A system design for elastically scaling transaction processing engines in virtualized servers

Angelos-Christos Anadiotis[‡], Raja Appuswamy[*], Anastasia Ailamaki[†],
Ilan Bronshtein[◁], Hillel Avni[◁], David Dominguez-Sal[▷], Shay Goikhman[◁], Eliezer Levy[◁]

[‡] Ecole Polytechnique & EPFL, [*] EURECOM, [†] EPFL & RAW Labs
[◁] Huawei Tel Aviv Research Center, [▷] Huawei Munich Research Center
{name.surname}@{[‡]polytechnique.edu, [*]eurecom.fr, [†]epfl.ch, [◁▷]huawei.com}

ABSTRACT

Online Transaction Processing (OLTP) deployments are migrating from on-premise to cloud settings in order to exploit the elasticity of cloud infrastructure which allows them to adapt to workload variations. However, cloud adaptation comes at the cost of redesigning the engine, which has led to the introduction of several, new, cloud-based transaction processing systems mainly focusing on: (i) the transaction coordination protocol, (ii) the data partitioning strategy, and, (iii) the resource isolation across multiple tenants. As a result, standalone OLTP engines cannot be easily deployed with an elastic setting in the cloud and they need to migrate to another, specialized deployment.

In this paper, we focus on workload variations that can be addressed by modern multi-socket, multi-core servers and we present a system design for providing fine-grained elasticity to multi-tenant, scale-up OLTP deployments. We introduce novel components to the virtualization software stack that enable on-demand addition and removal of computing and memory resources. We provide a bi-directional, low-overhead communication stack between the virtual machine and the hypervisor, which allows the former to adapt to variations coming both from the workload and the resource availability. We show that our system achieves NUMA-aware, millisecond-level, stateful and fine-grained elasticity, while it is not intrusive to the design of state-of-the-art, in-memory OLTP engines. We evaluate our system through novel use cases demonstrating that scale-up elasticity increases resource utilization, while allowing tenants to pay for actual use of resources and not just their reservation.

PVLDB Reference Format:
Angelos Christos Anadiotis, Raja Appuswamy, Anastasia Ailamaki, Ilan Bronshtein, Hillel Avni, David Dominguez-Sal, Shay Goikhman, Eliezer Levy. A system design for elastically scaling transaction processing engines in virtualized servers. *PVLDB*, 13(12): 3085-3098, 2020.
DOI: https://doi.org/10.14778/3415478.3415536

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415536

1. INTRODUCTION

The past few years have witnessed a growth in the popularity of cloud-hosted Online Transaction Processing (OLTP) engines. Today, both traditional enterprise applications and modern web services increasingly use OLTP engines as transactionally consistent storage back-ends. In addition to being able to provide with high throughput and low latency for application transactions, these OLTP engines also have to provide resilience to unpredictable variations in demand from the application workloads. Such variations are a norm, rather than an exception, in emerging application workloads where factors like skew due to few popular records, diurnal access patterns, and hockey-stick effect due to sudden increase in popularity of an application, result in sudden, unpredictable load spikes that OLTP engines must handle without scalability issues [9].

Traditionally, on-premise OLTP engines dealt with load spikes by over provisioning hardware resources. However, this approach has three problems. First, it leaves a substantial amount of resources underutilized in periods of normal activity. Second, it may not be possible in many cases to predict the peak of a load spike and, hence, over-provisioning will fail to meet performance requirements under excessively high loads. Third, upgrading a database installation in the on-premise case often requires a cumbersome forklift upgrade with long outages due to data reorganization. The growing popularity of public/private cloud infrastructures for hosting enterprise databases, has made it feasible today to right-provision computational resources and dynamically provision additional resources on the fly. Such elasticity in resource provisioning is often touted as one of the major benefits of migrating on-premise installations to the cloud.

Lately, several elastic OLTP engines exploit elasticity to deal with unpredictable load spikes in their workloads. A big part of existing elastic approaches focus on distributed, scale-out, shared-nothing OLTP engines, where elasticity is implemented through dynamic data redistribution. This is achieved by monitoring data access patterns, identifying an optimal partitioning of data across nodes in a shared-nothing cluster, and perform background data reorganization without any application down time [14, 35, 39]. By doing so, these OLTP engines can effectively meet load spikes in the workload by relying on cloud infrastructure elasticity to dynamically perform resource allocation and data migration in tandem. Other approaches addressing VM placement and migration, and query operator placement for cloud data

analytics [20, 21, 26, 28], do not consider the special requirements of an OLTP engine which needs to maintain its state consistent, while meeting its performance guarantees.

Another design approach for cloud-hosted OLTP engines is to enable multi-tenancy from within the database, effectively offering database-as-a-service. AzureSQL, which is based on the SQLVM research prototype [15, 16, 27], and AWS Aurora [40, 41] are two representative examples in this category. These engines introduce a novel design, which shifts the standard OLTP engine architecture and focuses on fine-grained resource isolation across tenants as well as on availability and fault-tolerance, offered at cloud-scale. However, following this approach requires tenants to switch from their own standalone engine deployment to a different one, which is hosted by the cloud provider.

As prior work focuses either on the challenges of elastically scaling out OLTP or on new elastic database designs, it fails to take into account two important aspects of the cloud infrastructure and the application workloads. On the infrastructure front, cloud providers have started offering more "beefier" hosted instances that offer a way to scale-up services instead of scaling out using multi-socket multi-core servers. For instance, Amazon EC2 offers instances packing up to 488 vCPUs and 12TB of DRAM per instance. On the application front, it is well-known that most OLTP installations comfortably fit in the memory of a single large-memory server [24, 34]. Several OLTP engines have been designed based on this argument to first scale-up on shared-everything servers before scaling out to a shared-nothing cluster. Over the past few years, such shared-everything OLTP engines are also migrating to the cloud and are increasingly being deployed on scale-up cloud instances. For instance, SAP HANA on EC2 can run on servers that contain up to 488 vCPUs and 24TB of DRAM [6].

As modern multi-socket servers transition from multi-core to many-core setups [3, 4, 5], cloud instances that offer several hundreds of CPUs are available for deploying shared-everything OLTP installations. Achieving elasticity in such a scale-up scenario is a different problem than the scale-out case. First, in a scale-up scenario, all data are stored in globally-accessible, cache-coherent shared memory. Thus, the issues of data partitioning and migration that play a crucial role in scale-out elasticity are not relevant in the scale-up context. Second, the granularity at which resources are allocated in the scale-out scenario is an instance with multiple cores. Expanding an OLTP installation is done by starting the engine on the new instance, repartitioning data as necessary, and modifying the distributed transaction execution machinery to redirect transactions to the new instance. In the scale-up context, resource allocation can happen at a much finer granularity of a core or a socket. However, to exploit such flexibility, an OLTP engine should be able to react to dynamic changes in the underlying resource availability. While prior work focuses on elasticity in either a scale-out or a multi-tenant databases context, there has been no study, to our knowledge, that targets such scale-up scenarios.

In this paper, we present a system design providing fine-grained elasticity for shared-everything, scale-up OLTP systems deployed in virtualized multi-socket, multi-core servers. Our design spans through the whole virtualization software stack and introduces novel components in the hypervisor and the virtual machine (VM), which enable the dynamic addition and removal of computing and memory resources. We introduce two-way communication protocols between the hypervisor (host) and the VM (guest), which enable guest applications to request resources from the hypervisor, and vice versa. We provide mechanisms guaranteeing compliance of the host and the guest to resource allocation and usage. We evaluate our system through novel and practical use cases that outline the benefits of scale-up elasticity in OLTP engines and can be used as a basis for more complex ones. In summary, we make the following contributions:

- We show that our design is non-intrusive to the current state-of-the-art in-memory OLTP engines, which makes it generalizable to other stateful cloud applications with an elastic thread and memory pool.
- We demonstrate that our system achieves millisecond-level, NUMA-aware, stateful, and fine-grained computing and memory elasticity allowing the fast reaction of the application to changes in the workload and in the availability of resources.
- We introduce a novel spot instance model which increases the resource utilization on large, virtualized servers, while allowing tenants to pay for the actual use of resources and not just for their reservation.
- We show that our approach, albeit more generic, does not impose overheads compared to specialized, multitenant databases, whereas it eliminates distributed transaction execution overheads that cause at least 8x performance degradation in our experiments.

2. BACKGROUND AND RELATED WORK

Our system focuses on several aspects of cloud-hosted OLTP engines. First, it provides an efficient solution for cloud deployments, offering elasticity and resource isolation. Second, it leverages this elasticity in order to provide an end-to-end adaptive system for scale-up servers. In this section, we present related work around these topics and we discuss the difference with our approach.

