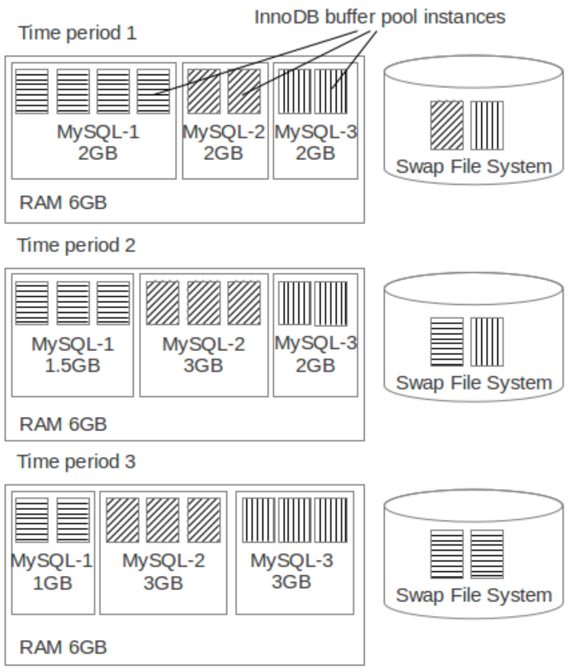

Figure 3: An example of runtime buffer pool adjustment in Elastic MySQL

`innodb_buffer_pool_instances`, that holds the number of buffer pool instances, a dynamic parameter that can be changed at runtime. Whenever the value of that parameter decreases, say from x to y, we change the hash function that maps data pages to buffer pool instances, so that all pages map to the first y buffer pool instances. The remaining $(x - y)$ buffer pool instances remain unused, so all their memory pages get swapped gradually to disk by the virtual memory manager of Linux, if there is another process that is requesting more memory pages. Conversely, when the value of `innodb_buffer_pool_instances` increases back from y to x, we re-adjust the hash function to map data pages to buffer pool instances from 1 to x; this forces MySQL to request more memory pages from Linux for the buffer pool instances from $y + 1$ to x, and Linux allocates those memory pages by swapping out unused memory pages of other tenants on the same server. Note that the value of `innodb_buffer_pool_instances` can never be more than the initial value of that parameter that is set at database launch time. Thus, for each tenant, the resource optimizer sets the initial value of `innodb_buffer_pool_instances`, as well as the initial size of the buffer pool, in a way to satisfy the following condition. The memory size that needs to be allocated to the tenant at any time period of the day, based on the resource allocation vector generated by the resource optimizer, should be able to be allocated to the tenant by maintaining a subset of the initial set of buffer pool instances in main memory, and swapping the remaining buffer pool instances to disk.

Figure 3 illustrates an example of a deployment in which the resource controller changes the memory sizes of tenants at runtime. In this deployment, three tenants share the same server, and each of them has different memory requirements for three different time periods of the day. For each tenant, the resource controller sets the startup configurations of MySQL such that the buffer pool size allocated to the tenant database equals the maximum amount of memory that the tenant needs for any time period of the day (i.e., 2GB, 3GB, and 3GB for MySQL-1, MySQL-2, and MySQL-3, respectively); at the same time, the resource controller sets the number of buffer pool instances of the tenant database such that the amount of memory that the tenant needs for any time period is a multiple of the size of a single buffer pool instance (i.e., 4, 3, and 3 buffer pool instances for MySQL-1, MySQL-2, and MySQL-3, respectively). Note that the buffer pool instances of different tenants do not have to have the same size; that is, the size of a buffer pool instance is not a unit of memory allocation. In Figure 3, MySQL-1 has a total buffer pool of size 2 GB, divided over 4 buffer pool instances, while each of the two other tenant has a buffer pool of size 3 GB, divided over 3 buffer pool instances. For the first time period, the resource controller sets the value of `innodb_buffer_pool_instances` for MySQL-2 and MySQL-3 to 2, and sets it to 4 for MySQL-1. Although MySQL-2 (and MySQL-3) has a buffer pool of size 3 GB, MySQL does not map any data pages to the third buffer pool instance, because of the modifications that we made. Therefore, all the memory pages of the third buffer pool instance get swapped to disk by the Linux virtual memory manager, and are replaced by memory pages of MySQL-1. In the second time period, the resource controller reduces the memory size of MySQL-1 by reducing `innodb_buffer_pool_instances` to 3, and increases the memory size of MySQL-2 by increasing `innodb_buffer_pool_instances` to 3. As MySQL-2 starts to map data pages to the third buffer pool instance, the data pages of that buffer pool instance replace those of the fourth buffer pool instance in MySQL-1, while MySQL-1 refrains from mapping any data pages to its fourth buffer pool instance. Similarly, in the third time period, the third buffer pool instance of MySQL-3 replaces another buffer pool instance of MySQL-1.

