Mitigating nesting-agnostic hypervisor policies in derivative clouds

Chandra Prakash, Prashanth, Purushottam Kulkarni, Umesh Bellur
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India
{*chandrap, prashanth, puru, umesh*}*@cse.iitb.ac.in*

***Abstract*—The fixed granularity of virtual machines offered by IaaS providers has prompted the evolution of ”derivative clouds” - one where a leased VM is broken down into smaller and variable sized units by a secondary provider who then offers these typically as PaaS instances. The derivative cloud is powered by container technology owing to the benefits of low overhead and fast spinup of containers compared to virtual machines.**

The guest OS manages (distributes to different containers) virtual resources inside a VM whereas the hypervisor manages the physical resources that are distributed among VMs. This results in two control centers over the set of resources used by containers. The hypervisor takes control actions such as memory ballooning or the withdrawal of a virtual CPU to manage over-provisioning without being aware of the effect these actions will have on individual containers inside the VM. The derivative cloud provider who has allocated resources to the PaaS instances executing on containers in the VM now needs a mechanism to react to such changes - based on resource management policies they setup for the purpose.

In this work we first analyze the consequences of hypervisor actions such as memory ballooning and vCPU scaling which are used to manage memory and CPU over commitment respectively on nested containers. Based on this analysis, we offer a design of a policy driven controller that smoothes over the effect of these hypervisor actions on containers nested inside a VM. We expose several useful policies for each resource type (CPU and Memory) that can help derivative cloud providers manage PaaS instances.

1. Introduction

A Derivative cloud is an intermediate layer between a cloud provider (such as Google Cloud Platform [1], Amazon EC2 [2] or an Enterprise cloud) and cloud service consumer. Derivative clouds [3] deliver services (infrastructure, platform or software) hosted on virtual machines purchased from primary IaaS providers. This model is motivated by the mismatch between the granularity of VMs leased by the IaaS providers and that of the services consumed off of that infrastructure [4]. For example, a PaaS provider [5] can offer fine grained PaaS instances that are co-hosted on larger VMs leased from infrastructure cloud providers. Further, motivation stems from the pricing schemes and lack of standardization of of IaaS providers that encourage the purchase of larger VMs that then drive the derivative cloud providers to share this infrastructure for multiple IaaS/PaaS/SaaS instances [6].

PaaS providers resolve this granularity mismatch by using nested virtualization [7], [8]. However, owning to the drawbacks of nested system virtualization (performance overheads) [6], [9], they employ light weight containers [10] corresponding to PaaS instances inside the leased VM. In addition to lower overheads, containers provide other benefits such as faster provisioning, faster spin-up and cost effectiveness [11]–[15].

Infrastructure providers frequently over commit resources, primarily along the CPU and Memory axes [16]–[18]. Today’s hypervisors provide adequate controls to manage this over-commitment using techniques such as memory ballooning [19] and dynamic vCPU (de)allocation [20], [21], vCPU scheduling [22] etc. However, hypervisor actions are limited to managing physical resources whereas the virtual resources allocated to a VM are managed by the guest OS. This separation of responsibilities results in two control centers over essentially the same set of resources. In such a setup, hypervisor actions are agnostic of the way the VM is managing its resources internally. The fact that virtual resources are further divided amongst a set of nested containers is of no consequence to the hypervisor. It is therefore entirely possible that a hypervisor action adversely affects a containers nested inside the VM.

A thorough survey of prior art indicates that the effects of hypervisor actions on nested containers has not been quantified thus far. Early indicators of our empirical studies point out that the impact of hypervisor actions relating to managing over-commitment are non-uniformly distributed over the multiple containers that are nested inside the VM. It is therefore imperative to bring a sense of control to this process and allow user specified policies to guide the manner in which hypervisor action affect nested-containers. Our goals are therefore to:

(i) Quantify the consequences of hypervisor actions related to managing memory and CPU over-commitment.

(ii) Design and incorporate policy driven control over the effect that hypervisor actions to manage over-commitment have on services provided via nested-containers.

In this paper, we analyze the impact of various hypervisor level resource management actions (for memory and CPU) which are agnostic to the containers executing inside the virtual machines and provide solutions for deterministic and reduced impacts. First, we show the impact of nesting-agnostic actions using the memory ballooning technique [19] for managing over-commitment scenarios by the hypervisor. A similar analysis is performed for hypervisor CPU management using dynamic scaling of vCPUs assigned to a virtual machine. Dynamic vCPU scaling [20] is a hypervisor-based technique that aims to mitigate the impact of VM preemption on synchronization and IO performance. Then we design and incorporate policy driven solutions to control the effect of these hypervisor actions. We evaluate the effectiveness of our solution by using several synthetic and realistic workloads.

2. Nesting-agnostic resource management

This section provides and the background and motivation for our work. We demonstrate the effects of nesting-agnostic resource management actions of the hypervisor in resource over-committed scenarios. The scope of this work includes the memory and the CPU resources.