Resource isolation. SQLVM [27] isolates performance across tenants for Database as a Service applications. It focuses on fine-grained resource isolation, whereas it includes a telemetry component, which monitors the resource allocation and scheduling for each tenant to make sure that they meet the tenants' requirements. This design is leveraged in the context of a full system in [16] which focuses on performance isolation of CPU resources. This system achieves fine-grained isolation, where each tenant can take a fraction of the CPU time. Based on this feature, the system also provides dynamic resource scheduling, which can be adapted to the dynamics of the workload. This last point is further elaborated in [15], where the authors exploit the database telemetry infrastructure to devise efficient adaptive resource allocation policies. In addition, the system enables tenants to reason about the resources that they will use and the price that they will pay based on the performance of the engine, rather than the resources that they will use.

This approach focuses on fine-grained isolation in a multi-tenant database setting. Instead, our system delegates this task to the virtualization infrastructure, and focuses on the end-to-end design of a generic system, including both the VM where the database is running and the hypervisor. This way, the OLTP engine can be deployed on the same scale-up server with other applications that are based on our design.

Elastic Scale-out: E-store [39] and P-Store [38] build on H-Store [23] and provide an end-to-end design for elastic scale-out OLTP engines. Since scale-out systems mostly suffer from distributed transactions and load imbalance, E-store provides a repartitioning scheme which adapts data placement to the dynamic characteristics of the workload. Instead, P-Store predicts the workload patterns to adapt the resource allocation before the workload changes.

Accordion [35] provides elasticity through partitioning and minimizes data movement through dynamic partition placement. Accordion implements a server capacity estimator which monitors the bottlenecks on each server and uses this information to adapt its partitioning and achieve load balancing. ElasTras [14] packs small tenants together and scales out large tenants across servers. ElasTras achieves load balancing through tenant migration with tenants moving from the most to the least busy nodes. Tenant migration is supported through Albatross [17] which enables the seamless transfer of the database state and caches to the destination server. In Zephyr [19], the database migrates to new nodes without any service disruption by making use of both active instances: the one already running and the one that started after migration. However, this system comes with the cost of making the index structures immutable during the migration process. Slacker [11] migrates the database to a different machine without any downtime and through an automated interface which does not require much human intervention. It performs the migration with minimum processing overhead to avoid interference with other tenants.

The above systems focus on data repartitioning and computing migration. However, apart from their complexity, they often depend on workload properties, like partitionability, whereas from some point on, they pay the overhead of moving data over the network inside the data center. Especially in case the changes in the workload are short-lived, their reconfiguration overhead becomes more significant. On the other hand, our approach is more suitable for such cases, since it can reconfigure the system with minimal overheads.

3. DESIGN AND IMPLEMENTATION

Scale-up elasticity addresses workload variations by distributing resources to VMs deployed on a single server [12, 37]. Resource allocation decisions are performed by a Resource Manager following scheduling policies which assign resources based on workload requirements and Service Level Agreements (SLAs) of each VM. Accordingly, resources are moved across VMs at execution time and to support this, a virtualization infrastructure requires: (i) a two-way communication channel between the guest applications and the resource manager, (ii) separation of the resources that can be removed from the VM from the ones that are statically allocated to it, and, (iii) ability to add and remove resources dynamically at runtime at three different levels, namely guest application, guest operating system, and hypervisor. To address the above requirements, we designed a system that cuts vertically the virtualization software stack.

We focus on computing (CPU) and memory resources, since they are the most commonly used ones and have a significant impact on the performance of OLTP engines. Hypervisors like QEMU and vSphere, allow CPUs and memory to be hotplugged into VMs dynamically. However, this functionality is either available via external tools as it is intended

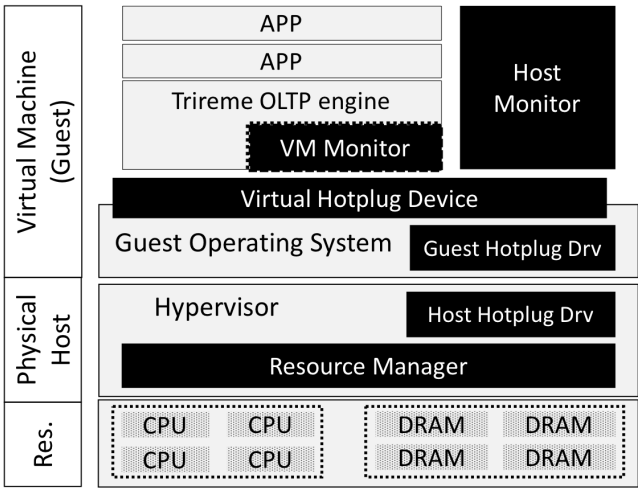

Figure 1: System architecture

to be an out-of-band operation that will be manually performed by an administrator [2], or does not involve the guest operating system which needs to be aware of the resources that have been hotplugged, in order to expose specialized control over them to the applications [1, 7, 37]. As a result, if the OLTP engine has to release specific memory blocks and CPUs and pass them to another tenant, it has no information about which part of its memory and worker pool corresponds to the memory and CPUs to be released, since such information is not exposed to the guest applications.

Figure 1 depicts the architecture of our elastic scale-up system which has been designed to meet the above requirements of cloud-hosted OLTP engines. In our design, we have introduced novel components in both the host and the guest sides of the virtualization software stack, indicated with dark color. The host side includes a modified version of QEMU hypervisor that provides support for dynamic addition and removal of CPUs and memory to VMs. We have extended the host–guest interface of QEMU to provide an API that can be used by OLTP engines running within VMs to poll for changes in the availability of resources, and to make explicit requests for addition or removal of CPUs and memory. On the guest side, the VM includes Trireme, an in-memory OLTP engine which can dynamically expand and shrink based on changes in the available resources. Trireme uses the VM Monitor which is responsible for collecting system and application level statistics. Finally, the VM also includes the Host Monitor which monitors the resource availability from the host. In the rest of this Section, we describe how these modules are used to provide scale-up elasticity.

We have chosen widely used and open source software modules, as this allows us to integrate our system design in GaussDB [10], which is deployed in Huawei Cloud, to provide scale-up elasticity. This allows us to take advantage of the multi-socket, virtualized scale-up Huawei servers.

3.1 Host–Guest Communication

The resources that can be assigned to each VM are maintained by the Resource Manager (RM). The RM has to communicate with all the VMs to receive requests and send notifications for changes in resource availability. This is achieved by virtualizing it as a Virtual Hotplug Device which is im-

		Hotplug		Poll	HV Hotplug		Poll
		add	rmv		add	rmv	
CPU	req	nCPUs numa_node allow_ht	cpu_id	*CPU code*	nCPUs hv_numa_node allow_ht	hv_cpu_id	*CPU code*
	res	cpu_ids[] is_ht[]	success/ fail	*code* nCPUs numa_node	hv_cpu_ids[] is_ht[]	*code*	*code* nCPUs numa_node
Memory	req	size numa_node	addr size	*MEM code*	size hv_numa_node	memdev[]	*MEM code*
	res	addr	success/ fail	*code* addr / node size	size memdev[]	*code*	*code* size numa_node

Figure 2: Communication protocols

plemented as a character device inside the guest OS. This device is triggered through `ioctl` calls that are handled by the Guest Hotplug Driver. The latter communicates with the Host Hotplug Driver leveraging `virtio` [32] which enables asynchronous host–guest communication through a set of ring buffers. Finally, the Host Hotplug Driver triggers the RM according to the request. Accordingly, the final decision on the resources requested by a VM is performed individually by every application, based on the expected benefit of the resources to the application performance.

Communication between the guest and the host supports three operations: (i) dynamically add (*hotplug*) CPUs/ memory, (ii) dynamically remove (*hotunplug*) CPUs/memory, and, (iii) poll for changes in the availability of CPUs/memory. As shown on the upper side of Figure 2, the hotplug device acts as a proxy between applications running inside the VM userspace and the hypervisor. Depending on the type of resource, the protocol is given on the table of Figure 2.