6.2 I/O Bandwidth

On a Linux system, the command `ionice` sets the I/O priority of a given process to a value from 0 to 7, where 0 is the highest I/O priority. The command `ionice` works only when the I/O scheduler of the Linux kernel is set to Complete Fair Queuing (CFQ), which is the default I/O scheduler starting Linux 2.6.18. By default, CFQ time-slices I/O fairly among processes, and gives higher priority to synchronous I/O over asynchronous I/O. Let x_o be the default time slice for synchronous I/O, which is a configurable parameter of CFQ. When `ionice` is used to set the I/O priority of a tenant t_i to n_i, the time slice assigned by CFQ to t_i equals $x_o + (x_o/5) * (4 - n_i)$. Thus, given a set of tenants running on the same server, in order to assign to each tenant t_i a share of I/O bandwidth IO_i, we find a feasible solution to the following system of equations.

$$\begin{aligned} x_i = x_o + (x_o/5) * (4 - n_i) \quad & \forall i \\ x_i / \sum_j x_j \geq IO_i \quad & \forall i \\ 0 \leq n_i \leq 7 \quad & \forall i \end{aligned}$$

In any feasible solution to this system of equations, the value of the variable n_i indicates the I/O priority that should be assigned to the tenant t_i, using the `ionice` command, in order to partition the available I/O bandwidth among tenants, and allocate a percentage of I/O bandwidth that equals IO_i to the tenant t_i. We also set the default time slice of CFQ for synchronous I/O to a value of x_o that makes this system of equations feasible. We implement this system of equations in AMPL, and solve it using any MINLP solver; e.g., FilMINT [2].

6.3 CPU Time

For CPU time, the `nice` command on Linux sets the CPU priority of a given process to a value from -20 to 19, where -20 is highest CPU priority. The ratio between the CPU time assigned to two tenants t_1 and t_2 with CPU priorities n_1 and n_2, respectively, equals $(20-n_1)/(20-n_2)$. Thus, given a set of tenants that are co-located together on the same server, in order to assign to each tenant t_i a percentage of CPU time CPU_i, we find a feasible solution to the following system of equations.

$$\begin{aligned} (20 - n_i)/(20 - n_j) = x_i/x_j \quad & \forall i, j \\ x_i / \sum_j x_j \geq CPU_i \quad & \forall i \\ -20 \leq n_i \leq 19 \quad & \forall i \end{aligned}$$

In any feasible solution to this system of equations, the value of the variable n_i indicates the CPU priority that should be assigned to the tenant t_i, using the `nice` command, in order to partition the available CPU time among tenants, and allocate a percentage of CPU time that equals CPU_i to the tenant t_i. Similar to the case with I/O bandwidth, we implement this system of equations in AMPL, and solve it using a MINLP solver.

7. EXPERIMENTS

We begin by explaining our experimental setup, then present a set of experiments to evaluate the effectiveness of our resource optimization algorithms and the accuracy of our profiling approach. The outputs of GRO and BRO, as well as other baseline heuristics that we compare our resource optimizer against, are passed as inputs to First-Fit Decreasing (FFD) [8], which approximates multi-dimensional vector packing with an approximation guarantee of $7/3$. The goal of our experiments is to demonstrate (1) that our online heuristic BRO achieves nearly the same results as our offline approximation algorithm GRO that has an approximation guarantee, when run on typical OLAP workloads, (2) that the resource allocations generated by GRO and BRO, when passed as input to multi-dimensional vector packing, achieve significant savings in the number of servers compared to the resource allocations generated by other baseline heuristics, and (3) that our profiler captures the trade-off between memory size and I/O bandwidth with good accuracy.

