Flexible Resource Allocation for Relational Database-as-a-Service

Pankaj Arora[1], Surajit Chaudhuri[1], Sudipto Das[3*], Junfeng Dong[1], Cyril George[1], Ajay Kalhan[1], Arnd Christian König[1], Willis Lang[1], Changsong Li[1], Feng Li[4*], Jiaqi Liu[1], Lukas M. Maas[1], Akshay Mata[1], Ishai Menache[1], Justin Moeller[1], Vivek Narasayya[1], Matthaios Olma[1], Morgan Oslake[1], Elnaz Rezai[2*], Yi Shan[1], Manoj Syamala[1], Shize Xu[5*],Vasileios Zois[1]
(Authors are ordered alphabetically)
[1] Microsoft Corporation [2] Amazon [3] Amazon Web Services [4] Meta Platforms Inc. [5] Stripe Inc.
fra-project@microsoft.com

ABSTRACT

Oversubscription is an essential cost management strategy for cloud database providers, and its importance is magnified by the emerging paradigm of serverless databases. In contrast to general purpose techniques used for oversubscription in hypervisors, operating systems and cluster managers, we develop techniques that leverage our understanding of how DBMSs use resources and how resource allocations impact database performance. Our techniques are designed to flexibly redistribute resources across database tenants at the node and cluster levels with low overhead. We have implemented our techniques in a commercial cloud database service: Azure SQL Database. Experiments using microbenchmarks, industry-standard benchmarks and real-world resource usage traces show that using our approach, it is possible to tightly control the impact on database performance even with a relatively high degree of oversubscription.

PVLDB Reference Format:
Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu and Vasileios Zois. Flexible Resource Allocation for Relational Database-as-a-Service. PVLDB, 16(13): 4202 - 4215, 2023.
doi:10.14778/3625054.3625058

1 INTRODUCTION

The last decade has seen widespread enterprise adoption of relational Database-as-a-Service (DBaaS) [41]. A few examples of relational DBaaS include Amazon Aurora [2], Microsoft Azure SQL Database [14] and Google Cloud SQL [30]. These cloud services predominantly cater to Online Transaction Processing (OLTP) and Hybrid Transaction/Analytical Processing (HTAP) workloads. Cloud database services are typically *multi-tenant*, i.e., multiple databases from different customers share physical data center resources.

The dominant consumption model for relational DBaaS today is *provisioned*, where the customer pays for a fixed set of resources such as CPU, memory, and I/O, regardless of the actual resource consumption of their workload. Meanwhile, in the past few years, there has been an increasing interest in, and adoption of, *serverless* DBaaS, e.g., Amazon Aurora Serverless [1] and Azure SQL DB Serverless [15]. Like provisioned databases, serverless databases are allowed to utilize resources up to a pre-specified maximum. However, in contrast to provisioned DBaaS, serverless DBaaS offer a *pay-per-use* consumption model, i.e., customers only pay for resources actually used by their workload. Serverless databases are a good fit for scenarios where it is difficult or impossible to determine the right amount of resources to provision in advance, such as workloads with intermittent and unpredictable usage patterns, bursty workloads with low average utilization but large spikes, or databases for which expert DBAs are not available.

DBaaS poses a significant challenge to cloud service providers in terms of cost-of-goods-sold (COGS). The straightforward approach of statically reserving capacity for the maximum resources allowed for each database tenant guarantees that resource requests can always be met, but is not cost-effective, since it incurs the cost of reserving significant unused capacity in the data center even when average resources utilization is low [41]. Thus, *oversubscription* of resources – where the sum of maximum resources promised to each tenant exceeds the physically available capacity – is imperative for the service provider to allow packing more databases per node, thereby reducing COGS. Although increasing the degree of oversubscription reduces costs for the service provider, it also increases the likelihood that the service is unable to satisfy the resource demands of tenant workloads due to resource shortage, thereby impacting performance and ability to adhere to service-level-objectives (SLOs). Oversubscription is feasible because the average resource utilization in cloud DBaaS tends to be low, and it is statistically unlikely that a large number of tenants on the same node or cluster simultaneously experience high resource demands. Therefore, a cloud DBaaS infrastructure that possesses the ability to flexibly redistribute resources from tenants with low resource demands to tenants with higher resource demands, and can do so quickly and with low overheads, has the potential to significantly reduce the likelihood of resource shortages and minimize impact on performance in this setting.

The need for flexible resource management leads to a few essential requirements. First, within a node, low-overhead mechanisms for quickly redistributing resources, such as CPU and memory, *across tenants* are needed, so that active tenants are able to acquire the resources required for their workload without a significant delay. Second, an effective oversubscribed DBaaS must decide how to co-locate databases on nodes so as to reduce the likelihood of resource shortages. These decisions need to be made when new

*Work done while at Microsoft Corporation.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625058

databases arrive into the cluster and when balancing load by moving existing databases across nodes in the cluster. Finally, moving a database to a different node can significantly impact performance due to the loss of cached state. Therefore mitigating the impact of such moves is important.

Prior work has developed techniques that allow the oversubscription of resources on a single node for arbitrary applications running in virtual machines (VMs) or as operating system processes (e.g., [17, 45, 55]). However, such generic techniques for multi-tenant memory and CPU management are insufficient when applied to DBMSs, since they are unable to take into account how these key resources impact database performance [51]. Similarly, cluster managers (e.g., Kubernetes [29] or Service Fabric [32]) are able to place newly arriving tenants on nodes and balance load by moving tenants between nodes. However, in contrast to VMs in the cloud, which tend to be short-lived (e.g., with a 90^{th} percentile lifetime of 1 day [23] in Azure), cloud DBaaS have much longer lifetimes (e.g., a 90^{th} percentile of 88 days - see Section 2.2). As a result, generic placement and load-balancing techniques available in cluster managers today can result in too many unnecessary moves for DBaaS (see Section 8.3, [35]). Lastly, techniques for live migration of VMs or databases [11, 27] are too heavyweight for our scenario, since they do not model the performance impact of migrated state, and move excessive amounts of memory and storage state.

Contributions. In this paper, we present a novel architecture using which a relational DBaaS can effectively oversubscribe resources while retaining control on database performance. By exploiting our understanding of how DBMSs use resources over their lifetime, we design database-specific techniques at the node and cluster levels, and orchestrate their interplay. This achieves significantly better control on DBMS performance, and thus the ability to meet the desired SLOs, when compared to generic techniques.

Our key contributions are as follows. First, we develop a cooperative white-box technique for multi-tenant memory redistribution across databases within a node that takes into account how memory is used by the DBMS. We show that this technique significantly reduces the impact on database performance due to memory shortages (Section 4) when compared to state-of-the-art multi-tenant memory management mechanisms in the hypervisor and OS. Second, we present an algorithm for dynamically re-balancing databases across cores within the node to minimize the likelihood of CPU shortage for tenants sharing a core (Section 5). Third, we develop techniques for modeling how database resource usage changes over time and for quantifying the potential disruption of moving a database by taking into account database activity and the amount of database state that needs to be moved. We use these models to influence how the cluster manager places tenants in the cluster and which databases are moved in response to resource pressure (Section 6). Our techniques significantly reduce both the number of moves and the disruption on performance. Finally, we describe techniques for mitigating the impact of unavoidable database moves within in the cluster, by identifying and migrating any non-persistent DBMS state that significantly affects workload performance (Section 7).

The mechanisms described in this paper have been implemented and evaluated in the codebase of a commercial cloud database service: Microsoft's Azure SQL Database [14]. Our techniques apply to both provisioned and serverless databases. We perform extensive experiments (Section 8) using microbenchmarks, industry-standard benchmark workloads, and resource traces of production database workloads. We demonstrate that, for a given degree of oversubscription, our techniques result in significantly fewer resource shortages than state-of-the-art techniques in research and industry, and, hence, lead to smaller performance impact on tenant workloads. This paper does not focus on what degree of oversubscription is appropriate for use in a particular setting and which tenants should be oversubscribed, which are decisions primarily driven by the business needs of customers and the cloud service provider.

2 BACKGROUND AND MOTIVATION

We first describe the major infrastructure components of a relational cloud database service [41] using the example of the Azure SQL Database service. Azure SQL Database [14] is a highly available, multi-tenant, relational cloud database service based on Microsoft SQL Server, targeted at OLTP and HTAP workloads. It is a managed service that automates several essential administration tasks including provisioning, upgrades, backups, ensuring high availability, managing security and offers auto-tuning capabilities [25].