The CPU hotplug request includes the number of CPUs, the associated NUMA node, and a flag indicating whether HyperThreads are allowed. The hotplug device forwards this request to the hypervisor which replies back with a list of CPUs added to the VM and a HyperThread indicator for each one of them. The application receives a response with the list of the CPU ids as they have been associated by the guest OS. This list does not necessarily contain the same ids as the one received by the hypervisor, since the guest OS may use different numbering than the hypervisor. The same holds for the NUMA node as well and for this reason, they are both indicated with the `hv_` prefix in the communication protocol with the hypervisor. The hotunplug request issued by an application includes the id of the CPU to be removed from the VM. The hotplug device forwards the request to the hypervisor which replies with a code indicating success or reason of failure and the application is notified accordingly. The poll request issued typically by the Host Monitor indicates that it expects updates on the availability of CPUs. The hypervisor replies to that request with a code indicating whether the VM should expand or shrink and if the given CPUs correspond to HyperThreads, the number of CPUs, and the NUMA node that they are associated with. The application receives this response from the hypervisor and decides whether to perform a request to hotplug or to hotunplug the CPUs listed in the response.

Similarly, the memory hotplug request includes the size and the desired NUMA node. The Guest Hotplug Driver forwards this request to the hypervisor which replies back with a list of the memory devices added to the guest OS and their total size. The application in turn receives a virtual memory address that it can use. To hotunplug memory, the applications send the address that the memory to be removed starts and its size. The guest OS resolves the address and finds the corresponding memory devices which are passed to its request to the hypervisor. The hypervisor returns a code indicating whether the memory devices have been removed and the application is notified by the guest OS with a success or failure notification. Finally, the poll request from the userspace indicates that it expects updates on the memory availability. This request is forwarded to the hypervisor by the guest OS. The response of the hypervisor includes a code indicating whether memory can be added or it should be removed, its size and the associated NUMA node. If this is a request to remove memory, the guest OS replies with the memory address and its size that the application should remove. Otherwise, if this is a request to add memory, the guest OS replies with the NUMA node that the memory is available and its size, so that applications can perform the corresponding request.

3.2 CPU and Memory Hotplug

The natural incentive for a VM is to expand as much as possible. On the other hand, the cloud provider typically gains more from hosting several tenants instead of providing more resources to a single one. Therefore, an elastic scale-up system should ensure that the resources of a VM cannot go either above or below a given threshold. Moreover, the cloud provider needs to be able to force the VM to shrink, especially when the VM is a spot instance, and the hotplug infrastructure needs to enforce shrinking. Finally, as it has been well-investigated, the memory topology of a multi-socket, multi-core server has a significant impact on the performance of the OLTP engine [25, 29]. Hence, it should be left on the OLTP engine to decide the NUMA node that hotplugged CPUs and memory should belong to.

In our system, the Guest Hotplug Driver keeps all information on the available CPUs and memory. For each CPU, this information includes its id, NUMA node, and if it is a HyperThread. For each memory block, this information includes its id, size and NUMA node. We rely on the native OS device data structures to represent hotplugged CPUs and memory blocks and we extend them with further information on their mapping with the applications and the hypervisor. The CPUs and the memory blocks that have been initially allocated to the VM are kept in the *startup set* and the ones that have been hotplugged are kept in the *hotplug set*. This separation is used by the hotplug device to block any requests to remove resources that have not been hotplugged. The startup set is determined by the driver at boot time and remains constant throughout the VM lifetime.

3.2.1 CPU Hotplug

When an application sends a CPU hotplug request to the hotplug device, the Guest Hotplug Driver forwards it to the hypervisor. Upon a successful hotplug, the hypervisor returns the CPU ids and their NUMA node. The Guest Hotplug Driver then makes a call to the CPU hotplug infrastructure of the Linux kernel and boots the added CPUs. Finally,

for each CPU, it adds their guest id in the hotplug set and associates it with the id sent by the hypervisor. When an application sends a hotunplug request for a specific CPU id, the Guest Hotplug Driver first makes sure that the CPU belongs to the hotplug set. In this case, it first requests the Linux kernel to shut down the CPU, then it retrieves the associated CPU id that was sent by the hypervisor and sends the hotunplug request to the hypervisor with this id.

On the host, similar information is kept by the Resource Manager (RM) for each VM, which further keeps a pool of CPUs that can be hotplugged to a VM. We refer to this pool of CPUs as the *available hotplug set*. CPUs in the available hotplug set may be shared across VMs, in case they are spot instances, or they may be statically associated to a specific VM. Therefore, the availability of these CPUs depends on the SLAs between the users and the cloud providers and the RM implements policies to comply with these SLAs.

When the Host Hotplug Driver receives a request to hotplug CPUs of a NUMA node, it retrieves from the RM a set of host CPUs that are available to the requesting VM and match its requirements. Then, it issues a request to the QEMU hotplug infrastructure to add CPUs of the requested NUMA node to the VM. QEMU starts a new thread for each CPU and assigns it to the VM, enabling the latter to pass instructions to the CPUs. Then the Host Hotplug Driver affinitizes the newly created threads to the hotplugged CPUs and adds them to a shielded set dedicated to that VM so that the OS does not schedule any threads on them in the future. Finally, the Host Hotplug Driver generates a random id for each hotplugged CPU, maps it to the physical CPU id, and passes it to the VM. In case of failure the corresponding error code is returned with an empty set of CPUs. Failures can happen due to lack of available CPUs for that VM on the NUMA node it requested, an SLA violation, or a hardware failure of the requested CPU.

Accordingly, when the Host Hotplug Driver receives a request from a VM to hotunplug a set of specific CPUs, it first retrieves their host ids and then requests QEMU to use its hotplug infrastructure to remove them from the VM. Finally, the Host Hotplug Driver removes the CPUs from the shield and notifies the RM that they are available again. The RM, then, places them in the available hotplug set again. However, requests to hotunplug CPUs may also come from the RM, in order to enforce SLA conformance. The most typical case is that the VM is a spot instance and it has to remove some of its CPUs so that they are given to another VM. Even though the VM will be notified through the Host Monitor that it has to remove a set of CPUs, the application using them may either not be willing to release them, or simply crash and not issue any requests to hotunplug resources. For this reason, the RM issues a timeout to each hotunplug request to the VMs. In case the VMs have not issued an explicit request for hotunplug until that timeout has passed, the RM issues a request to the Host Hotplug Driver to remove the corresponding CPUs and they are force removed from the VM as described above.

3.2.2 Memory Hotplug

Memory hotplug is performed similar to CPU hotplug and therefore the workflow is the same as the one described for the CPU in terms of adding and removing memory banks to/from the VM. However, there is a difference in the way that memory becomes available to the applications. Specifically, applications allocating memory with `malloc`, or related calls, are not aware whether they are using hotplugged memory. Then, in case memory has to be hotunplugged, the application cannot know which data it keeps on each memory block and will have to ask the OS for the memory addresses. In the worst case, the OS will have to page walk the whole memory allocated by the application to find the corresponding memory blocks to remove. We solve this problem by exposing through the Virtual Hotplug Device an `mmap` handler to the applications which we implement in the Guest Hotplug Driver. This way, applications can allocate memory only through explicit calls to the driver, enabling the former to keep track of the data that they keep on each part of the memory and the latter to keep track of the usage of hotplugged memory across applications.

Despite its usefulness, memory elasticity also brings overheads, as every call to `mmap` requires to update the page tables with the newly allocated memory, similar to [33]. To alleviate this overhead, we follow a lazy approach, where we process only the amount of hotplugged memory which is requested every time by the application, by following the page faults. The granularity of hotplugged page map can be adjusted based on the workload and the time required to fill every page. Instead, when hotplugged memory arrives to the VM, the Linux kernel registers the page frame numbers and keeps an index separating the hotplugged from the static ones. This way, the application can easily handle hotplugged memory as a dynamic memory pool.

3.3 Resource Management and Monitoring

Resource management is associated with the SLA governing the use of cloud resources from the VMs. Each SLA is implemented as a policy in the Resource Manager which decides the resource distribution to the VMs. To make these decisions, the RM abstracts the host hardware topology and maintains a data structure with information such as the NUMA nodes, the amount of memory that each node has, and the CPUs associated with it. For each CPU, the RM maintains its physical id and an indicator pointing out whether that CPU is a HyperThread (i.e., whether there is another CPU on the same core). Accordingly, for each memory block, the RM keeps an id, its starting address and its size. Finally, it distributes CPUs and memory to the startup, hotplug, and available hotplug sets for each VM.

We have implemented two policies: *fair*, and *strict resource allocation*. The fair policy allows each VM to takes an equal share of the resources in the available hotplug set. Conversely, the strict policy restricts VMs from sharing any resources. Since the RM receives all the hotplug requests from the VMs, it can monitor the resource allocation of the host across the VMs and decide whether a VM can add or should remove resources. However, this information cannot be exposed to the VMs, whereas they still need to receive notifications from the host regarding changes in the resource availability. For this reason, the Host Monitor in the guest polls the hypervisor periodically for changes in the availability of resources for the VM. This information is kept locally by the Host Monitor, which is also polled by the applications to find out whether they need to take any actions.