2.1. Memory management issues

Memory over-commitment, a common resource allocation technique in virtualization based infrastructure-as-a-service solutions [8], [23], [24], over-provisions memory relative to the physical memory capacity. Several hypervisor-based techniques—demand paging, ballooning [19], memory deduplication [25], [26], and hypervisor-based caching [8]—exist to multiplex memory across virtual machines in over-commitment setups. As part of this work, we focus in the interplay of hypervisor based memory ballooning actions and its impact on nested containers. Typical usage of memory ballooning based management employs a balloon controller which works in conjunction with the hypervisor to change the effective memory allocation of virtual machines. Dynamic memory allocation is achieved through inflating or deflating a memory balloon within virtual machines. Which virtual machines to select for ballooning and the extent of ballooning on policies that the controller employs to adhere to per-VM SLAs. In a derivative cloud setup, hypervisor actions are nesting-agnostic and enforce policies based on per-VM SLAs without any knowledge and intent about impact on applications within containers.

2.1.1. Memory management with nested containers.

With the Linux cgroups-based containers, memory is provisioning to each application container using two configuration knobs—hard-limit, the maximum memory that can be use by a container and soft-limit, the minimum memory that an operating system will allocate to a container. In derivate cloud setups, memory to a virtual machine is allocated and managed by the hypervisor, where as *inside* a virtual machine, the guest operating system controls allocations to the nested containers. The Linux cgroup subsystem also employs per-container policies and system-wide policies to multiplex memory across containers. For example, if a container tries to allocate memory beyond its hard-limit, a set of pages local to the container will be evicted for fulfilling the required allocation. Further, the system can be under memory pressure either due to simultaneous demand for memory from execution entities of the virtual memory or due to extraneous pressure generated to ballooning operations. In such situations, the Linux cgroups subsystem resorts to system wide memory reclamation across containers.

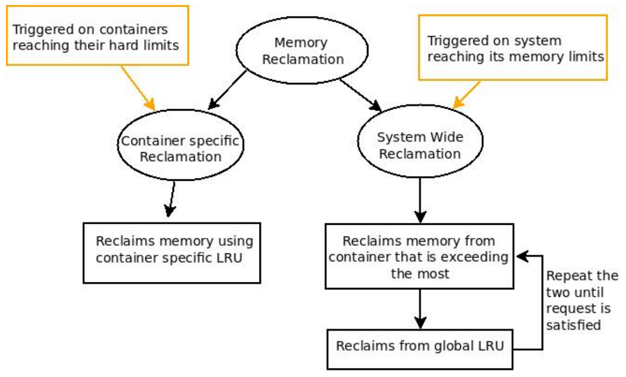
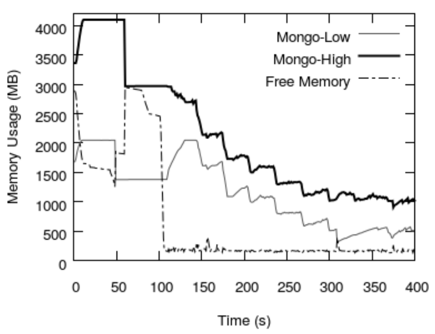
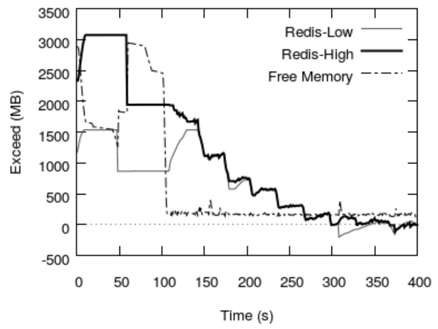

Triggered on containers reaching their hard limits
Memory Reclamation
Triggered on system reaching its memory limits
Container specific Reclamation
System Wide Reclamation
Reclaims memory using container specific LRU
Reclaims memory from container that is exceeding the most
Repeat the two until request is satisfied
Reclaims from global LRU

Figure 1: Existing policy for Memory Reclamation

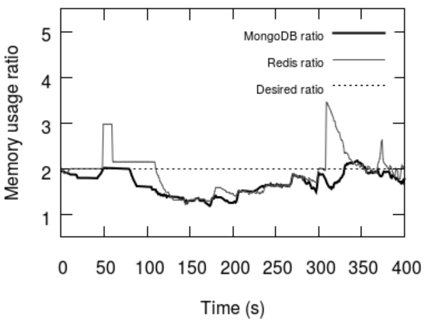
To orchestrate these reclamations, Linux maintains a list of memory pages maintained in least-recently-used order for each container, and also a list of system-wide memory pages in LRU order. The reclamation policy employed by the Linux cgroups subsystem is illustrated in Figure 1. During system-wide memory pressure, i.e., when the operating system is required to reclaim pages across containers, the extent of *exceed* relative to soft-limit of each container is estimated. Every system-wide memory reclamation request is triggered by the system memory dropping below it's free memory threshold. This reclamation request is primarily targeted at an container that is exceeding it's soft-limit by the largest value —we refer to this process as the soft memory reclamation(SMR). Depending on amount of memory being reclaimed using SMR, the global page list is also used to evict pages based on a LRU ordering across all pages — we refer to this process as the global reclamation policy (GLR). Experimental evidences, show that the amount of pages reclaimed using GLR is inversely proportional to the pages reclaimed using SMR. The global reclamation policy is agnostic to containers and with derivative cloud setups, as when memory is reclaimed from the VM, memory reclamation from containers in a non-deterministic manner is expected.