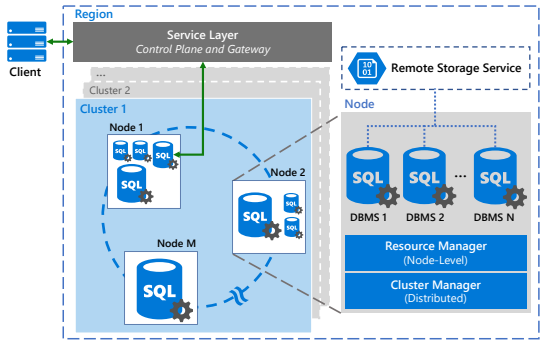
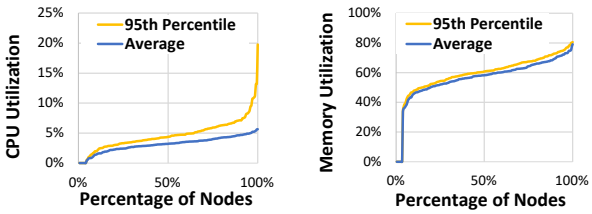


Figure 1: Architecture overview of Azure SQL Database [35]

2.1 Azure SQL Database

Database tenants are hosted on shared nodes in multiple *clusters*, as shown in Figure 1. Each tenant may have one or more replicas distributed across these nodes to achieve high availability. Multiple tenants can share the same physical node while being isolated into separate containers (e.g., using VMs or separate OS processes running their own private DBMS instance). A node-level resource manager controls how resources are shared across all tenants on that node. Multiple nodes are grouped together into logical clusters, each managed by a distributed cluster manager. SQL Database uses the *Azure Service Fabric* (SF) cluster manager [32], which is responsible for the placement of tenants onto nodes and for handling database moves, aka *failovers*, i.e., situations in which a tenant replica needs to be moved to a different physical machine, or where primary/secondary replicas swap roles. Such moves can be disruptive to database performance since the cached state of the primary is not preserved. All data and log files for a tenant are either stored in *Azure Storage* or on local SSD (and replicated within the cluster), depending on the tenant's *service tier* [13]. Each region is comprised



(a) CPU Utilization (b) Memory Utilization

Figure 2: CDF of resource utilization in a cluster

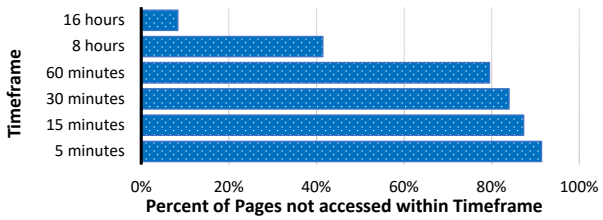


Figure 3: Access frequency of buffer pool pages in cluster

of a set of such clusters. A region-level *control plane* is responsible for directing newly created databases to one of the clusters, as well as client connection routing.

Provisioned vs. Serverless Billing Models. Azure SQL DB supports both *provisioned* and *serverless* compute models. In the provisioned model, tenants pay for a fixed amount of resources, regardless of how much of those resources is actually consumed by the workload. In contrast, the pay-per-use serverless model bills for compute usage per second. Tenants are categorized into *tenant classes*, which are defined by the combinations of a tenant's service tier, the maximum resources available to the tenant and the tenant's compute model (e.g., provisioned vs. serverless). Finally, while the average resource utilization can be low in both provisioned and serverless models, serverless tenants put an even greater pressure on the service provider to improve resource utilization and reduce cost, since serverless DB customers only pay for resources used.

2.2 Resource Usage Patterns

To motivate the need for resource oversubscription and to illustrate the resulting challenges, we analyzed the resource consumption on Azure SQL Database production clusters. The collected data captures a one-week trace from a representative 40 node cluster.

CPU Utilization. On each node, we collected CPU and memory utilization readings as the average over 15 second intervals. Subsequently, we computed aggregates over those readings, including average and 95th percentile for the entire week observed. Figure 2(a) shows the CDF of average and 95th percentile CPU usage across nodes in the cluster. On almost all nodes, the average CPU utilization is below 5% of capacity, with even the 95th percentile usage being below 20%. Even though the average CPU utilization on nodes is generally low in production, we observe cases where at least one CPU core on a node is hot, i.e., that has CPU utilization above 70% for at least one metering interval. In particular, in a representative oversubscribed production cluster, over a period of one week, each node saw on average 13 times per day where at least one CPU core was hot, with a median of 3, and 90/99/99.9th percentiles of 33/164/303 occurrences per node per day, respectively.

Memory Utilization. Figure 2(b) shows cluster-wide main memory utilization, which is significantly higher than CPU utilization. The differences between the average and 95th percentile utilization are much less pronounced, indicating that, in aggregate, memory utilization is much more stable. However, memory consumption does exhibit significant changes during a tenant's lifetime. Database systems tend to grow their memory consumption over time, as the DBMS accumulates state from caches, especially at the beginning of a tenant's lifetime, when memory often grows rapidly.

Although the memory utilization of the cluster is much higher than CPU utilization, a significant fraction of cache memory tends to be "cold". To quantify this, we took a random sample of the last access times of pages in the page buffer pool (which is the dominant consumer of cluster memory in Azure SQL DB) across all database tenants in the cluster. As shown in Figure 3, almost 80% of the pages in the buffer pools across the cluster had not been accessed in over 60 minutes, and over 40% of the pages had not been accessed in over 8 hours, indicating a large fraction of cold buffer pool memory.

Database Lifetimes. We measured the lifetimes of all databases created in Azure SQL DB within a two week window. We found these databases to have a median lifetime of 56 days and the 90th percentile of 88 days. In comparison, for VMs in Azure IaaS, which run a wide variety of applications, the 90th percentile lifetime is only about 1 day, and the median lifetime is below 12 hours [23]. This long-lived nature of databases brings additional challenges for how databases should be co-located onto nodes in a cluster.

3 ARCHITECTURE OVERVIEW

Resource competition among tenants in oversubscribed DBaaS clusters can lead to resource shortages. The key challenges of handling such shortages are: (a) Detecting impending or actual resource shortages; (b) Proactively reducing the likelihood of resource shortages; (c) Minimizing the impact resource shortages have on database performance. Our white-box design relies on exploiting our understanding of how DBMSs use resources over their lifetime and how resource allocations impact database performance. We develop *database-specific* resource allocation techniques, which allow us to achieve significantly better control over performance compared to generic oversubscription techniques designed for arbitrary applications [9], e.g., VM/OS level oversubscription [17, 45, 55], traditional approaches to load-balancing in clusters based on current utilization [29, 32], and techniques for VM and database live migration.

Figure 4 shows how our techniques fit into the software architecture of a cloud DBaaS. We develop new techniques within the DBMS, in the node-level resource manager which serves as the control plane for the node, as well as in the cluster manager. We describe a system that oversubscribes each resource by a resource-specific *oversubscription ratio*. The oversubscription ratio of a resource refers to the ratio of the total amount of the resource promised to tenants on a node or cluster (i.e., the sum of the maximum amount each tenant can consume) divided by the physical capacity of the resource on the node or cluster, respectively. A cloud service provider

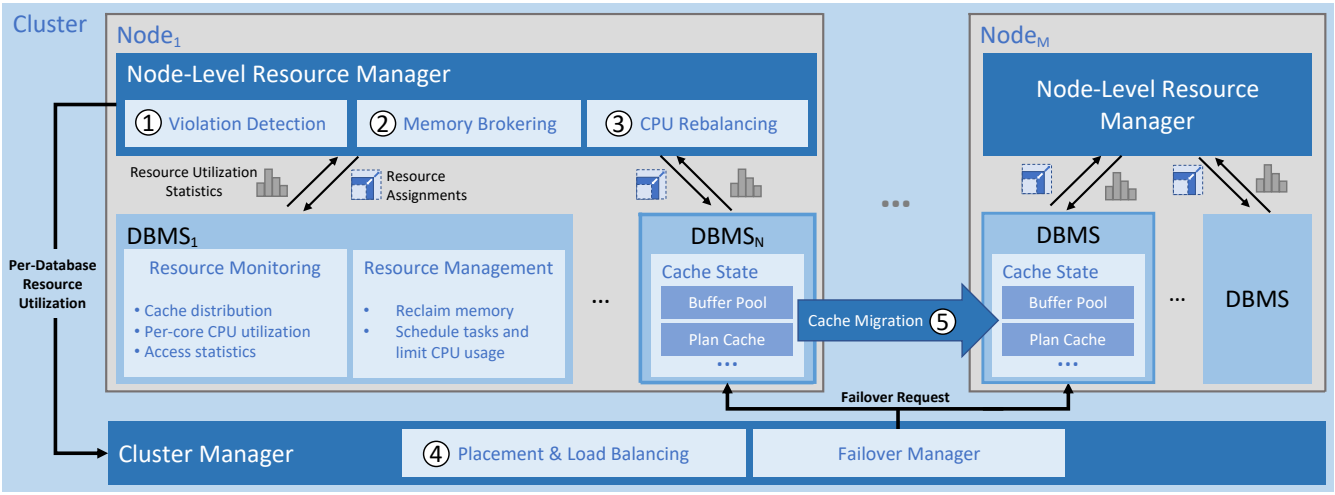


Figure 4: Overview of flexible resource allocation mechanisms in cloud DBaaS

can control the oversubscription ratio by limiting how many databases are admitted into the cluster. This paper focuses on describing effective techniques for managing resources in a DBaaS for any given oversubscription ratio, i.e., our goal is to minimize resource shortages, and thereby minimize the impact on performance of active databases.