In general, applications running in the VM will directly execute hotunplug requests coming from the hypervisor, since the latter will anyway remove the resources after a timeout. However, they will not always hotplug all the available re-

sources of the host. As explained in Section 4, there are cases where the VM prefers to drop CPUs, for instance due to high contention in the workload. Moreover, using memory from a remote NUMA node can have negative effects in the performance. To make such decisions, the application needs to monitor its performance, workload and CPU utilization. For this reason, we have introduced the VM Monitor, which keeps track of these metrics and an API to store and retrieve information from it. This way, when an application receives an offer from the Host Monitor to hotplug CPUs, it checks with the telemetry data of the VM Monitor to decide whether it will issue a hotplug request.

3.4 Elastic OLTP Engine

The infrastructure described so far enables VMs to add or remove CPUs and memory dynamically, as well as to monitor the resource usage from the host and the guest sides. To exploit elasticity, an OLTP engine has to dynamically adjust its thread and memory pool to the changes in the availability of resources. In this work, we use Trireme, an open source OLTP engine that has been used in prior research to perform a side-by-side comparison of various OLTP engine architectures [8]. We configure Trireme as a shared-everything, in-memory OLTP engine and we integrate it with our hotplug infrastructure to make it elastic. In the following, we first present the system design of Trireme, and then we describe the steps required to exploit CPU and memory elasticity.

Trireme consists of three components, namely the Storage Manager (SM), the Transaction Manager (TxM), and the Worker Manager (WM). The SM provides in- memory storage of database records. The TxM works together with the SM to provide transactional access to records. Accordingly, the TxM maintains data structures necessary for implementing various concurrency control protocols. In this work, we configured TxM to use two-phase locking (2PL) with deadlock avoidance. WM is responsible for managing the thread pool. Trireme runs as a multi-threaded process with one worker thread assigned to each available CPU. The WM in unmodified Trireme creates the thread pool at system initialization time and affinitizes each worker thread to one CPU, whereas it also keeps the id of the NUMA node that the CPU belongs to. Once started, each worker thread repeatedly executes transactions one after another.

CPU elasticity in Trireme is managed by the WM. Each worker thread periodically reports to the VM Monitor its status, including transactional throughput, abort rate, latency, and access frequency. Moreover, the WM periodically polls the Host Monitor for changes in the availability of CPUs. In case the Host Monitor requests to remove a set of CPUs from a specific NUMA node, the WM iterates over the worker threads and signals the ones that should be removed to stop transaction execution. After the worker threads have stopped executing transactions, they report their final statistics to the VM Monitor and the WM removes them from the thread pool. Instead, if the Host Monitor suggests to add CPUs, the WM checks the statistics from the VM Monitor to decide whether more computing resources will be beneficial. In this case it makes a call to the hotplug device to add the CPUs, and when they become available, the WM starts new worker threads, affinitizes them to the corresponding hotplugged CPUs and starts transaction execution on them.

Similarly, memory elasticity in Trireme is driven by the SM which also reports its status to the VM Monitor and receives notifications from the Host Monitor about changes in the availability of memory resources. In case memory has to be removed, the SM first tries to release memory which is either unused or used for maintenance tasks, like index maintenance. If this is not enough, it releases memory where it keeps database records. In this case, it first flushes the memory to the disk, and then releases it, following an anti-caching approach [18]. On the other hand, when memory is available, it checks its locality and uses it either to restore data from the disk or for maintenance tasks.

Generalization. Even though our system design is driven by OLTP engines, any stateful cloud application with similar properties can also be integrated. Specifically, the application needs to have an elastic worker and memory pool which can enforce the hypervisor decisions within the specified time bounds. Further, by keeping the Host Monitor as a separate component in the userspace, any application inside the VM can communicate with the hypervisor to elastically manage its resources. Therefore, our system design can be used to pack several applications within the same server, by relying on VMs for isolation across tenants. Accordingly, the cloud provider can schedule multiple VMs on the same server to maximize resource utilization and the tenants can take up, and pay for, resources only when needed.

4. USAGE SCENARIOS AND EVALUATION

In this section we evaluate the performance of our system considering different scenarios that outline the benefits of scale-up elasticity for OLTP. We first present a proof of concept and four different scenarios focusing on CPU hotplug. Then, we demonstrate the case of memory scaling, which can apply to a number of different scenarios, which we also give. Even though CPU and memory elasticity can clearly be mixed, we have split them in order to isolate their effect on the performance. We present each scenario as follows: first we give the motivation, then the experimental setup, then the execution timeline, and finally, the control flow that maps all the operations to our design. In the first, proof of concept, scenario we also compare our elastic scale-up approach with two alternative ones, namely multi-tenant databases and scale-out with distributed transactions.

We conducted our experiments on three servers with different hardware configurations to demonstrate the scalability of our infrastructure. More specifically, we used: (i) a server equipped with 2x8-core Intel Xeon E5-2640 v2 processors (32-KB L1I + 32-KB L1D cache, 256-KB L2 cache, and 20-MB LLC) clocked at 2.0-GHz with HyperThreads enabled and 256-GB of DDR3 DRAM, (ii) a server equipped with 4x18-core Intel Xeon E7-8890 v3 processors (32-KB L1I + 32-KB L1D cache, 256-KB L2 cache, and 45-MB LLC) clocked at 2.5-GHz with HyperThreads enabled, and 512-GB of DDR4 DRAM, and, (iii) a server with 8x10-core Intel Xeon E7-L8867 processors (32-KB L1I + 32-KB L1D cache, 256-KB L2 cache, and 30-MB LLC) clocked at 2.13-GHz with HyperThreads enabled and 192-GB of DDR3 DRAM.

The benchmark that we used for our experiments is the Yahoo! Cloud Serving Benchmark (YCSB) [13]. We used YCSB because it has been widely used for the evaluation of both OLTP engines (scale-up and scale-out) and other cloud applications, which can be benefited from our system [14, 17, 35, 39, 36, 42, 43]. Each YCSB record has a single primary

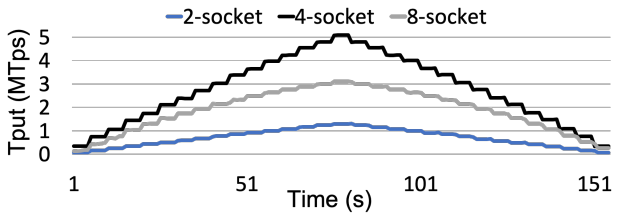

Figure 3: Trireme throughput as VM size changes

key and ten string fields with 100 characters length. Each transaction either reads or updates 10 records. Unless explicitly stated otherwise, we use a 72GB database containing a single table with 72M records, and we set the update rate of our workload to 20%. All record accesses follow the Zipfian distribution with parameter theta 0.5, except a single scenario where we vary theta to switch between low and high contention. All the experiments are performed in timed executions and the metrics reported are in one-second intervals, similarly to existing work [39]. As for all our servers there is a one-to-one mapping between CPU sockets and NUMA nodes, we will use these terms interchangeably.

Every scenario is supported either by the fair or the strict resource allocation policy. We use the former to uniformly distribute the resources across all the active VMs on a server, and the latter to restrict the resource distribution within certain bounds for every VM. These policies represent the two extreme cases, allowing several combinations in-between. Accordingly, we demonstrate that our system can efficiently adapt in the extreme cases, whereas we leave more complex policies for future work.

4.1 Proof of Concept

Motivation. A cloud-hosted OLTP engine should be able to dynamically scale-up and down based on the availability of resources on top of different hardware configurations. Accordingly, our goals in this scenario are: (i) to show how the components of our system interact in order to expand or shrink the VM CPUs based on triggers generated either by the host or the guest side, and (ii) to show that Trireme can effectively scale over different hardware configurations and at different hotplug granularities.

Experimental setup. A VM starts with a small startup set of CPUs and the rest of the CPUs available in its hotplug set. Then, we periodically add a part of the hotplug set to the VM until all the physical cores of the host are used. Finally, we follow the reverse the procedure, until the VM has remained with the startup set of CPUs.

To demonstrate the scalability of the platform under different CPU hotplug granularities, we hotplug a different number of CPUs every 5 seconds on every server. For the 2-socket server, we start with 1 CPU available and we hotplug 1 CPU every 5 seconds. For the 4-socket server, we start with 4 CPUs available and we hotplug 4 or 5 CPUs every 5 seconds alternating the number at each iteration. For the 8-socket server, we start with 5 CPUs available and we hotplug 5 CPUs every 5 seconds. For all servers, after all the CPUs have been added, we follow the inverse procedure.