7.1 Experimental Setup

We implement the profiler and the resource optimizer as Linux Bash scripts, and implement the rest of the system in C++. We begin by profiling a set of 15 tenants. We use the standard TPC-H schema (version 2.14.4), and populate each database with random data that we generate using the TPC-H `dbgen` tool. For each tenant we generate a stream of queries, randomly chosen from the set of 22 queries defined by the TPC-H benchmark. During profiling, the profiler measures the throughput of each tenant under different memory sizes, from 100MB through 1000MB, with a step size of 100MB or 200MB, depending on how quickly the throughput of the tenant drops as the memory size decreases; steeper curvatures require more fine-grained profiling. For each memory size, tenants are profiled for I/O bandwidths, raning from 20% through 100%, with 20% step size. We refer to this set of profiles as T_{real}. We then construct another set comprised of a thousand profiles that we refer to as T_{scaled}. Each profile t_i in T_{scaled} is constructed by randomly picking a profile t_k from T_{real}, and scaling t_k by a random factor, say x, where x is a positive integer no more than server memory size. We scale a profile t_k by setting $P_i^j(m) = P_k^j(m/x)$, for

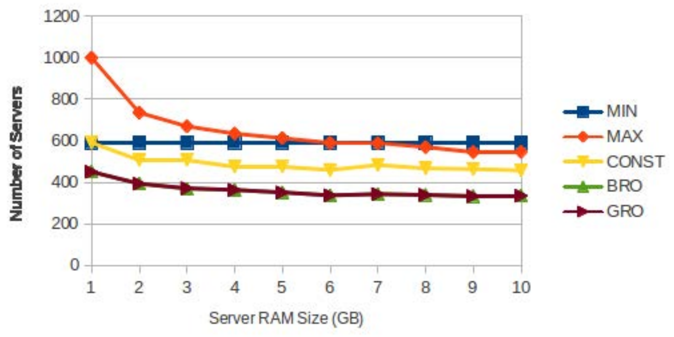

Figure 4: Effectiveness of GRO and BRO, for scaled profiles.

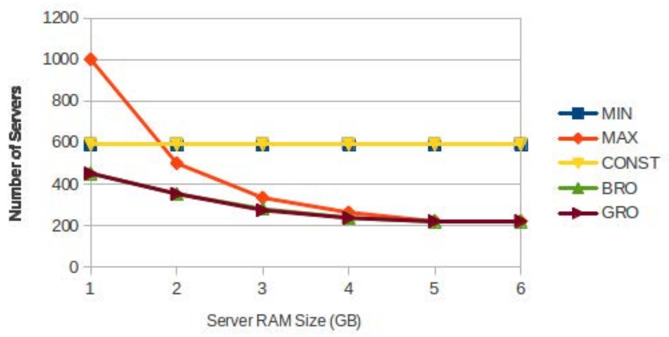

Figure 5: Effectiveness of GRO and BRO, for unscaled profiles.

all m, and for all j. This scaling operation maintains the original curvatures of the profiles in T_{real}, but with larger units of memory.