To manage oversubscribed resources, whenever a database requests additional resources, the local resource manager first attempts to draw from *unused resources* on the node. If the available resources on the node fall below a threshold, node-local mechanisms attempt to free up the required resources by acquiring unutilized resources (such as "cold" cache memory or unused cores) from other database tenants on the node. Such resource acquisition is done in a low-overhead manner that minimizes disruption. If these node-level mechanisms do not alleviate the resource shortage, the cluster manager is invoked to free resources by moving (i.e., failing over) one or more databases to other nodes within the cluster. We observe that, although developed in the context of Azure SQL DB, this architecture is broadly applicable to any relational DBaaS. Our techniques only assume standard capabilities of operating systems (resource isolation and limiting, and process affinitization), and the extensions we propose to DBMSs and cluster managers are applicable to most modern DBMSs or cluster managers (e.g., Kubernetes and Service Fabric). In this paper, due to lack of space, we focus on CPU and memory resources, and are unable to cover techniques related to local disk space and I/O. Below, we provide an overview of the key techniques described in the remainder of this paper.

Detecting Resource Violations. In order to trigger the various techniques described in this paper, we need to be able to detect impending resource shortages. We use *resource violations* for this purpose, which we define as a node's usage of a resource exceeding a predefined threshold. To capture violations, the node-level resource manager monitors the resource usage at the level of both, the individual databases and the overall node, and periodically reports them to the cluster manager (①). Tracked resources include CPU utilization, memory consumption, local disk storage, and disk and network I/O utilization. For each resource, when the resource demand surpasses a predetermined threshold for a specified period of time, a resource violation is recorded. The exact thresholds are set based on historical data and reflect the speed at which the resources can grow, how disruptive a resource shortage for a particular resource would be, and what node-level mechanisms are in place to mitigate the violation without causing database failovers.

Mitigating Resource Shortages on a Node. Long-term resource shortage on a node can only be relieved by moving one or more databases to other nodes in the cluster. However, given the relatively low resource utilization in practice (see Section 2.2), the fact that resource demands of different databases on a node tend to be spread out over time, and the burstiness of many database workloads, temporary spikes in resource demand can often be directly handled by *node-level mechanisms*. To govern the memory distribution among tenants on the node, we develop *multi-tenant memory brokering* (②), a new technique which selectively redistributes memory across tenants with the goal of maintaining a buffer of free memory on the node while minimizing the total performance impact on databases. It is instrumental in addressing short-term spikes in memory demand, and in many cases can avoid the need to move databases to other nodes (Section 4). When multiple DBMSs share CPUs and are affinitized to overlapping CPU sets, there is potential for a resource violation on a core even when the the overall CPU utilization of the node is relatively small (see Section 2.2). We develop a dynamic node-wide core re-balancing algorithm (③) that prevents such resource violations by re-affinitizing DBMSs to different cores based on observed utilization on each core (Section 5).

Tenant Placement. We proactively optimize the placement of database tenants on nodes in the cluster to reduce the longer term likelihood of resource shortages on each node (④). Since databases are typically long-lived (see Section 2.2 and [47]), and their resource usage can vary significantly over time [35], our technique aims to place databases such that the likelihood of resource violations is minimized. We make tenant placement decisions when a new database arrives, or existing databases are moved within the cluster.

Placing tenants in a way that minimizes service disruption requires non-trivial extensions to the cluster manager. Due to space limitations, we focus on the novel aspects of our approach in Section 6; the remaining details appeared previously in [35] and [32].

Mitigating Failover Impact. After a database failover, the cached state of the original primary database is not available on the new cluster node (e.g., buffer pool and plan cache). To mitigate the performance impact of failovers due to oversubscription, we develop low-overhead techniques in the DBMS that can selectively migrate the subset of the cached state that has the most impact on performance (Section 7, ⑤).

4 MULTI-TENANT MEMORY BROKERING

Oversubscribing memory in a multi-tenant DBaaS means the total memory promised to databases on the node can exceed the amount of physical memory available on the node. Therefore, multiple databases actually demanding memory at the same time can potentially result in memory shortage. In practice the memory footprint of each database tends to grow over its lifetime, even when their in-memory caches are cold (e.g., see Azure SQL DB memory usage patterns in Section 2.2). The above behavior of DBMSs is problematic since we can run out of memory on the node and, therefore, active databases may be unable to satisfy their demand, even when most of the memory on the node is cold; requiring moves of tenants to other nodes in the cluster, which can be disruptive. It is therefore crucial for an oversubscribed cloud DBaaS to have the ability to redistribute memory across tenants while taking into account how memory usage affects the performance of a database, which we refer to as *multi-tenant memory brokering*. For the common case where there is a relatively short-lived memory demand from a single or a handful of active databases, such redistribution of memory is often sufficient to tide over the spike with minimal impact on any database's performance, thereby avoiding tenant moves.

Figure 5 shows the different levels of memory governance techniques that we employ depending on the amount of remaining node memory. To address short-term memory pressure, when the resource manager registers memory consumption above a defined limit, *multi-tenant memory brokering* is invoked to attempt to reduce the consumed memory on the node by freeing up cold cache objects across all tenants. If, despite that, memory usage grows and increases above a second *Failover Threshold*, tenant moves are initiated. If memory usage becomes too high before the tenant moves are completed, we use OS-level *memory throttling* as a a last resort to avoid the entire node becoming unresponsive. This can be achieved using memory limits in Linux cgroups [45] or Windows Job Objects [17], and max memory settings in VMs [55]. Since this technique is agnostic to the impact of memory on database performance, it can cause significant performance impact when invoked. Therefore, the combination of multi-tenant memory brokering and tenant moves are designed to make this situation very unlikely.

4.1 Problem Formulation

Any technique for redistributing memory across tenants must address a few key challenges. First, it must be able to model the impact of memory on each database's performance; taking into account the various use cases of memory within a DBMS, including various

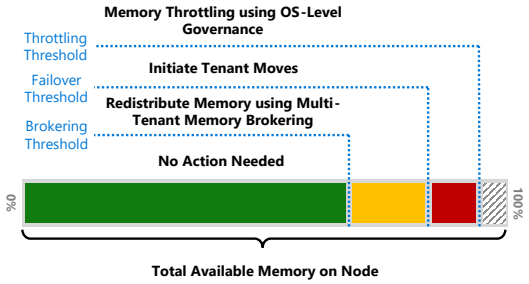


Figure 5: Node-level memory mechanisms and thresholds

types of caches. Second, we need a common currency *across tenants* for reasoning about how memory impacts performance, and we must be able to handle cases where tenants have different priorities (i.e., importance). Finally, to be a viable solution for production, the technique must be efficient, i.e., it must have low overheads imposed on each DBMS and should be able to respond to memory pressure quickly, usually in a matter of seconds.

Value of Memory. At the core of our design is the concept of *Value of Memory* (VoM). We use cached objects to define VoM, although this definition can be extended to other use cases of memory in databases, such as working memory. The VoM of a cache object tracks how much *system time* we expect to save by caching that object as compared to not caching it. We base the VoM on the notion of *system time saved* (STS) from [52], which refers to the amount of time required to create the cached object in memory; this normalizes costs across objects that mainly consume CPU to be created (e.g., query plans) and objects that require I/O to be issued (e.g., buffer pages). Concretely, the VoM of an object i is defined as

$$VoM_i = STS_i \times \lambda_i,$$

where λ_i corresponds to the expected number of accesses per unit of time. In essence, the VoM of an object represents the *expected time saved* by caching the object. Finally, we note that the VoM metric can be used as a common currency to reason about memory across different tenants since the *STS* is a measure of time.

Referring to Figure 5, multi-tenant memory brokering is triggered when the memory usage on the node exceeds the *memory brokering threshold*. If current memory usage is α units higher than the memory brokering threshold, our goal is to reduce the aggregate memory consumption of all tenants to a lower *reclamation target* by reclaiming a total of

$$M = \alpha + (MemoryBrokeringThreshold - ReclamationTarget)$$

units of memory from one or more databases on the node such that the aggregate performance impact over all tenants (captured by the VoM metric) is minimized. In turn, this task can be formulated as selecting a set of memory objects to retain such that their total memory requirements are below the reclamation target and their aggregate VoM is maximized. This corresponds to an instance of the *0-1 Knapsack Problem* [56], with the item *values* corresponding to the VoM of the cache objects and the item *weights* being the respective object sizes in memory. From the solution to this Knapsack instance, we can – by tracking which objects belong to which tenant – compute the amount of memory m_i to be released by $tenant_i$.

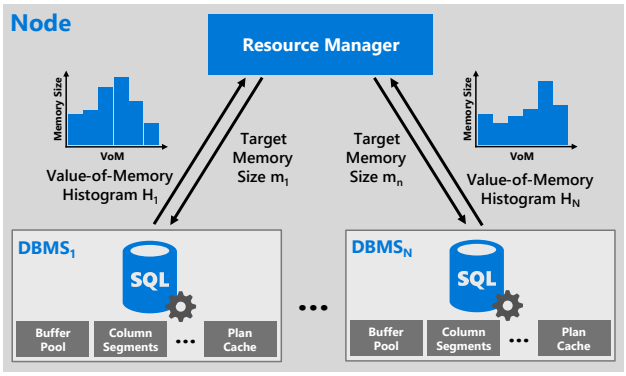


Figure 6: Multi-tenant memory brokering architecture

Tenant Prioritization: In addition, to handle cases where the cloud provider wants to provide differential priorities across tenants (e.g., third-party vs. first party tenants, expensive vs. inexpensive SLOs, etc.), each tenant $tenant_i$ can be assigned a *tenant weight* w_i. In that case the optimization objective is to optimize the *weighted* sum of VoM over all retained objects.