Timeline. Figure 3 shows the throughput of Trireme in million transactions per second (MTps), reported every 1 second, throughout the execution of this scenario. Each line in the figure corresponds to a different hardware configuration, namely the 2-, the 4-, and the 8-socket server. Note that the servers are equipped with different generations of x86 processors and with different core counts. More specifically, the 8-socket server has Intel Westmere CPUs, which are older that 2-socket server's Ivy Bridge CPUs, which, in turn, are older than the 4-socket server's Haswell CPUs. Thus, our goal here is not to provide an apples-to-apples comparison of throughput across servers. Instead, our goal is to show that there are no bottlenecks in our elastic scaling framework that limits scalability to a few CPU sockets.

As can be seen in the figure, after the experiment starts, Trireme increases its throughput periodically, every time a set of CPUs is hotplugged to the VM. This continues until all the physical cores of the host have been added to the VM, at 80 seconds. From that point on, performance starts decreasing in a symmetric way, and when all the CPUs belonging to the hotplug set of the VM have been removed, then it becomes the same as it was at the beginning of the experiment. This behaviour is reflected in all the three different configurations used for this experiment.

We have also measured the latency for dynamically adding and removing sets of CPUs on all the three servers. The average latency for dynamically adding CPUs to the VM is within the range of 20ms–30ms for all servers, whereas the average latency for dynamically removing CPUs from the VM is within the range of 30ms–50ms for all servers. Since hotplug operations do not take place very frequently during the workload execution of the OLTP engine, these latencies do not impose any overhead to transaction execution. We study this in more depth in Section 4.5.

Control flow. To implement this scenario, we injected specific policies in Trireme WM and in the hypervisor RM. For simplicity, we describe these policies with the parameters used for the 2-socket server. There, RM allows a VM to take 1 CPU of the host after every hotplug request. If there are no CPUs left in the available hotplug set, the policy requires that the VM returns back 1 CPU. The Host Monitor polls the hypervisor every second, whereas Trireme WM polls the Host Monitor every 5 seconds.

Following this set of policies, when the experiment starts, the Host Monitor waits for 1 second and then polls the hypervisor. The hypervisor replies back that there is 1 CPU available and the NUMA node of this CPU. After 5 seconds that the WM of Trireme polls the Host Monitor, it sees that there is 1 CPU available, and performs a hotplug request for that CPU. The request is granted by the Resource Manager, the CPU is added to the VM and the WM starts a new worker thread on that CPU. After the request to add the last CPU available in the host, the Resource Manager changes its response to the Host Monitor to a request to remove 1 CPU from a given NUMA node. The WM takes this value, finds a worker thread running on a CPU of that NUMA node, stops it and hotunplugs the CPU. In the meanwhile, each worker thread reports its throughput every 1 second to the VM Monitor, and we report the aggregate in Figure 3.

Comparison with multi-tenant databases. An alternative to our design is SQLVM, a cloud-based design for SQL server. Instead of deploying databases inside VMs, SQLVM provides performance isolation within the database. This provides flexible resource allocation for the database, but lacks the generality of our approach which (i) supports any application with an elastic worker and memory pool, and,

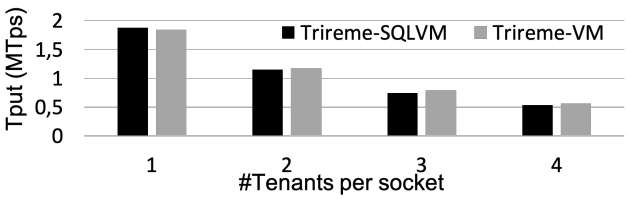

Figure 4: Performance of SQLVM and Trireme

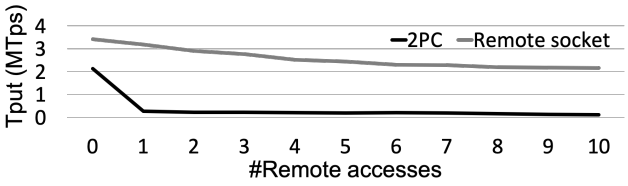

Figure 5: Trireme performance with remote accesses

(ii) can host multiple applications within a single server. To investigate possible overheads of our design compared to SQLVM, we have implemented a SQLVM emulator inside Trireme. When Trierme runs in SQLVM mode, it starts as single multi-threaded process and assigns a set of threads and a data partition to each tenant, while transaction execution for each tenant is limited in its own partition.

We have executed an experiment on our 4-socket server, where we vary the number of tenants per socket. We use YCSB, but in order to have a fair comparison, the size of the database is 1 GB per worker thread. We consider a read-only workload and we set the theta parameter of the Zipfian distribution to 0.5, to avoid any overheads coming from the workload. When we use 1 tenant per socket, in the SQLVM case we use one process with 72 threads and a 72 GB database split across 4 tenants, with each tenant taking the 18 CPUs of each socket. For Trireme-VM, we deploy 4 VMs, one on each socket. Inside each VM we deploy a Trireme instance with 18 worker threads performing transactions on an 18 GB database. Similarly, we run the same experiment for 2, 3, and 4 tenants per socket. As shown in Figure 4, as we increase the number of tenants per socket from 1 to 4, the performance of Trireme remains the same regardless the configuration. This shows that the overheads of using a general purpose software infrastructure do not produce any significant interference with the workload execution.

Comparison with scale-out databases. Traditionally, cloud-hosted applications were designed to scale-out to exploit the availability of servers in data centres. One approach to elastically scale-out is data repartitioning, which works best with partitionable workloads, otherwise it has overheads for moving data across instances over the network. An alternative is to use a distributed transaction coordination protocol like two-phase commit (2PC). 2PC has two phases: first, the thread that issued the transaction (coordinator) sends a prepare message to all the participants; then, each participant sends back either a `commit` or an `abort` message; if all participants sent `commit`, then the transaction commits, otherwise it aborts and the coordinator notifies the participants accordingly.

To investigate the impact of 2PC on main-memory OLTP performance, we have implemented 2PC in Trireme, and we have configured it to work as a scale-out engine. We have executed an experiment using YCSB with a 72 GB database where each transaction executes 10 read-only operations under low contention (theta=0.5). In this experiment, we used two nodes, each one running on a different socket of our 4-socket server and using all their 18 cores. Figure 5 shows Trireme performance as we increase the number of remote operations. As shown in the figure, a single remote operation causes a performance drop around 8x, whereas after that point performance plateaus. Therefore, the communication latency imposed by 2PC, both due to the network communication and the protocol complexity, cause significant performance issues even with just a single remote access. In the same figure, we also show the impact of remote memory accesses on the performance of the system, when every worker thread access a record located on a different NUMA node. In this case, Trireme uses two sockets and 36 worker threads and we observe that the effect of NUMA is much less than the overheads of 2PC. Thus, in the presence of enough resources on a single server, scaling up is simpler and more beneficial than scaling out.

Comment on comparison. This work does not intend to replace multi-tenant or scale-out approaches in all cases. Instead, it focuses on the cloud migration of in-memory OLTP engines to large-scale servers, provided that they have enough resources to satisfy the workload requirements. Accordingly, it relies on existing virtualization frameworks for resource isolation, whereas its design is non-intrusive to the OLTP engine. In the following, we focus on use cases that outline the benefits of scale-up elasticity in OLTP systems.

Having demonstrated the scalability of our infrastructure on all our servers, for the remainder of this paper, we only present results obtained from the 4-socket server.

4.2 Elasticity and Dynamic Scaling

Motivation. Cloud providers offer spot instances which are VMs billed by the second at a cheaper price than reserved instances with the agreement that the provider can withdraw their resources any moment. Depending on the demand, cloud providers move resources between spot and reserved instances. Spot instances improve the ratio of billed resources while providing cheaper prices to customers.

However, the current spot instance model is very strict since it means an all-or-nothing condition for customers as spot instances can be terminated without notice. In many cases, particularly when the VM is running stateful services like an OLTP engine, an all-or-nothing compromise is not a suitable option. Thus, today, cheaper VMs provided by spot instances are only used for non-critical systems and tasks without a hard deadline or well-defined SLA requirements.

In our use case, spot instances are VMs that are started with minimum and ideal computational requirement specifications. Under low-demand, where CPUs are available, they can be allotted to spot instances at a cheaper rate. However, as more tenants use the server, CPUs are taken away from spot instances shrinking them back to the specified minimum and allotted to other tenants. By using such spot instances, customers can make flexible cost-performance tradeoffs. They can also schedule low priority services, like index maintenance and data reorganization, to be performed only when resources are available at spot prices.