7.2 Server Savings

We evaluate the effectiveness of our algorithms and heuristics by comparing them against three baseline resource allocation heuristics: (1) MIN, which assigns to each tenant t_i its minimum feasible buffer size m_i^{min}, (2) MAX, which assigns to each tenant t_i its maximum buffer size m_i^{max}, that is either the database size of t_i if it fits in main memory, or the server main memory size otherwise, and (3) CONST, which assigns the same buffer size m^{const} to all tenants, that is the minimum buffer size that satisfies the throughput SLAs of *all* tenants. The I/O bandwidth assigned to each tenant t_i is defined by its profile P_i. Thus, the I/O bandwidths assigned to t_i by MIN, MAX, and CONST are $P_i(m_i^{min})$, $P_i(m_i^{max})$, and $P_i(m^{const})$, respectively. We use the set T_{scaled} as input to these three baseline heuristics, as well as our algorithms GRO and BRO, to obtain for each tenant five different resource allocation plans, then we use these configurations as input to multi-dimensional vector packing to obtain five tenant placement plans. We compare the number of servers needed by different placement plans. Figure 4 illustrates the number of servers needed by each plan as the size of the main memory of servers increases from 1GB to 10GB. Figure 4 shows that GRO and BRO always require the same number of servers on typical OLAP workloads, and they both achieve significant savings in the number of servers compared to other baseline heuristics. We get the same results when repeating the experiment with server RAM sizes that range from 100MB to 1000MB; in such case we use fractional scaling factors when generating T_{scaled}, since the maximum feasible memory size of any tenant can not be more than the main memory size. Next, we replace the set T_{scaled} with another set T'_{real} that consists of 1000 tenants whose profiles are randomly picked from T_{real} but without scaling. Figure 5 shows the number of servers needed by each placement plan when we use T'_{real} as input to the system. Figure 5 shows that GRO and BRO still achieve significant savings in the number of servers compared to other baselines. As the size of the main memory of the servers increases, the MAX heuristic catches up with GRO/BRO. This is due to the fact that all tenants have database sizes of no more than 1GB, and we have 5 possible I/O bandwidths (20%-100%, with 20% step size), thus the number of tenants placed on any server can not be more than 5.

7.3 Profiling Accuracy

We run a set of experiments to evaluate the accuracy of the predictions made by the profiler about the runtime performance of different tenants. We compare the observed throughputs of tenants, after they are deployed on shared servers, against their expected throughput that is given by their profiles. We do this evaluation for two types of environments. The first environment is a set of stand-alone servers in NEC Labs; each server runs Linux 2.6, and has a single disk and 2 Intel Quad-Core Xeon processors with hyper-threading. The second environment is Amazon EC2 virtual machines[1]; we use medium-size instances (m1.medium) that run Linux 2.6, and each has 1 virtual core with 2 EC2 Compute Units (i.e., about 2-2.4 GHz Opteron or Xeon processor). Note that in the case of EC2 machines, we still use a shared OS model of multi-tenancy, because all tenants that are co-located together by our system are deployed on the same EC2 machines. We deploy the tenants in the set T_{real} on shared data servers, and divide server resources among them based on their profiles, then compare their observed throughputs against the throughputs that are expected based on their profiles. The servers that we use to host tenants have the same configurations as the servers that we use during the profiling phase; so for each of the two environments that we use for our experiments, we do the profiling and the evaluation on the same environment. Profiling is done on private servers, while evaluation is done on shared server deployments. Figure 6 illustrates the observed throughputs of tenants compared against their expected throughputs as indicated by their profiles. Each tenant is represented as a point whose x-coordinate is the expected throughput of the tenant, while the y-coordinate is the observed throughput on the shared server. Our experiments show that, when using stand-alone servers, our profiler captures the trade-off between the different resources allocated to tenants, with minimal violations of SLAs at runtime. Profiling on EC2 virtual machines results in more violations, even when we add

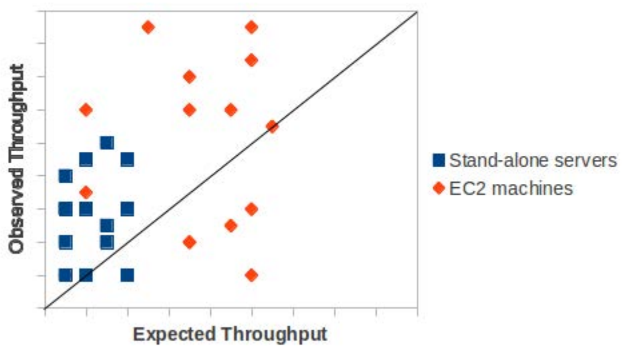

Figure 6: Observed throughput versus expected throughput.