4.2 Implementation in Azure SQL DB

Figure 6 shows the high-level architecture of our multi-tenant memory brokering approach. When the resource manager registers increased memory utilization, it requests from every tenant information about its VoM distribution and uses this information to calculate a new target memory size for each tenant. The tenant is then notified of its new memory limit m_i and its SQL engine reduces its memory footprint by reclaiming cached objects to meet its new target size. To allow memory reclamation inside each DBMS to take effect and to detect consistent memory pressure that needs to be resolved through a failover, we add a 1 minute cooldown period between two successive invocations of the memory broker.

Since databases can contain very large numbers of cached memory objects of different sizes, we use a compact equi-depth histogram to reduce communication and computation overheads by summarizing each tenant's VoM distribution across its memory consumers, such as buffer pool, column store cache, plan cache etc. Each histogram bucket summarizes a range of VoM values, and tracks the total number and the aggregate size of all objects within that VoM range. We describe the details of the histogram computation below. Given the histograms, the node-level resource manager can solve the optimization problem at the level of histogram buckets instead of individual objects. Because the 0-1 knapsack problem is NP-hard, we use a greedy heuristic that proceeds to select the next bucket with the lowest VoM interval endpoint across all tenants until we have reclaimed at least M units overall. This heuristic is computationally efficient with a time complexity of $O(n \log n)$, where n is the number of tenants. In practice, histogram computation is non-intrusive and occurs only in the presence of memory pressure and at most once per tenant per minute. Space and communication overheads scale linearly with the number of tenants at a bounded maximum histogram size of $\approx$ 11 KB per tenant.

VoM Histogram Computation. To generate its histogram, each tenant iterates over the objects in its various caches as well as its *free list* [21]. We note that in Microsoft SQL Server working memory (e.g., for sorting or hashing) is not reclaimable *during query execution*, and hence actively used working memory is not included in VoM distribution computation. However, once query execution completes, the working memory is returned to the free page list and included in the histogram.

The exact approach to adding to the VoM histogram varies by the type of the respective cache, as follows:

Buffer Pool: The VoM distribution of the *buffer pool* is computed by sampling a small number of pages (i.e., a few hundred) uniformly at random, and computing their VoM using the existing per-page STS value (i.e., the latency of a disk read). Based on this, we scale the number of objects of a specific VoM by the inverse sampling fraction. Sampling is a crucial performance optimization since scanning all pages in the buffer pool would increase latency and adds non-trivial CPU overheads. λ_i is estimated from the last page references, which are retained by the (modified) LRU-K [44] page eviction scheme.

Other System Caches: SQL Server maintains separate caches for larger objects such as *column segments* [20] and cached query execution plans. While these caches usually contain significantly fewer elements than the buffer pool, their contents can differ significantly in size and STS. Because of this, instead of sampling, the VoM distribution of such caches is computed by a sweep over the entire cache. Like for the buffer pool, STS is already stored for each cached object using type-specific cost-models (e.g., time to compile a query plan) and λ_i is estimated from the last references stored.

Free Pages: The free page list contains pages that the database system process keeps pre-allocated to assign to various memory consumers when requested. Free page list sizes can be significant since, for example, working memory used by memory-consuming operators, such as Sort, is returned to the free list upon completion of the operator. Free pages are reported with a VoM reflecting the cost of an OS page allocation, which is typically much cheaper than the VoM of any cached objects. This ensures that the DBMS will at first try to release free pages before other cached objects.

5 UTILIZATION-BASED CPU REBALANCING

In a multi-tenant DBaaS with oversubscribed CPU resources, multiple database tenants may need to share the same physical CPU core. When two or more of those tenants simultaneously experience high CPU demand, a core can become overutilized and the tenants might not be granted their assigned share of CPU cycles. As noted in Section 2.2, such *core-level* CPU shortages can occur even when the node-wide CPU utilization is relatively low. It is, therefore, important that database tenants can be *dynamically re-balanced* across the available physical CPU cores. If the high core utilization cannot be resolved by re-balancing, CPU-consumption is throttled until CPU cycles can be freed up by moving tenants to different nodes.

Constraints. Most modern database systems use custom thread management to control the parallelism and priority of internal tasks, to reduce the overhead of context-switches, and to guarantee cache and data locality, including NUMA-awareness[1]. Furthermore, each tenant's SLO comes with an allocation of a certain number of

[1]In Microsoft SQL Server, this explicit thread management is implemented in the *SQL Operating System* [34], a platform abstraction layer providing custom facilities for thread-scheduling, memory management, asynchronous IO and synchronization.

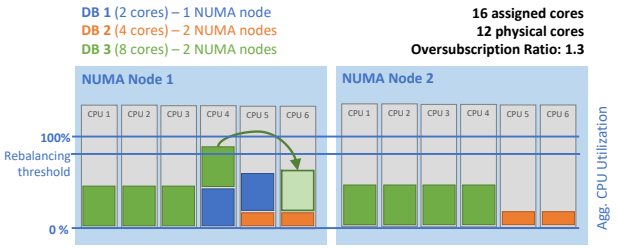


Figure 7: Example of a rebalancing operation to avoid high CPU utilization on an oversubscribed core

vCores. This number is significant for a DBMSs since it controls the maximum degree-of-parallelism for a query. Therefore, a technique like "slicing" a 1 vCore database across 2 physical cores and rate limiting its utilization to 50% on each core is not practical.

Affinization and CPU Rebalancing. The above constraints lead us to manage CPU resources proactively, rather than relying exclusively on the OS to load-balance threads using work-stealing capabilities [57]. We control core assignments by explicitly affinitizing each database tenant to a set of cores. On a given core, for the common case, where typically at most one database is active at any point in time, the active database achieves good data and instruction cache locality. Affinitization also allows us to satisfy soft-constraints such as NUMA-awareness. For tenants with a small number of vCores it is preferable to ensure that all vCores are assigned to the same NUMA node to avoid NUMA effects. However, for large tenants where using multiple NUMA nodes is unavoidable, it is preferable to have their vCores assigned roughly equally to each NUMA node to allow for better utilization of memory bandwidth.

To avoid CPU shortages on any individual core when databases are affinitized directly to physical cores, we use a *utilization-based* re-affinitization algorithm. For this the node-level resource manager monitors the *per-core* CPU utilization to ensure all over-subscribed cores have sufficient headroom to accommodate the assigned tenants. If at any time an over-subscribed physical core is considered highly-utilized (i.e., their utilization crosses a predefined threshold for a window of time), the node-level resource manager re-balances one or more active tenants by re-affinitizing the associated database processes to different subsets of cores (Figure 7) using a greedy rebalancing algorithm. We omit algorithmic details due to lack of space. In practice, we consider CPU rebalancing every 15 seconds, which provides a good trade-off between accommodating imbalances caused by changes in the average CPU utilization of tenants, and the overheads of re-affinitizing database processes.

CPU Throttling. In the unlikely case that the high core utilization cannot be resolved by re-balancing, including when the average CPU utilization *across all cores* increases above the high-utilization threshold, the node is marked as hot and the cluster manager is triggered to move tenants off the node. During this process, OS-level *CPU rate-limiting* [2] provides a last line of defense to ensure a fair distribution of CPU resources and to protect the host from overload while the system resolves the CPU violation.

[2]In Windows and Linux operating systems, this functionality is captured through *Job Object* [17] and *Control Group (cgroup)* [45] abstractions, respectively.

6 TENANT PLACEMENT

In an oversubscribed DBaaS, the assignment of tenants to nodes in the cluster becomes critical for achieving a balance between resource availability and effective resource utilization. Tenant placement decisions are challenging since database tenants are long-lived [47], accumulate large state [41] and exhibit significant changes in resource demand over time [35]. As a result, basing placement decisions on snapshots of current resource usage, as in prior work (e.g., [10, 31, 46, 48]) as well as industrial cluster managers, is likely to incur unnecessary resource violations. In contrast, our tenant placement approach leverages resource demands seen in previous tenants to compute future demand distributions (as well as their uncertainty) and incorporates these estimates into tenant placement. The resulting technique is implemented in a modified version of the *Placement and Load Balancer* (PLB) component of *Service Fabric*.

Background (PLB). The placement of tenant replicas within a cluster is an online optimization problem, with a number of interacting constraints such as *affinity* and *anti-affinity* between replicas, constraints on resource usage, etc. [3, 4, 6]. We refer to an assignment of tenants to nodes in the cluster as a *configuration*. When new tenants are placed, or existing tenants are moved, PLB considers a space of *candidate configurations*, each of which is associated with a *score*. Among all candidates satisfying the constraints, the candidate configuration with the minimal score is selected. Our tenant placement approach can be characterized by (a) the *search algorithm* used to enumerate candidate configurations and (b) the *scoring function* used to select among them. We describe these components next.