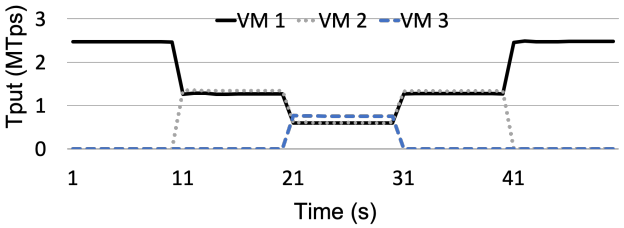

Figure 6: Trireme throughput while spot scaling

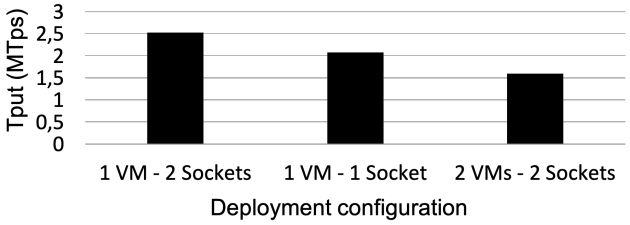

Figure 7: VM/Socket deployment combinations

As resources are added to and removed from the VMs, the Resource Manager is expected to allocate resources which are local to the data used by each VM. This is achieved by using the protocol semantics of our infrastructure, where the RM suggests available CPUs from specific NUMA nodes. Then, applications can implement NUMA-aware policies by issuing requests for specific NUMA nodes [30].

Experimental setup. In this experiment we show that our system adapts to the changes in the CPU availability when an OLTP engine is deployed inside a spot instance. We consider 3 VMs that will share the CPUs of two sockets of our server. Thus, each VM starts with a startup set of 1 CPU and a hotplug set of 35 CPUs. First, a VM starts executing transactions and expands to all the CPUs of the two sockets. Then, after 10 seconds, the second VM starts and gets its share of the CPUs. Finally, after 10 more seconds, the third VM joins and also starts using the CPUs of these sockets. After it has executed transactions for 10 seconds, the third VM stops and releases its CPUs. Then, after 10 seconds, the second VM also stops and releases its CPUs.

Timeline. Figure 6 shows Trireme performance, obtained by the VM Monitor of each VM throughout the experiment. As shown in the figure, the throughput of VM 1 remains high for the first 10 seconds, and then it drops as VM 2 starts. At that point VM 1 and VM 2 have almost the same throughput, since they have use the same number of CPUs. After 10 seconds, VM 3 also starts, receives cores from the hypervisor, and starts executing transactions. The throughput of VM 1 and VM 2 drops accordingly and matches the throughput of VM 3. After VM 3 has finished executing transactions, it releases its cores and the throughput of VM 1 and VM 2 returns back to the level as it was before VM 3 had started. Following this VM 2 finishes execution and terminates, returning its CPUs back to VM 1.

Control flow. In this experiment, the Resource Manager employs the fair resource allocation policy which distributes CPUs uniformly across the VMs. When VM 1 starts, its Host Monitor polls the hypervisor and receives a reply that it can use all the cores of the two sockets since it is the only active VM. Accordingly, Trireme WM requests to hotplug all CPUs of both NUMA nodes that are available. After the CPUs have been added to the VM, Trireme uses them to execute transactions. In the meanwhile, the Host Monitor polls the hypervisor for changes in the availability of resources every 100 milliseconds. When VM 2 starts, the hypervisor asks VM 1 to remove 18 CPUs and gives them to VM 2. Following these instructions, the WM of Trireme running in VM 1 issues a request to hotunplug 18 CPUs, whereas the WM of Trireme running in VM 2 issues a request to hotplug these CPUs. Similarly, when VM 3 starts, the Resource Manager notifies VM 1 and VM 2 to release 6 CPUs each, and it gives them to VM 3. Finally, when each VM finishes executing transactions, Trireme WM hotunplugs its CPUs and the Resource Manager distributes them to the remaining VMs.

4.3 Elasticity and Performance Isolation

Motivation. Cloud providers often need to ensure that tenants applications do not interfere. Figure 7 depicts the throughput of Trireme in three different multi-tenancy deployment setups. In the first setup, a single instance is deployed in a VM that is allocated 36 threads across two sockets, using 18 cores per socket. In the second, the single tenant VM is allocated 36 threads from a single socket, thus using 18 cores and 18 HyperThreads. In the third, we run two VMs representing different tenants, each hosting an independent instance and we report the average throughput of the two VMs. Therefore, in this configuration, contrary to what was presented in the previous section, all the 72 hardware threads that are available on each socket are being used. There are two important observations to be made in this experiment. First, in the single tenant case (first two scenarios), we see that thread placement plays an important role as the setup with two sockets outperforms its single-socket counterpart. This is expected given that using two sockets instead of one provides a much larger cumulative cache space, whereas using cores instead of HyperThreads results in less contention at the microarchitectural level. Second, in the two-tenant case (third scenario), we see that the throughput drops further 20% due to resource sharing across tenants.

Therefore, resource allocation should be dynamically decided based on the degree of multi-tenancy. An elastic OLTP infrastructure should spread out the threads of an OLTP engine across multiple sockets when resources are available to ensure peak performance. However, in the presence of multiple tenants, the infrastructure should allocate threads to VMs in such a way that VMs do not share sockets. In our system, this is achieved by the RM which can redistribute CPUs to keep the VMs within the socket boundary. Even though this is relatively cheap for compute resources, moving memory between nodes can be costly, and thus, it may be better to share the memory bandwidth between the VMs instead of full memory-level isolation.

Experimental setup. In this scenario, we consider that there is one spot VM already executing transactions using 36 cores across two sockets available on our server. Then, we start a second VM that also requires 36 cores and we isolate transaction execution in a single socket for both of them. In the following, we show the benefit of using elasticity to isolate VMs execution to the socket boundary.

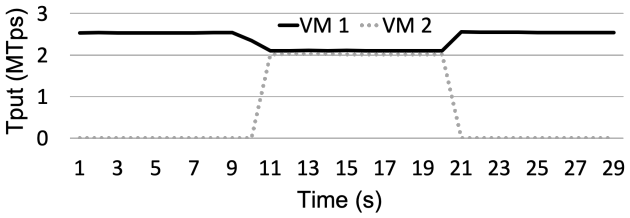

Figure 8: Performance isolation within socket

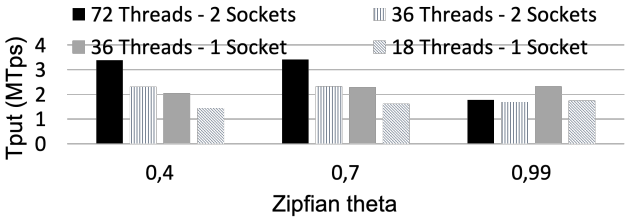

Figure 9: Effect of contention on performance

Timeline. Figure 8 shows the throughput of each VM when we isolate execution to the socket boundary. As shown in the figure, after VM 1 executes transactions for ten seconds, VM 2 starts. Then, VM 1 releases all the CPUs from the second socket and uses HyperThreads on the first socket, whereas VM 2 does exactly the same on the second socket. As a result, the performance of VM 1 drops by 20%, whereas the performance of both of them is better by about 35% compared to the non-isolation case, as shown in Figure 7.

Control Flow. In this experiment, the Resource Manager employs the performance isolation policy. When the Host Monitor of VM 2 polls the hypervisor, then the Resource Manager requests VM 1 to remove all of the CPUs of the second socket and suggests to add the HyperThreads of the first one. Likewise, it suggests VM 2 to add all the physical cores and HyperThreads of the second socket. Similarly to the previous experiments, Trireme adapts its execution accordingly. When VM 2 finishes its execution, then the Resource Manager suggests VM 1 to add CPUs of the second socket. The WM of Trireme in this case hotplugs all the CPUs corresponding to the second socket, moves the affinity of the worker threads to these CPUs and then hotunplugs the CPUs of the first socket that remained idle.

4.4 Elasticity and Contention Scaling

Motivation. The scalability of OLTP engines depends on the degree of workload contention. For low contention workloads, modern in-memory OLTP engines can provide near-linear scalability. However, when the contention is high due to skewed data access, OLTP engines suffer because of concurrent transactions performing conflicting operations that cannot be serialized. Thus, beyond a degree of contention in the workload, the addition of more resources can have the adverse effect of reducing the system throughput.