[1]http://aws.amazon.com/ec2/

a constant error margin to the profiles; this may be due to the fact that EC2 machines are virtual machines that share their resources with other virtual machines in the Amazon EC2 environment. In a production environment, anomalous tenants whose runtime performance does not meet their expected performance are typically moved to other servers using load-balancing and live migration techniques. Our profiling approach, when used with stand-alone servers, helps system administrators keep the number of tenants that require migration low.

8. CONCLUSION

We study the trade-off between memory size and I/O bandwidth for I/O-bound OLAP workloads with SLAs. We propose a profiling approach that captures this trade-off. Then we develop an approximation algorithm, namely GRO, that approximates a globally-optimum resource allocation to minimize the total number of servers needed to host a set of tenants, while satisfying the SLAs of each tenant. We also propose an online heuristic, namely BRO, that performs almost the same as GRO on typical OLAP workloads despite the lack of an approximation guarantee. We develop a resource controller to dynamically change the amount of resources allocated to each tenant at runtime, to account for fluctuations in workloads, relying on a version of MySQL that we modified for that purpose. Finally, we conduct an experimental study to demonstrate the effectiveness of GRO and BRO compared to other baseline heuristics, and to evaluate the accuracy of our profiling approach on different environments.

Acknowledgements

The work in this paper was partially funded by a gift grant from NEC Labs America and NSF grant CNS 1053594.

9. REFERENCES

[1] http://www.salesforce.com.

[2] K. Abhishek, S. Leyffer, and J. Linderoth. Filmint: An outer approximation-based solver for convex mixed-integer nonlinear programs. *INFORMS Journal on computing*, 22(4):555–567, 2010.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[4] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, 2008.

[5] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 185–194, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[6] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[7] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Who's driving this cloud? towards efficient migration for elastic and autonomic multitenant databases. Technical Report CS-2010-05, UCSB, 2010.

[8] M. R. Garey, R. L. Graham, D. S. Johnson, and Andrew. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory*, 21:257–298, 1976.

[9] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the 3rd international workshop on Data management on new hardware*, DaMoN '07, pages 6:1–6:9, 2007.

[10] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.*, 26(4):63–68, Dec. 1997.

[11] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *SIGMOD*, 1987.

[12] H. Hacıgümüş, J. Tatemura, W.-P. Hsiung, H. J. Moon, O. Po, A. Sawires, Y. Chi, and H. Jafarpour. CloudDB: One size fits all revived. In *SERVICES*, 2010.

[13] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *ICDE*, 2009.

[14] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.

[16] B. Reinwald. Database support for multi-tenant applications. In *IEEE Workshop on Information and Software as Services*, 2010.

[17] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, 2011.

[18] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, 2011.

[19] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.

[20] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

[21] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, 2011.

APPENDIX

In this appendix, we prove Theorem 5.1. To keep the notation simple, we use $\delta_i(l)$ to indicate both the amount of memory, and the range $[m_i(l)-\delta_i(l), m_i(l)]$. For any m in the range $\delta_i(l)$, such that m is a multiple of μ, we refer to the range $[m_i(l) - \delta_i(l), m]$ as a tail of $\delta_i(l)$, and the range $[m, m_i(l)]$ as the remainder of the tail. For any tail β, we use the symbol β to refer to the range of the tail as well as the size of the tail. Similarly, for any remainder α of the tail β, we use the symbol α for both the range and its size. For any m in the range $[m_i^{min}, m_i(l) - \delta_i(l))$, such that m is a multiple of μ, we refer to the range $[m, m_i(l) - \delta_i(l)]$ as a prefix of $\delta_i(l)$. For any prefix γ, we use the symbol γ to refer to both the prefix range and its size. We also use the function $P()$ to indicate the I/O bandwidth for a given buffer size as determined by the buffer size, or to indicate the difference of I/O bandwidth across a range, and we use $\bar{P}()$ to indicate the average difference in I/O bandwidth across a range; for example, $P(\delta_i(l)) = P_i(m_i(l) - \delta_i(l)) - P_i(m_i(l))$ and $\bar{P}(\delta_i(l)) = P(\delta_i(l))/\delta_i(l)$.