Enumerating the Space of Configurations. PLB generates candidate configurations by first identifying, for each replica to be placed/moved, all *target nodes* which can host this replica without generating a constraint violation, and placing the replica at a target node chosen at random. Subsequently, the configuration with the largest number of placed replicas is chosen as the initial *seed configuration*. After this, the overall cluster score is optimized using *Simulated Annealing* (SA) [33]. SA explores the configuration space by generating random moves (e.g., moving a replica) and computing the score for each resulting configuration. Depending on this score, the new configuration is adopted with a certain probability and subsequently used as the seed for further SA iterations. This process continues until a timeout expires. Details of this algorithm can be found in [32].

Scoring Candidate Configurations. The scoring function we use to place tenants (see [12]) minimizes the product of (a) the weighted sum of the standard deviations over all metrics[3] [19] across all nodes and (b) the *weighted* [7] sum of *failovers* of all replicas being moved to reach the target configuration and any future *expected failovers* due to resource violations[4]. The *expected failover* term is computed based on the distribution of resource demands of previous tenants. Next, we will describe the key components of this score in more detail, describing (a) the computation of the expected number of future failovers, (b) how weights are assigned to different tenants

[3]Metrics typically correspond to individual resource demand.
[4]The scoring function includes additional terms such as penalties associated with insufficient free nodes and the total replica count, which we omit for clarity.

to minimize service disruption and (c) a mechanism used to avoid resource violations for new tenants.

Estimating Expected Failovers. To compute the expected number of future failovers for a candidate configuration, we first estimate – for a set of tenants co-located on a node – the probability of a future resource violation. Accurate point estimates of future tenant resource demands are inherently difficult, especially for new tenants, for which only the tenant class (see Section 2.1), but no usage information is known. Consequently, we model the distribution of possible resource demand curves over time.

For this, we perform a *Monte-Carlo* (MC) simulation that uses resource demand traces from previous tenants and, in each MC iteration, replays the traces to obtain a possible outcome of resource demand on a node. The estimated probability of violation then corresponds to the fraction of MC iterations with at least one violation. We observe that the *expected number of resource violations* computed above is a lower bound on the expected number of *future* failovers, since each resource violation requires at least one failover to resolve. Therefore, we add this lower bound to the *failover* term of PLB's scoring function, effectively summing up the required and the *expected* failovers resulting from a new configuration. The details of the MC simulation are described in [35].

Assigning Weights to Tenants. To minimize the amount of service disruption due to tenant failovers, the scoring function also needs to account for the differences in resource usage between replicas: For example, replicas that use large local disk space require longer for a failover to complete, as the local state needs to be copied to the new node. Similarly, replicas using more cache memory require either more cache state to be transferred (see Section 7) or may experience a more pronounced (temporary) performance degradation as a result of a failover. Finally, the activity level of a tenant is a key factor – failing over tenants that are *inactive* masks much of the impact of the failover: No running queries are canceled, there are no lost connections, and the failover may complete before tenant becomes active again.

Therefore, we need to take these factors into account when selecting tenants to fail over in response to violations. The PLB mechanism we use is *move costs*, which assigns weights to the failover of each replica (with PLB minimizing the product of the move costs and the resource imbalance in its scoring function). Obviously, tenant (in)activity needs to be traded off against the resource usage of the moved tenants. We found the following equation to perform well in practice, with disk and memory usage specified in MB, and $Activity \in \{0, 1\}$ denoting whether a tenant is active:

$$MoveCost = \alpha \log_2(DiskUsage) + \beta \log_2(MemUsage) + \gamma \cdot Activity$$

Here, the constants α, β, γ are calibrated offline to minimize aggregate service disruption. Tenants are considered to be inactive if they have no active workers, no active sessions and no active requests for a sufficient period of time (with the time-period varying between *Provisioned* and *Serverless* tenant classes). Because Service Fabric only supports a limited number of distinct move cost settings, the resulting move costs are bucketized into 3 buckets of Low, Medium and High move costs. We evaluate the effects of move costs on overall service disruption in Section 8.3.

Handling New Tenants. The approach described above, which estimates future failovers, optimizes for the expected resource usage of tenants. However, newly placed tenants present a specific challenge: Such tenants tend to grow very quickly early in their lifetime[5] and tend to be highly active; in fact, a significant fraction of tenants are placed, do some continuous computation and then are deleted within a few hours. As this type of tenant is both highly active and common, we introduce a technique to ensure that, for newly placed tenants, the available resources on a node correspond not only to the *expected* usage of all tenants on the node, but also that free resource capacity exists for new tenants to "grow into".

For this purpose, we first record, for every tenant class, the 90^{th} percentile of resource consumption that members of this class achieve within the first 24 hours of their lifetime, based on historical traces of previously seen tenants and broken down by primary/secondary replicas. When initially placing a tenant, we report this 90^{th} percentile to PLB as the initial resource demand for the new tenant. Implicitly, this ensures that the corresponding replicas are only placed on nodes that likely can accommodate the corresponding resource growth. As we show experimentally in Section 8.3, this approach reduces the incidence of resource violations further than the use of the probability of violation estimates are able to, without a noticeable reduction in overall cluster capacity.

7 MITIGATING FAILOVER IMPACT

Failing over a tenant involves aborting all its currently running transactions, forwarding all client connections to a new physical target node and attaching the database files to a new database engine process hosted on that node. Unfortunately, this also voids any non-persistent state of the failed-over database process, such as in-memory caches that can have a substantial performance impact.

To address this issue, we deploy low-overhead mechanisms that preserve the contents of important caches across failovers. Our mechanisms are customizable to selectively migrate only the portions of the cache with large impact on database performance, thereby limiting the cost of migration. Due to space constraints, this section focuses on migrating the state of the buffer pool. Migrating caches such as the query plan and the column segment caches follows similar ideas, but requires the inclusion of cache-specific metadata to ensure cache entries can be efficiently reconstructed.

Buffer Pool. Because a network transfer within a cluster is significantly faster than reading from remote storage in the cloud, the buffer pool can be efficiently re-hydrated by directly copying the buffer pool content from the source node to the target node of the failover. To that end, we employ a push-based *iterative pre-copy live migration* scheme that copies a snapshot of the buffer pool to the target node, while continuing normal query execution on the source node. During the migration, the contents of the buffer pool are iteratively transferred to the target node using a background task until either the rate of newly modified pages stabilizes or the the transfer takes longer than a safe timeout, at which point the ownership of the connection is transferred to the target node.

During the ownership transfer, all existing write transactions are first paused, after which the remaining modified pages are migrated

[5] In fact, the vast majority of tenants achieve 95% of their maximal resource consumption within the first 5% of their lifetime, for both disk and memory usage.

to the target node. Once the target node is considered caught up, the source node cancels all open transactions and releases its handle on the database files. The target node then opens the database files and resolves any potential inconsistencies in the migrated buffer pool. Because our implementation allows partial copies of the buffer pool, at the time connections are switched to the target database process, some migrated pages could be stale or contain dirty changes that were rolled back after the respective page was copied. However, in databases that rely on a no-force/steal recovery strategy (such as ARIES [40]), any inconsistencies in the migrated buffer pool can be corrected at recovery time, provided that only consistent pages were copied and recovery is started from an LSN that is older than the oldest dirty page LSN at the time the transfer begins.

8 EXPERIMENTAL EVALUATION

This section presents an empirical evaluation of the multi-tenant resource allocation techniques described in this paper. Our experiments use either a variation of the standard TPC-C benchmark [54] or Microsoft's Cloud Database Benchmark (CDB) [16]. We note that due to the sensitive nature of customer workloads running in Azure SQL DB production clusters, no controlled end-to-end experiments on production clusters are possible. However, where appropriate, we use resource traces obtained from production clusters to drive our experiments. The experiments were conducted on a local cluster made up of nodes with dual-socket AMD EPYC 7352 processors (24 cores each) and 256 GB of DRAM, with all data and log files stored on a remote SSD hosted in the same cluster; providing similar performance characteristics to SQL Database's *General Purpose* service tier. We limit the available IOPS for each database according to their equivalent Azure SQL Database SKU [22].

8.1 Multi-Tenant Memory Brokering

Multi-tenant memory brokering (Section 4) aims to reduce the number of memory-capacity violations without significantly impacting performance. Below, we evaluate the effectiveness of our VoM-based, database-aware multi-tenant memory brokering technique against three database-agnostic brokering policies, which represent some of the ways in which users can configure memory sharing across tenants using VM- or OS-level memory governance.

We compare with two intuitive process-level brokering schemes that reclaim the required M units of memory either uniformly across all tenants (Equal) or proportionally to each tenant's current memory consumption (Proportional). In addition, we use a black-box memory reclamation approach inspired by memory ballooning (Balloon) [55] that reclaims the required memory by creating system-wide memory pressure. For this, we use a balloon driver that allocates and pins enough physical node memory to create memory pressure equivalent to the required M units of memory, which causes each SQL instance to independently release memory until the OS-level memory pressure has been relieved.