Figure 9 exemplifies this behaviour showing the performance of Trireme when we increase the level of contention in the workload by increasing the theta parameter of the Zipfian distribution. The four bars correspond to four scenarios. In the first scenario, the VM has 72 threads running

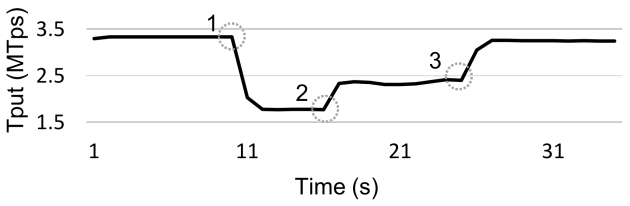

Figure 10: Throughput as contention changes

across two sockets using 36 hardware threads per sockets (physical cores and HyperThreads). In the second, the VM has 36 threads from two sockets by using only physical cores. In the third, the VM has 36 threads in one socket by using HyperThreads. In the fourth, the VM has only the 18 physical cores of a single socket. As shown in the figure, when the contention is low, it is better to use all the hardware threads available in both sockets, whereas when the contention is high, it is better to limit within a single socket.

Therefore, an elastic scale-up OLTP engine should monitor the degree of contention in the workload to decide whether it should proactively release some CPUs and limit itself to the socket boundary to improve performance. Doing so results in three benefits. First, by releasing CPUs, a VM experiencing contention can optimize for cost rather than performance, due to the lack of workload scalability. Second, releasing CPUs will improve throughput due to a proportionate reduction in the number of conflicting transactions. Third, the released CPUs can be used by other VMs or simply put in a low power state to reduce power consumption, thus benefiting other tenants and the cloud provider.

Note that, releasing CPUs of a socket which contains data, might lead to remote data accesses. However, as shown in Figure 5, the impact of remote operations in a transactional workload is much less than the impact of cross-socket atomics in high-contention, as transaction operations do not generally access big amounts of data.

Experimental setup. We consider a VM running on two sockets using all the hardware threads. Initially, the VM executes a low contention workload that we have used throughout this section. At some point, contention changes to high with theta becoming 0.99, and then it changes back to 0.5 as it was initially. In the following, we demonstrate that Trireme adapts its CPU allocation in order to achieve better throughput and resource utilization.

Timeline. Figure 10 shows the performance of Trireme when the contention in the workload changes. In the beginning the workload has low contention (theta 0.5). After 10 seconds, theta increases to 0.99 and throughput drops, as shown in the figure (1). Then, the WM of Trireme removes all the CPUs of the second socket and executes the workload with half its worker threads. This isolates contention within the socket boundary (2) and provides an increase in throughput. At time 25, contention drops as theta is reduced back to 0.5. The WM of Trireme then hotplugs all the CPUs of the second socket and the throughput becomes the same as it was in the beginning (3).

Control Flow. In this experiment, the Resource Manager uses the strict resource allocation policy, where the VM controls fully a fixed set of resources. The WM of Trireme polls the VM Monitor every second. When the VM Moni-

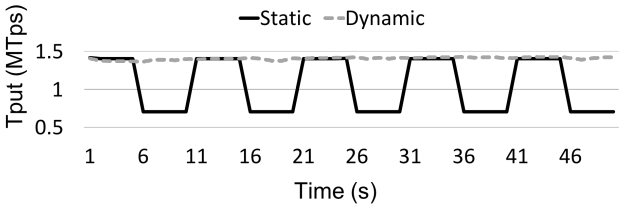

Figure 11: Throughput as the traffic levels change

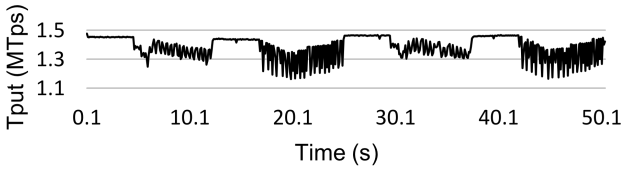

Figure 12: Throughput during CPU hotplug

tor captures the change in the contention, the WM adapts its resource allocation by hotunplugging all the CPUs from the second socket. Similarly, when the contention becomes again low, the VM Monitor captures it and the WM expands again its worker pool with all the CPUs of the second socket.

4.5 Elasticity and Peak Traffic Scaling

Motivation. One of the main incentives to move from on-premise to cloud deployments is to address peak traffic demands that take place for short time periods and require fast adaptation. In this execution model, an OLTP engine runs normally with some pre-allocated CPUs which remain fixed throughout the lifetime of the VM. However, when there is a peak in traffic, for instance due to an increase to the users' interests, then the OLTP engine needs to expand its worker pool to meet its performance requirements. In this case, the cloud provider can take CPUs from spot instances running on the same server and give them to the VM where the OLTP engine runs. This model has benefits both for customers and cloud providers. The former gain from not overprovisioning resources to their VMs, and the latter by allocating the unused resources to spot instances, thereby increasing the number of users in their infrastructure.

Experimental setup. In this experiment, we consider that there is a VM with Trireme executing transactions on one socket, using 18 CPUs. We split execution in two periods, of 5 seconds each. In the first period, the traffic is normal, whereas in the second period, the traffic doubles. Since each worker thread of Trireme first generates a transaction and then executes it, to emulate the traffic doubling we set it to execute transactions for 500 ms and then sleep for another 500 ms, while we report throughput every second. In the static allocation case, we keep the number of threads fixed to 18 to show the impact on the performance of the system. In the dynamic allocation case, we hotplug one CPU from the second socket every time a worker thread goes to sleep and, then, we hotunplug it when the worker thread wakes up again. Then, to give a microscopic view of the system, we take shorter time intervals, where each period lasts for 500 ms, threads switch every 50 ms, and we report throughput every 100 ms.

Timeline. Figure 11 shows the performance of Trireme. During the first 5 seconds, both the static and the dynamic approach have the same performance, since they run with the same 18 CPUs. After 5 seconds that the traffic doubles, the performance of the static approach cuts in half, as expected, whereas the performance of the dynamic approach remains the same. The reason is that at that point, Trireme hotplugs the 18 CPUs of the second socket to execute transactions. This behaviour repeats consistently over time showing the stability of the system.

Figure 12 shows the microscopic view of Trireme performance. After the first 500 ms, we start putting worker threads to sleep and hotplugging the CPUs of the second socket. As we see the performance has very small variations during that period, whereas it stabilizes right away. After running with 36 worker threads for another 500 ms, the reverse process is followed and for each CPU that is hotunplugged, a worker thread wakes up. Again, after some small variations, performance becomes again stable. We also show a second round of the same procedure to show the system stability. Accordingly, we observe that Trireme performance is negligibly affected during elastic scale-up.

Control flow. In this experiment, the Resource Manager employs the static resource allocation policy. The Host Monitor polls the hypervisor every 100 ms, and therefore, it is aware that the CPUs of the second socket are available when it does not use them. The WM of Trireme does not use them until it realizes that there are transactions waiting to be executed. At that moment, it hotplugs the CPUs and expands its worker pool. Conversely, when the transactions stop waiting, the WM of Trireme shrinks its worker pool back to its original size, thus enabling the Resource Manager to use the unplugged CPUs for another (spot) VM.

4.6 Elasticity and Memory Scaling

Motivation. In all the above scenarios, memory can also be considered in parallel with CPU elasticity, since thread and memory placement can affect the performance of the applications running in a single multi-socket server [25] [12].

A typical scenario showing the benefits of memory elasticity is when traffic demands increase. Then, memory elasticity increases the available memory size while the traffic is high, and then shrinks it back when traffic becomes normal again. Similarly, memory elasticity is beneficial during periodic database operations, like index maintenance, where the system needs additional memory for some time in order to build the new index that will replace the old one. As in both cases the memory is released when it is no longer needed, it comes with almost zero overhead to the OLTP engine. Moreover, memory elasticity can be used for moving data across VMs. There, we can consider a buffer of memory blocks which are removed from one VM and added to another VM, implementing send and receive operations inside a single server. Given that the size of the buffers is not generally high in these cases (order of few pages), these operations can enable fast inter-VM communication that bypass the network stack. In general, it may be required to release memory which contains useful data for the OLTP engine. In this case, solutions like anti-caching can be used to store the least popular data to the disk. However, anti-caching brings overheads, which depend on the data size and access patterns and are investigated in [18].