The proof of the approximation guarantee of GRO is based on the following two lemmas.

Lemma A.1: If β is a tail of $\delta_i(l)$ then $\bar{P}(\beta) \leq \bar{P}(\delta_i(l))$.

Proof: Assume for contradiction that $\bar{P}(\beta) > \bar{P}(\delta_i(l))$. We consider the non-trivial case when $0 < \beta < \delta_i(l)$, since otherwise the lemma is obvious. Let α be the remainder of β. Since $P(\delta_i(l)) = P(\alpha) + P(\beta)$, therefore $\bar{P}(\delta_i(l)) = \frac{P(\alpha)}{\delta_i(l)} + \frac{P(\beta)}{\delta_i(l)} = \frac{\alpha\bar{P}(\alpha)}{\delta_i(l)} + \frac{\beta\bar{P}(\beta)}{\delta_i(l)}$. Since $\delta_i(l) = \alpha + \beta$, thus $\bar{P}(\delta_i(l))$ is a convex combination of $\bar{P}(\alpha)$ and $\bar{P}(\beta)$. Since we assume that $\bar{P}(\beta) > \bar{P}(\delta_i(l))$, and $0 < \beta < \delta_i(l)$, therefore $\bar{P}(\alpha) < \bar{P}(\delta_i(l))$. Note that one of the endpoints of the range α is at $m_i(l)$, thus α is among the possible δ's for tenant t_i at iteration l. But by definition, $\delta_i(l)$ is the δ with the smallest value of $\bar{P}()$ for the tenant t_i at iteration l. Therefore we reach a contradiction.

Lemma A.2: If γ is a prefix of $\delta_i(l)$ then $\bar{P}(\gamma) > \bar{P}(\delta_i(l))$.

Proof: Assume for contradiction that $\bar{P}(\gamma) < \bar{P}(\delta_i(l))$. Let δ' be the concatenation of the ranges γ and $\delta_i(l)$. Then $\bar{P}(\delta')$ is a convex combination of $\bar{P}(\gamma)$ and $\bar{P}(\delta_i(l))$. Since we assume that $\bar{P}(\gamma) < \bar{P}(\delta_i(l))$, and by definition $\gamma > 0$ and $\delta_i(l) > 0$, therefore $\bar{P}(\gamma) < \bar{P}(\delta') < \bar{P}(\delta_i(l))$. Note that one of the endpoints of the range δ' is at $m_i(l)$, thus δ' is among the possible δ's for tenant t_i at iteration l. But by definition, $\delta_i(l)$ is the δ with the smallest value of $\bar{P}()$ for the tenant t_i at iteration l. Therefore we reach a contradiction.

With Lemma A.1 and A.2 proved, now we are ready to prove Theorem 5.1.

Proof of Theorem 5.1: Let $\dot{m}_i$ denote the buffer size allocated by $\dot{\mathcal{A}}$ to tenant t_i. We begin by proving that there is no other resource allocation $\tilde{\mathcal{A}}$ that satisfies both of the following two inequalities: (1) $\sum_i \tilde{m}_i < \sum_i \dot{m}_i$ and (2) $\sum_i P_i(\tilde{m}_i) < \sum_i P_i(\dot{m}_i)$, where $\tilde{m}_i$ indicates the buffer size allocated by $\tilde{\mathcal{A}}$ for tenant t_i. Then we show that if there exists $\tilde{\mathcal{A}}$ that satisfies only one of the two inequalities, then the difference between $Lwr\_kDVP(\tilde{\mathcal{A}})$ and $Lwr\_kDVP(\dot{\mathcal{A}})$ is no more than 1. Recall that the resources that we consider are CPU, RAM, and I/O. We set the percentage of CPU time equal to the percentage of I/O bandwidth; thus the resource allocation of any time period is a two-dimensional vector representing memory size and I/O bandwidth.