We create a test setup with two 8-core databases, DB_1 and DB_2, which differ in their degree of activity and, therefore, in the recency of their buffer pool contents. Together, both databases share 26 GB of main memory and are running the identical workload (TPC-C). Each database is guaranteed a minimum of 3 GB of main memory. To highlight the impact of both recency and relative database size,

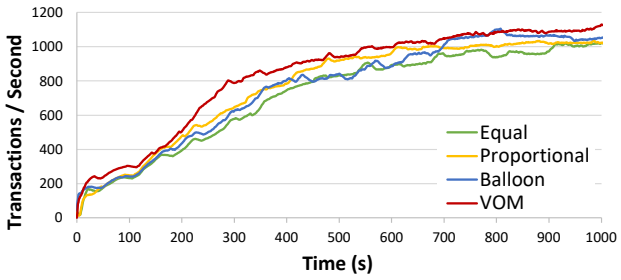


Figure 8: TPC-C transaction throughput for different memory reclamation policies (2-minute moving average)

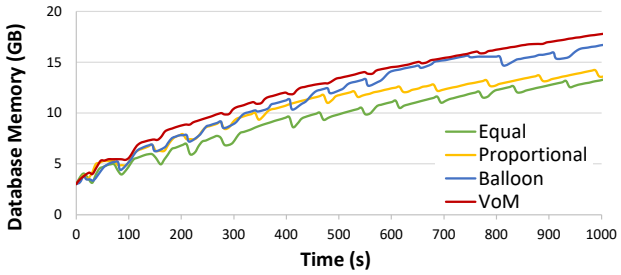


Figure 9: Memory consumption of an actively growing database instance using different memory reclamation policies

we focus on a common case that sees a warm database grow memory by reclaiming memory from a colder, inactive database. Memory brokering is triggered whenever the aggregate memory consumption on the node crosses a threshold of 25 GB, but at most once per minute. If memory consumption reaches the node maximum of 26 GB in between two consecutive invocations of the memory broker, memory throttling caps each database at its current memory consumption. The reclamation target is set to 24 GB, resulting in a reclamation of 1–2 GB per memory-broker invocation.

We run a high-contention variation of TPC-C with 500 warehouses (~50GB of data), 1 worker thread per warehouse and no keying time and think time. Both database instances are warmed up to ensure they fully utilize their minimum memory. To create an environment with a large but cold database, we start the main phase of the experiment by running DB_1 in isolation until it reaches stable performance and consumes most of the node's memory; followed by 30 minutes of inactivity to let its buffer pool pages become cold. At this point DB_1 (cold) and DB_2 (warm) consume 22 GB and 3 GB of main memory, respectively. We now start the workload on DB_2, which, given the limited memory size of the node, will only be able to grow its memory footprint by reclaiming memory from DB_1, which highlights the performance impact of each policy for different database sizes,

Results. Figure 8 shows the 2-minute moving average of transactions per second (TPS) of DB_2 as it ramps up execution after the workload on DB_1 stops. Note that it is during the first 10 minutes of this ramp up phase where the database needs access to additional memory at a high rate, and hence where the impact of the multi-tenant brokering policy on performance is felt the most. We observe that all policies perform significantly worse compared to

VoM. During this ramp up period, compared to the next best policy, Balloon, the median TPS improvement for VoM is 14% and the 95th percentile TPS improvement is 26%. The other two policies, Proportional and Equal suffer even more significantly in TPS.

To explain the differences in performance between the policies, we compare the memory consumption of DB_2 for the same time period (shown in Figure 9). As expected, because DB_1's memory is largely cold, using VoM heavily skews memory reclamation towards DB_1, allowing DB_2 to grow quickly. In particular, most of DB_1's buffer pool pages have a considerably lower VoM than new pages accessed by DB_2. Accordingly, DB_2 grows primarily by reclaiming globally cold memory from DB_1. While using a memory Balloon allows DB_2 to grow quickly, it performs badly during the most critical phase of the initial ramp up. This happens because memory pressure is observed by all DBMS processes at the same time and, since there is no coordination between tenants, all databases, regardless of their activity, must respond by releasing memory. In particular, in response to this pressure DB_2 gives up a substantial amount of memory even though its pages are actively being used, thereby significantly impacting performance.

The Proportional and Equal strategies demonstrate important shortcomings. Because DB_2 starts execution while only consuming 12% of the node's memory, Proportional reclamation quickly assigns new memory to DB_2 by reducing DB_1's memory budget. However, as the memory consumption of the new database increases, proportional reclamation becomes less and less effective, even though DB_1 consists only of cold memory. In contrast, Equal reclamation outperforms Proportional reclamation when the active database is comparatively large, but cannot match Proportional or VoM-based reclamation during initial ramp-up.

Cluster Deployment. To highlight the importance of database-aware memory brokering in real-world DBaaS environments, we measured the impact of deploying multi-tenant memory brokering on a representative 200-node production cluster with multiple databases on each node. After introducing multi-tenant memory brokering, we observed a 68% decrease in the number of memory-capacity induced failovers, without a noticeable change in database performance characteristics such as throughput and latency.

8.2 Utilization-based CPU Rebalancing

To highlight the importance of dynamically rebalancing database CPU assignments across cores, we compare the dynamic utilization-based rebalancing technique described in Section 5 with a *static* utilization-based core assignment scheme where no rebalancing occurs after the initial core assignment of a database. We show that rebalancing is necessary to reduce the number of hot oversubscribed cores, and verify that not re-assessing core assignments periodically can have a substantial impact of workload performance.

To this end, we create a set of eight 8-core databases that share 32 physical cores at an oversubscription ratio of 2× and vary their CPU utilization over time. To create a CPU-heavy workload distribution, we run CDB (SF=2) with a workload consisting 100% of *CPU-Heavy* transactions and vary the activity level of each database between a High-CPU and a Low-CPU variation, with 48 and 16 active connections per database, respectively. We create a bursty workload pattern with shifting CPU demand by creating 20-minute phases in which we randomly select a database to boost to the High-CPU state for 15 minutes, followed by a 5 minute pause. All remaining databases during a burst run the Low-CPU workload. To account for the inherent randomness of the workload and core allocation, we execute 5 distinct 3-hour runs for each configuration and report averages and standard deviation. We measure the physical core utilization of the host as the average in discrete 15-second intervals. Lastly, because initial core allocation is based on currently observed core utilizations, we allow each database 5 minutes for initial warm up, before placing the next database on the node. This prevents fluctuations in CPU utilization, which occur early in the database's lifetime, from influencing the core assignment. To ensure stable results we made sure all databases are fully warmed up before starting any measurements.

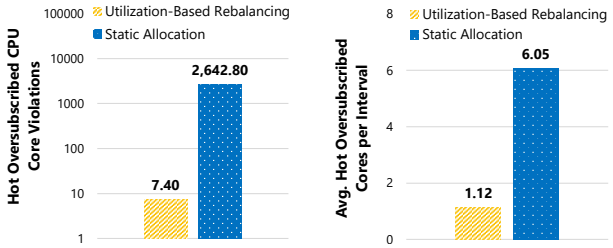


Figure 10: Number of hot oversubscribed CPU cores during a bursty CDB workload, in distinct 15-second intervals

Hot Oversubscribed CPU Cores. We define a core as *hot* if its average utilization during the measurement interval is ≥ 95%, and as *over-subscribed* if it hosts at least 2 active databases at any point in the interval. We first show that, for a workload with shifting CPU utilization, rebalancing is necessary to reduce the number of hot oversubscribed CPU cores. Hot CPU cores are problematic, because they can stall the execution of co-located database cores of other tenants. Figure 10 shows the average number of hot oversubscribed CPU cores per experiment run. We count an average of 2642.8 (SD=141.4) hot oversubscribed cores occurring in 436.8 (SD=2.2) distinct 15-second measurement intervals. During each interval that saw at least one hot core, we observed an average of 6.0 (SD=0.3) hot cores belonging to on average 4 co-located tenants. With utilization-based rebalancing enabled, the number of hot oversubscribed cores drops significantly, to an average of 7.4 (SD=4.5) in 6.4 (SD=3.5) distinct measuring intervals, with an average of 1.1 (SD=0.1) hot cores per interval and affecting on average 2.1 co-located tenants.