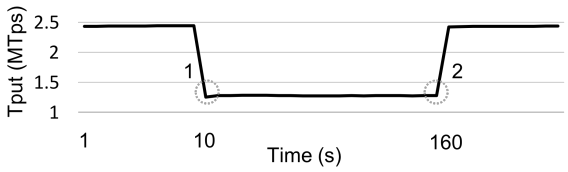

Figure 13: Throughput as memory size changes

Experimental setup. We run an experiment where we dynamically remove and add memory back. When the memory is removed, it is stored to a file in the hard disk, from which it is read again when memory is added back. We consider a single VM with Trireme hosting a 3.6GB database, logically partitioned across 36 threads running on two sockets of our server. Each thread is responsible for a data partition consisted of records and associated metadata of around 500MB. Note that there is an overhead from Trireme on the size of the database, which is due to the nature of the system being a testbed supporting several concurrency control protocols and system architectures. Nevertheless, this does not impact the insights of the experiment. Transaction execution lasts for 10 seconds with 36 threads, and then the 18 ones running on the second socket are requested back by the hypervisor, along with their memory. Each thread stores its partition to a file, releases its memory and then its CPU. After all CPUs and memory has been removed, Trireme waits for 5 seconds and it requests them back from the hypervisor and starts again the worker threads running on the second socket which reload the data from the files.

Timeline. Figure 13 shows the performance of Trireme. As expected, after the first 10 seconds, performance drops about to half, since the CPUs of the second socket are only used to write data to the disk. Each worker thread writes its data in a different file to avoid synchronization overheads. Writing and reading data to/from the disk takes 70-75 seconds every time, since the size is about 9GB and the disk is a SATA 7200RPM HDD accessed as a shared directory over QEMU. This explains that at 10 seconds (pt. 1) throughput drops and until 160 seconds (pt. 2), when data have been reloaded and Trireme executes transactions on both sockets, like in the beginning. The cost for memory hotplug operations is similar to the CPU, between 10-20ms.

Control flow. The Resource Manager requests the VM to release CPUs and memory from the second socket after 10 seconds, and make it available again 5 seconds later. The Host Monitor polls the hypervisor every second. When it receives the notification to release memory and CPUs, Trireme is informed and it writes the memory to the files to finally release it along with the CPUs. Then, after 5 seconds, the Host Monitor receives the notification from the hypervisor that memory and CPUs are available, so Trireme takes them back and restores its previous configuration.

5. DISCUSSION

Cloud elasticity is typically realized in a scale-out fashion. Even though scale-out elasticity allows us to expand throughout a whole cluster, we show that it also imposes significant overheads in the execution of stateful services, like OLTP, due to coordination across the distributed partitions of the system state. These overheads cannot be amortized in workloads with short-lived variations. Such variations have been addressed by scheduling solutions in the cloud, whereas services like AzureSQL and AWS Aurora already offer scale-up elasticity. Despite the relevant work, there has been no end-to-end architecture that relies on open source software and allows the enforcement of different schedules to stateful applications running inside VMs. Further, current elastic infrastructures only support a pre-defined range of database engines and services. Instead, our system design can encapsulate different types of stateful applications, with an elastic thread and memory pool. Thus, it allows the deployment of third-party applications in the cloud and provides scale-up elasticity with respect to their service-level agreements.

The combination of CPU and memory elasticity has several practical use cases. First, it allows applications to adapt to traffic spikes which require additional processing and memory to serve additional requests, like placing new orders. Second, it allows applications deployed in isolation to communicate by exchanging memory and data-local compute resources, for instance when considering heterogeneous workload scheduling [22], or even HTAP systems, where data and compute are exchanged between different engines following an elastic schedule [31]. Third, it allows applications to execute periodic batch jobs at times when several resources are available, for instance index and log file maintenance, data compaction and integrity checks.

The use of a generic infrastructure for elastic scale-up deployments, also brings up a new spot instance model for stateful application. We show that, in this model, compute and memory resources can be added and removed to and from the VM, dynamically at runtime. This way, users can trade resources based on their workload requirements at every point in time. In this case, the benefit for the cloud provider is that host several users in the same virtualized server. While, the benefit for the users is that they can benefit from a pay-as-you-use pricing model, rather than a pay-as-you-reserve, like it is today.

6. CONCLUSION

In this paper we present a system design providing fine-grained CPU and memory elasticity to scale-up OLTP systems. Our design cuts through the whole virtualization software stack and introduces novel components in the hypervisor and the virtual machine. We show that the extensions required by the OTLP engines are minimal and, thus, any application with elastic thread and memory pool can leverage our system. We present the benefits of scale-up elasticity in different practical use cases, which are further used to evaluate the performance of our system. We demonstrate that our system design enables a new spot instance model for stateful services, like OLTP, where instances are started with minimum resource guarantees. Finally, we show that our system, despite more generic, does not impose significant overheads compared to specialized, database-specific approaches, whereas it avoids overheads like data repartitioning and distributed transaction coordination.

7. ACKNOWLEDGMENTS

The authors would like to thank Anthony Iliopoulos for providing the first hotplug implementation from within a VM. This work was partially funded by the EU H2020 project SmartDataLake (825041).

8. REFERENCES

[1] Cluster configurations with dynamic LPARs. https://www.ibm.com/support/knowledgecenter/SSPHQG_7.2/concept/ha_concepts_config_lpar.html.

[2] Features/CPUHotplug – QEMU. https://wiki.qemu.org/Features/CPUHotplug.

[3] HPE Intergrity Superdome 2 QuickSpecs. https://h20195.www2.hpe.com/v2/GetPDF.aspx/c04123326.pdf.

[4] Kunlun Mission Critical Servers – Huawei. http://e.huawei.com/en/products/cloud-computing-dc/servers/mc-server/kunlun.

[5] Oracle SPARC M8-8 Server. https://www.oracle.com/servers/sparc/m8-8/.

[6] SAP HANA on AWS. https://aws.amazon.com/sap/solutions/saphana.

[7] vSphere Web Services API. https://code.vmware.com/apis/358/vsphere.

[8] R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64, 2012.

[10] H. Avni, A. Aliev, O. Amor, A. Avitzur, I. Bronshtein, E. Ginot, S. Goikhman, E. Levy, I. Levy, L. Fuyang, L. Mishali, M. Yeqin, N. Pachter, D. Sivov, V. Veeraraghavan, V. Vexler, W. Lei, and W. Peng. Industrial Strength OLTP Using Main Memory and Many Cores. *PVLDB*, 13(12):3099–3111, 2020.

[11] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, and P. J. Shenoy. "cut me some slack": latency-aware live migration for databases. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 432–443, 2012.

[12] S. Blagodurov, D. Gmach, M. Arlitt, Y. Chen, C. Hyser, and A. Fedorova. Maximizing server utilization while meeting critical slas via weight-based collocation management. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 277–285, May 2013.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.

[14] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, 2013.

[15] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1923–1934, 2016.

[16] S. Das, V. R. Narasayya, F. Li, and M. Syamala. CPU sharing techniques for performance isolation in multitenant relational database-as-a-service. *PVLDB*, 7(1):37–48, 2013.

[17] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[18] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[19] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 301–312, 2011.

[20] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 20, 2012.

[21] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. GRAPHENE: packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 81–97, 2016.

[22] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 519–532. USENIX Association, 2018.

[23] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[24] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[25] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, Santa Clara, CA, 2015. USENIX Association.

[26] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 253–267, 2018.

[27] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[28] J. F. Pérez, R. Birke, M. Björkqvist, and L. Y. Chen. Dual scaling vms and queries: Cost-effective latency curtailment. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 988–998, 2017.

[29] D. Porobic, I. Pandis, M. Branco, P. Tozun, and A. Ailamaki. Characterization of the impact of hardware islands on oltp. *The VLDB Journal*, 25(5):625–650, Oct. 2016.

[30] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, 2016.

[31] A. Raza, P. Chrysogelos, A. G. Anadiotis, and A. Ailamaki. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2043–2054. ACM, 2020.

[32] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5):95–103, 2008.

[33] T. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 337–350. ACM, 2013.

[34] M. Schüle, P. Schliski, T. Hutzelmann, T. Rosenberger, V. Leis, D. Vorona, A. Kemper, and T. Neumann. Monopedia: Staying single is good enough - the hyper way for web scale applications. *PVLDB*, 10(12):1921–1924, 2017.

[35] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12):1035–1046, 2014.

[36] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojevic, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 433–448, 2019.

[37] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith. Runtime vertical scaling of virtualized applications via online model estimation. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 157–166, Sep. 2014.

[38] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. J. Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 205–219, 2018.

[39] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *PVLDB*, 8(3):245–256, 2014.

[40] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052, 2017.

[41] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 789–796, 2018.

[42] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdarson, Y. Vigfusson, and A. Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 162–173, 2018.

[43] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. Sdpaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 68–81, 2018.