Assume for contradiction that there exists a resource allocation $\tilde{\mathcal{A}}$ that satisfies both Inequality 1 and Inequality 2. Let t_a be any tenant such that $\tilde{m}_a < \dot{m}_a$. Also, let t_b be any tenant such that $P_b(\tilde{m}_b) < P_b(\dot{m}_b)$. Since P_b is monotonically non-increasing, therefore $\tilde{m}_b > \dot{m}_b$. Let Δ_a denote the range $[\tilde{m}_a, \dot{m}_a]$, and let l_a be the iteration at which GRO sets the buffer size of tenant t_a to $\dot{m}_a$. Similarly, let Δ_b denote the range $[\dot{m}_b, \tilde{m}_b]$. GRO removes the range Δ_b from the buffer of t_b in one or more δ's. Let these δ's be ordered chronologically as $\delta_b^1, \delta_b^2, ..., \delta_b^3$, where δ_b^1 is the first removed δ. Let l_b be the iteration at which δ_b^1 is picked by GRO; that is, $\delta(l_b) = \delta_b^1$. Note that δ_b^1 may be not completely contained within the range Δ_b. We refer to the intersection between the ranges Δ_b and δ_b^1 simply as $\Delta_b \cap \delta_b^1$. The range $\Delta_b \cap \delta_b^1$ is actually a tail of δ_b^1. Therefore, from Lemma A.1, $\bar{P}(\Delta_b \cap \delta_b^1) \leq \bar{P}(\delta_b^1)$. We prove that for any such t_a and t_b, $\bar{P}(\Delta_b) \leq \bar{P}(\Delta_a)$, the we use this inequality to reach a contradiction with Inequality 2.

Consider the two possible cases, either (1) $l_a < l_b$, or (2) $l_b < l_a$. For Case 1, at iteration l_b, the buffer size of tenant t_a is still at $\dot{m}_a$, unchanged since l_a. Therefore Δ_a is one of the δ's that are considered for comparison at iteration l_b. Since GRO picks δ_b^1 at iteration l_b, therefore $\bar{P}(\Delta_a) \geq \bar{P}(\delta_b^1) \geq \bar{P}(\Delta_b \cap \delta_b^1)$. Similarly, since $\delta_b^2, ..., \delta_b^c$ are all picked by GRO at iterations subsequent to iteration l_b, therefore $\bar{P}(\Delta_a) \geq \bar{P}(\delta_b^j)$, for all j. Note that $\bar{P}(\Delta_b)$ is a convex combination of $\bar{P}(\Delta_b \cap \delta_b^1), \bar{P}(\delta_b^2), ..., \bar{P}(\delta_b^c)$. Therefore $\bar{P}(\Delta_b) \leq \bar{P}(\Delta_a)$. For Case 2, at iteration l_b, the buffer size of tenant t_a is at $m_a(l_b) > \dot{m}_a$. Let Δ_a' denote the range $[\dot{m}_a, m_a(l_b)]$. GRO removes Δ_a' from the buffer of t_a in one or more δ's. Let these δ's be ordered chronologically as $\delta_a^1, \delta_a^2, ..., \delta_a^d$, where δ_a^1 is the first removed δ. Note that these δ's are prefixes of one another, therefore $\bar{P}(\delta_a^1) < \bar{P}(\delta_a^2) < ... < \bar{P}(\delta_a^d)$. Note also that Δ_a is a prefix of δ_a^d, therefore $\bar{P}(\delta_a^d) < \bar{P}(\Delta_a)$. Since δ_b^1 is picked by GRO at iteration l_b, rather than δ_a^1, therefore $\bar{P}(\Delta_b \cap \delta_b^1) \leq \bar{P}(\delta_b^1) \leq \bar{P}(\delta_a^1) < \bar{P}(\Delta_a)$. Similarly, for all δ_b^j, if δ_b^j is picked by GRO at an iteration before l_a, then $\bar{P}(\delta_b^j) < \bar{P}(\Delta_a)$. Otherwise, if δ_b^j is picked by GRO after l_a, then we apply the logic of Case 1 to show that $\bar{P}(\delta_b^j) \leq \bar{P}(\Delta_a)$. Therefore, for all j, $\bar{P}(\delta_b^j) \leq \bar{P}(\Delta_a)$. As with Case 1, since $\bar{P}(\Delta_b)$ is a convex combination of $\bar{P}(\Delta_b \cap \delta_b^1), \bar{P}(\delta_b^2), ..., \bar{P}(\delta_b^c)$, therefore $\bar{P}(\Delta_b) \leq \bar{P}(\Delta_a)$. From Inequality 1, since $\sum_i \tilde{m}_i < \sum_i \dot{m}_i$, therefore $\sum_a \Delta_a > \sum_b \Delta_b$. And since $\bar{P}(\Delta_a) \geq \bar{P}(\Delta_b)$ for all a and b, therefore $\sum_a \Delta_a \bar{P}(\Delta_a) > \sum_b \Delta_b \bar{P}(\Delta_b)$, which can be re-written as $\sum_a P(\Delta_a) > \sum_b P(\Delta_b)$. Therefore, $\sum_i P_i(\tilde{m}_i) > \sum_i P_i(\dot{m}_i)$, which contradicts with Inequality 2. Thus, there is no resource allocation $\tilde{\mathcal{A}}$ that satisfies both inequalities, 1 and 2.