Workload Performance. Hot CPU cores can have a significant impact on the overall performance of co-located tenants. Spiking CPU utilization of co-located tenants can cause an increase in query latencies even when the physical node has sufficient resources available to accommodate all active tenants. To demonstrate this point, we create heavy CPU-skew by actively running CDB on only 4 of the 8 database tenants and intentionally creating a core allocation scenario where the active tenants are completely overlapping; leaving half the cores inactive. We ran CDB at a level such that each database utilizes at least 50% of each core. This ensures that any two active databases co-located on a core will interfere with each other's performance. Figure 11 shows the average CPU-utilization

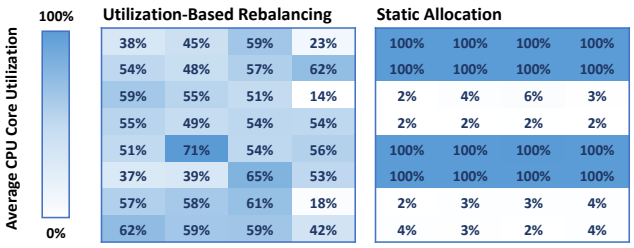


Figure 11: Average CPU utilization of all database cores on a node while executing CPU-heavy CDB, with heavy skew

for each database core on the test node during the execution of a single burst. We can see that, given the unfavorable initial placement of tenants, not re-balancing CPU assignments among tenants results in multiple shared host CPU cores reaching an average utilization of 100%, even though the machine itself has sufficient CPU resources available to accommodate the active databases. Meanwhile utilization-based rebalancing visibly spreads out the core assignment by co-locating the active database tenants with the 4 idle database tenants. In this experiment, not rebalancing database tenants results in a transaction throughput drop of 20% on all active databases (from an average 335 TPS to 280 TPS), as well as an increase in query latency from 0.016/0.032/0.047 to 0.141/0.250/0.297 seconds for 50/95/99th percentile, respectively.

8.3 Tenant Placement

We evaluate the tenant placement technique of Section 6 by comparing the performance of our modified PLB to the unmodified version. More details, as well as experiments comparing to other approaches, such as [31, 46, 48], can be found in [35]. We conduct the experiments using a high-fidelity cluster-level PLB simulator developed for Service Fabric [32], which tracks resource violations and tenant moves in simulated clusters. To simulate tenant resource usage, we use 7-day resource demand traces for a set of Azure SQL Database production tenants sampled uniformly at random from two different geographical regions with significantly different resource usage profiles. Each trace contains CPU, memory and local disk usage at 10-minute granularity. We evaluate the techniques on a 40 node cluster. Nodes are divided into 10 upgrade domains and 5 fault domains (selected using the logic described in [4]).

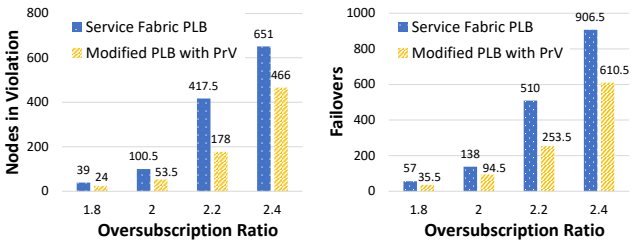


Figure 12: Average violations and failovers for tenants from different Azure regions using PLB simulator

Figure 12 shows the average number of violations and failovers across both regions at multiple oversubscription ratios. The unmodified Service Fabric exhibits, on average, 1.83× as many violations, and 1.67× as many failovers as our proposed technique. This highlights the importance of accounting for resource demand changes over time when making placement and load-balancing decisions.

Real Cluster Deployment. To show that the observed gains hold when deploying to a real cluster, we repeat the above experiment on a real 40-node cluster within Microsoft Azure [5].

To obtain realistic resource demand profiles without executing real customer SQL workloads (which is not possible, as workloads constitute customer IP), each application reports to PLB resource usage corresponding to real customer traces, selected at random. Because of the time required, we repeat the experiment for only one region; each experiment covers a week of time.

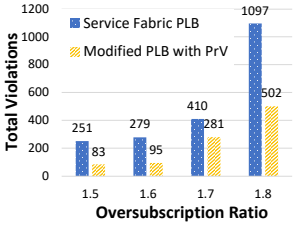


Figure 13: Violations seen for real cluster deployment

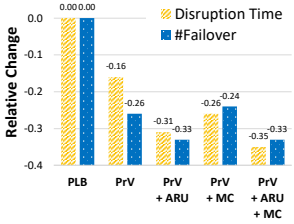


Figure 14: Service quality improvements for different algorithm combinations

The results are shown in Figure 13. The observed reduction in violations is similar to the simulation-based experiments, with the unmodified Service Fabric seeing, on average, 2.4× as many violations as PLB with probability of violation estimates.

Evaluating the Effects of Move Costs and ARU. To evaluate the effect of the specific techniques used to (a) assign weights to different tenants and (b) prevent failovers for new tenants, we conducted additional experiments where we evaluate different combinations of algorithms and measure (1) the total number of failovers, and (2) the aggregate *disruption time* experienced by *active* tenants (i.e., the time during which one or more replicas are being failed over). The experimental setup is similar to the previous simulator-based experiment. We executed this experiment using traces from a single region at an over-subscription ratio of 1.8. We measure the performance of the unmodified PLB code (PLB), the technique to estimate the probability of future violations (PrV) and combine these with (a) *move costs* to assign weights to different failovers (MC) and (b) the approach to reserve resources on nodes hosting new tenants based on previously observed *aggregate resource usage* (ARU).

The results are shown in Figure 14. The use of MC consistently reduces the overall disruption time by a significant percentage, when compared to the same algorithm(s) without MC. The same holds for ARU with respect to the number of failovers. Moreover, adding MC and ARU does not lead to any degradation of the metric they are not targeting.

8.4 Failover Performance

To assess the performance impact of preserving database caches across failovers, we ran an experiment that fails over an active 16-core database to a new node in the cluster. Figure 15 shows the

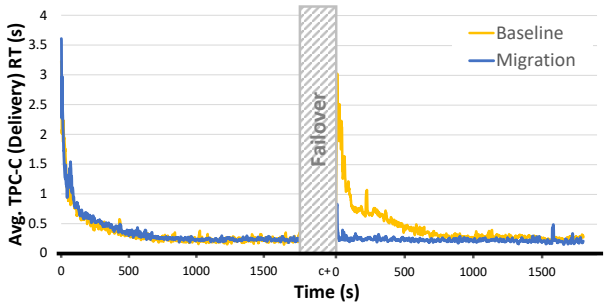


Figure 15: Failover impact on average query response time on TPC-C's delivery transaction, with (MIGRATION) and without (BASELINE) preserved buffer pool content (500 WH)

average response time for delivery transactions of an active TPC-C workload with 500 Warehouses (~50GB of data), 1 worker per warehouse and no keying/think time. Because the transfer time of the buffer pool depends on the cluster's network topology, we omit the exact time consumed by the network transfer. We observe that, when a new database process is provisioned with a cold buffer pool, the average latency of the workload spikes significantly as query execution becomes bottle-necked by the high number of remote storage reads from re-hydrating the buffer pool. In our specific setup it took the workload more than 15 minutes to fully revert to pre-failover latencies. In contrast, when buffer pool contents are migrated asynchronously using a direct network connection we only see a short increase in average latency, which is a result of active transactions being aborted and restarted on the target node.

9 RELATED WORK

Cloud providers use oversubscription for different resources to control costs. For instance, power oversubscription and power capping [28, 38, 49] are used to reduce data-center costs by keeping the power consumption in data-centers below circuit breaker limits, while increasing the number of machines hosted on the same infrastructure. Similarly, virtualization technology used by cloud providers, such as VMs or containers, allows oversubscription of CPU, memory and I/O resources using mechanisms in the hypervisor or OS [8, 18, 45]. These black-box techniques were developed to work with arbitrary applications. In contrast, our white-box approach exploits our knowledge of how DBMSs use resources internally to ensure minimal impact on performance, thereby enabling much higher degrees of oversubscription.

There is a body of work related to multi-tenant resource management for databases. SQLVM [26, 42, 43] studies multi-tenant buffer pool and CPU sharing across databases within a *single* DBMS instance. In contrast, our architecture is designed for databases in separate containers (i.e., each DBMS is a different process) thereby raising the need for a common currency across tenants. In addition, our multi-tenant memory brokering technique extends to multiple kinds of DBMS caches beyond the buffer pool. [36] studies the benefit and cost of shutting down idle databases and resuming them on-demand. Our approach is complementary and applies to any set of *currently executing* databases by dynamically redistributing resources across databases. In [39], the authors describe infrastructure in Azure SQL DB to benchmark the impact on database performance due to resource contention across tenants. Such an infrastructure is useful in quantifying the impact of oversubscription.

The problem of efficient tenant placement and consolidation on a cluster of nodes has been investigated in the context of arbitrary workloads (for example, [10, 31, 46, 48]), where tenant placement decisions are based on a snapshot of resource usage in the cluster. In comparison, our approach also factors in the change in resource usage over time. For databases, which tend to be long-lived, we show that this approach can lead to better placement that significantly reduces resource shortages. [37] proposes a system for resource optimization of cloud database services, with resources allocated statically to replicas (while in our scenario they are shared dynamically among co-located replicas), including a variant of the *Best Fit* heuristic with logic to avoid skew/fragmentation. In [50] the authors study a the problem of minimizing the number of servers needed to place a given set of databases subject to constraints on node load and quality of service. In contrast, in our scenario, the cluster size is fixed and minimizing the incidence of resource violations is the optimization criterion. There is work on modeling database resource usage for the purpose of database consolidation, e.g., [24, 53]. These techniques observe the demand of each tenant, use these observations to predict future usage, and subsequently consolidate tenants on fewer nodes. However, the consolidation itself requires failing over new tenants at least once, making the approach impractical, as we seek to avoid failovers altogether.