Consider the case when $\tilde{\mathcal{A}}$ satisfies only one of the two inequalities, either 1 or 2. We prove that in such case $Lwr\_kDVP(\dot{\mathcal{A}}) \leq Lwr\_kDVP(\tilde{\mathcal{A}}) + 1$. To prove this, we consider two special cases, either (1) $\sum_i \dot{m}_i \leq \sum_i P_i(\tilde{m}_i), \sum_i \tilde{m}_i < \sum_i P_i(\dot{m}_i)$, or (2) $\sum_i P_i(\dot{m}_i) \leq \sum_i P_i(\tilde{m}_i), \sum_i \tilde{m}_i < \sum_i \dot{m}_i$. Other than these two cases, $Lwr\_kDVP(\tilde{m}) > Lwr\_kDVP(\dot{m})$, and the theorem follows directly. For Case 1, note that the loop of GRO terminates once total RAM is no more than total I/O. Let L denote the last iteration, and let $d = \sum_i m_i(L) - \sum_i P_i(m_i(L))$, and $d' = \sum_i m_i(L-1) - \sum_i P_i(m_i(L-1))$. Since each iteration of the loop of GRO decreases the buffer size of a single tenant, and since the RAM and I/O of each tenant is at most 1, therefore each iteration decreases total RAM by no more than 1 and increases total I/O by no more than 1. Thus the difference between total RAM and total I/O decreases by no more than 2 after each iteration. In our case, $d \leq 0$, therefore $d' \leq 2$. If $d' \geq 1$, then $d \geq -1$, thus the gap between total RAM and total I/O when GRO terminates is no more than 1, and the theorem follows. If $d' < 1$, then d may be less than -1, however in such case the last iteration degrades the solution instead of improving it. We check for this case after the loop terminates, and if it occurs we rollback the last iteration, thus the final gap size is d'. Therefore, the gap between total RAM and total I/O is never more than 1. Since $Lwr\_kDVP(\tilde{m})$ lies within this gap, therefore the difference between $Lwr\_kDVP(\tilde{m})$ and $Lwr\_kDVP(\dot{m})$ is no more than 1. For Case 2, $\sum_i \dot{m}_i > \sum_i \tilde{m}_i$. Then there must be a tenant t_a such that $\tilde{m}_a < \dot{m}_i$. But from the definition of GRO, the two termination conditions are either (i) $m_i(l) = m_i^{min}$ for all t_i, which is not true in this case since $m_a \neq m_a^{min}$, or (ii) $\sum_i m_i < \sum_i P_i(m_i)$, which is also not true by the definition of Case 2. Therefore we reach a contradiction. ■