Finally, live migration of VMs and databases has been studied before e.g., [11, 27]. However, these techniques are too heavyweight for our scenario, since they can move large amounts of database memory and storage state, and can take a long time to converge. Meanwhile, the buffer pool migration scheme described in this paper enables time-bound transfers that move only the most important pages in the buffer pool. However, because our technique relies on aborting running transactions on the source node, it can create visible impact on query latencies. This is acceptable in our setting where most failovers affect relatively inactive tenants and single tenants are unlikely to be affected twice in a row.

10 CONCLUSION

Reducing COGS is a major challenge for DBaaS providers, in particular in light of the growing trend of serverless databases. Resource oversubscription is a valuable tool to achieve this goal by improving the resource utilization of database clusters. We have developed flexible resource allocation techniques for multi-tenant DBaaS architectures that leverage our understanding of how database performance is affected when server resources are constrained. This enables a higher packing density of databases than otherwise possible, while providing the ability to control the impact on database performance. Our evaluation, performed in the Azure SQL Database service, shows significantly better performance for databases using our approach compared to generic oversubscription techniques that are not database-aware. This paper lays the groundwork for future research on resource allocation in multi-tenant DBaaS, e.g., improving efficiency by controlling the decision on how databases are assigned to clusters within a data center, and enabling trade-offs between performance and cost in serverless databases.

REFERENCES

[1] AWS. 2021. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/ Last accessed on Sep 27, 2023.

[2] AWS. 2023. Amazon Aurora. http://aws.amazon.com/rds/aurora/ Last accessed on September 27, 2023.

[3] Microsoft Azure. 2020. Configuring and using Service Affinity in Service Fabric. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-advanced-placement-rules-affinity Last accessed on September 27, 2023.

[4] Microsoft Azure. 2021. Service Fabric Cluster Resource Manager. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description Last accessed on September 27, 2023.

[5] Microsoft Azure. 2022. Create a Service Fabric Cluster. https://docs.microsoft.com/en-us/azure/service-fabric/scripts/service-fabric-powershell-create-secure-cluster-cert Last accessed on September 27, 2023.

[6] Microsoft Azure. 2022. Describe a Service Fabric cluster by using Cluster Resource Manager. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-cluster-description#node-properties-and-placement-constraints. Last accessed on September 27, 2023.

[7] Microsoft Azure. 2022. Service Fabric Movement Cost. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-movement-cost Last accessed on September 27, 2023.

[8] Ishan Banerjee, Fei Guo, K. Tati, and R. Venkatasubramanian. 2014. Memory Overcommitment in the ESX Server.

[9] Salman Abdul Baset, Long Wang, and Chunqiang Tang. 2012. Towards an Understanding of Oversubscription in Cloud.. In *Hot-ICE*.

[10] Sebastian Berndt, Klaus Jansen, and Kim-Manuel Klein. 2020. Fully Dynamic Bin Packing Revisited. *Mathematical Programming* (2020). https://link.springer.com/article/10.1007/s10107-018-1325-x, Last accessed: September 27, 2023.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.

[12] Microsoft Corporation. 2018. Service Fabric. https://github.com/Microsoft/service-fabric/ Last accessed on Last accessed on September 27, 2023.

[13] Microsoft Corporation. 2021. Azure SQL Database and Azure SQL Managed Instance Service Tiers. https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tiers-general-purpose-business-critical Last accessed on September 27, 2023.

[14] Microsoft Corporation. 2021. Azure SQL DB. https://docs.microsoft.com/en-us/azure/sql-database/ Last accessed on September 27, 2023.

[15] Microsoft Corporation. 2021. Azure SQL DB Serverless. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-serverless/ Last accessed on September 27, 2023.

[16] Microsoft Corporation. 2021. DTU Benchmark. https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tiers-dtu#dtu-benchmark Last accessed on September 27, 2023.

[17] Microsoft Corporation. 2021. Job Objects — Win32 apps | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/procthread/job-objects Last accessed on September 27, 2023.

[18] Microsoft Corporation. 2021. JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure (winnt.h) | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-jobobject_cpu_rate_control_information Last accessed on September 27, 2023.

[19] Microsoft Corporation. 2021. Managing Resource Consumption and Load in Service Fabric with Metrics. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-metrics Last accessed on September 27, 2023.

[20] Microsoft Corporation. 2021. *SQL Server Columnstore indexes: Overview*. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15 Last accessed on September 27, 2023.

[21] Microsoft Corporation. 2022. SQL Server, Buffer Manager object. https://learn.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-buffer-manager-object?view=sql-server-ver16 Last accessed on September 27, 2023.

[22] Microsoft Corporation. 2023. Single database vCore resource limits - Azure SQL Database. https://learn.microsoft.com/en-us/azure/azure-sql/database/resource-limits-vcore-single-databases?view=azuresql Last accessed on September 27, 2023.

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

[24] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 313–324.

[25] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[26] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. 2013. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-service. *Proc. VLDB Endow.* 7, 1 (Sept. 2013).

[27] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (may 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[28] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. Power Provisioning for a Warehouse-Sized Computer. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 13–23. https://doi.org/10.1145/1273440.1250665

[29] The Linux Foundation. 2021. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/ Last accessed on September 27, 2023.

[30] Google. 2021. Google Cloud SQL. https://cloud.google.com/sql/. Last accessed on September 27, 2023.

[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[32] 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. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[33] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680.

[34] Scott Konersmann, Slava Oks, and Tobias Ternstrom. 2016. SQL Server on Linux: How? Introduction. https://cloudblogs.microsoft.com/sqlserver/2016/12/16/sql-server-on-linux-how-introduction Last accessed on September 27, 2023.

[35] Arnd Christian König, Yi Shan, Tobias Ziegler, Willis Lang Aarati Kakaraparthy, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. 2022. Tenant Placement in Over-subscribed Database-as-a-Service Clusters. *PVLDB* 15, 1 (2022), 2559–2571.

[36] Willis Lang, Karthik Ramachandra, David J DeWitt, Shize Xu, Qun Guo, Ajay Kalhan, and Peter Carlin. 2016. Not for the Timid: On the Impact of Aggressive Over-booking in the Cloud. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1245–1256.

[37] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud. *Proc. VLDB Endow.* 16, 12 (Sep 2023), 3795–3807.

[38] Sulav Malla and Ken Christensen. 2019. A Survey on Power Management Techniques for Oversubscription of Multi-Tenant Data Centers. *ACM Comput. Surv.* 52, 1, Article 1 (Feb. 2019), 31 pages. https://doi.org/10.1145/3291049

[39] Justin Moeller, Zi Ye, Katherine Lin, and Willis Lang. 2021. Toto - Benchmarking the Efficiency of a Cloud Service. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2543–2556. https://doi.org/10.1145/3448016.3457555

[40] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (mar 1992), 94–162. https://doi.org/10.1145/128765.128770

[41] Vivek Narasayya and Surajit Chaudhuri. 2021. Cloud Data Services: Workloads, Architectures and Multi-Tenancy. *Foundations and Trends in Databases* 1 (2021), 1–107. https://doi.org/10.1561/1900000060

[42] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. (2013).

[43] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment* 8, 7 (2015), 726–737.

[44] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering *(SIGMOD)*.

[45] Linux Kernel Organization. 2021. Control Group v2 — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. Last accessed on September 27, 2023.

[46] Rina Panigrahy, Vijayan Prabhakaran, Kunal Talwar, Udi Wieder, and Rama Ramasubramanian. 2011. *Validating Heuristics for Virtual Machines Consolidation*. Technical Report MSR-TR-2011-9. https://www.microsoft.com/en-us/research/publication/validating-heuristics-for-virtual-machines-consolidation/

[47] Jose Picado, Willis Lang, and Edward C. Thayer. 2018. Survivability of Cloud Databases - Factors and Prediction. In *ACM SIGMOD*. 811–823.

[48] Kamali S. 2015. Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud. *ALGOCLOUD* (2015).

[49] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. 2020. Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 497–511. https://doi.org/10.1145/3373376.3378533

[50] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs. 2013. RTP: Robust Tenant Placement for Elastic in-Memory Database Clusters. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 773–784. https://doi.org/10.1145/2463676.2465302

[51] Michael Stonebraker. 1981. Operating system support for database management. *Commun. ACM* 24, 7 (1981), 412–418.

[52] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2 *(VLDB)*.

[53] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 388–400. https://doi.org/10.1145/2987550.2987575

[54] TPC. 1992. TPC-C Benchmark. http://www.tpc.org/tpcc/ Last accessed on September 27, 2023.

[55] Carl A Waldspurger. 2002. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 181–194.

[56] Wikipedia. 2021. Knapsack Problem. https://en.wikipedia.org/wiki/Knapsack_problem Last accessed on September 27, 2023.

[57] Pavel Yosifovich, Mark E. Russinovich, David A. Solomon, and Alex Ionescu. 2017. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More (7th Edition)* (7th ed.). Microsoft Press, USA.