(a) Memory usage for Redis applications.



(b) Extent of memory usage exceed for Redis applications.



(c) Memory allocation ratios.

Figure 2: Plots illustrating limitations in maintaining memory ratios in derivative clouds

Container	hard-limit (GB)	soft-limit (GB)	Key Size (# records)	Usage (GB)
Redis-Low	2	0.5	500K	1.3
Redis-High	4	1	1000K	2.6
Mongo-Low	2	0.5	500K	1.3
Mongo-High	4	1	1000K	2.6

TABLE 1: Default configuration of the four application containers running in the derivative cloud setup

2.1.2. Limitations of reclamation in derivative setups.

To establish the limitations of nesting-agnostic memory reclamation we performed the following experiments. Two Linux+KVM virtual machines were used, one in a nesting setup, which hosted containers, the second generated workloads for the container-hosted applications. The nested VM was provisioning with 6 vCPUs and 16 GB memory and the workload generating VM has allocated 1 vcPU and 6 GB memory. The nesting setup consisted of four Docker containers [27] which executed with Redis [28] and MongoDB [29] workloads. Default configurations of the applications executed within the contains is as shown in Table 1. The YSCB [30] workbench was used to generate workload datasets and as a workload generator. For the first 100 seconds the applications were executed without memory pressure to consume as much memory as required. Beyond 100 seconds, memory pressure was generated from the host and memory from the nested VM reclaimed at rate of 2 GB every 30 seconds.

Figure 2a shows the memory consumed by the containers executing the two Redis workloads and free memory available in the virtual machine. The two containers are configured to consume memory in the ratio of 1:2 (refer to Table 1), via hard-limit and soft-limit specifications. At 50 seconds, both the Redis workloads, which are already at their hard-limit, try to allocate more memory, this results in container-specific memory reclamation—the free memory in the system increases and the memory usage of the two containers drops to 1.35 GB and 2.7 GB. At 100 seconds, the hypervisor-based balloon controllers starting exerting memory pressure—demanding 2 GB every 30 seconds. As shown in Figure 2a. just after 100 seconds, the free memory in the system drastically drops. The extent of memory *exceed* beyond the soft-limit specifications is shown in Figure 2b. Since, the Redis-high application exceeds by a large extent (2 GB more that soft-limit), the cgroups subsystem penalizes this container for reclamation. Further, due to the free memory available, and reduced memory consumption of the Redis-high workload, the usage for the Redis-low application increases for an epoch and just before 150 seconds, the extent of exceed beyond the soft-limit specifications for both the containers is similar. *Note that this is how the cgroups subsystem operates, it penalizes the container with the maximum extent of exceed, till all of them exceed by the same absolute value and then penalizes them equally.* This process essentially does not adhere to the memory usage ratios that were expected with the soft-limit and hard-limit specifications. Figure 2c shows the memory usage of the Redis and MongoDB applications, which drops below the 1:2 ratio from 100 seconds to 300 seconds. At 300 seconds, the extent of exceed decreases to zero and the system switches to the global reclamation (GLR) mode. Beyond this point, the memory reclaimed is not container-agnostic and variable ratios of usages are observed.

The key takeaways from this experiment are as follows,

- Memory reclamation uses the extent of exceeded usage above a containers soft-limit. Containers with higher exceed values are penalized to a larger extent.
- The Linux cgroups systems attempts to maintain similar extent of exceed for memory usage during reclamation when all container are above their soft-limit's.
- A per-container soft-limit and hard-limit specification does not guarantee proportionate memory allocation during memory pressure situations.

This Linux cgroups reclamation policy does not accommodate different user-specified policies that are desired—proportionate memory usages at all instances, order-of-reclamation across containers etc. This is especially required in nested hosting environments where derivative service provides would be benefited by providing a rich set of prioritization features.

2.2. CPU provisioning issues

Similar to memory over-commitment, CPU over-commitment is a common technique for improving CPU efficiency in virtualization environments. Guest operating systems in virtual machines schedule processes on vCPUs, which in turn are scheduled by the hypervisor on physical CPUs [21], [31], [32]. In the derivative cloud setup, the set of vCPUs allocated to a VM can be distributed among multiple containers by using two Linux cgroups specification parameters. Either one of these parameters or a combination can be used for provisioning.

(i) **cpuset.cpus:** This parameter lists the set of vCPUS on which processes of a container can be scheduled. In effect, a container is *pinned* to the specified set of vCPUs.

(ii) **cpu.share:** This parameter is used to specify a weight to estimate the relative share of a vCPU for a container from its set of assigned vCPUs. The CPU share parameter is applicable only to the vCPUs in the intersection set of containers—vCPUs assigned to single containers are not impacted.

Pinning containers to vCPUs/CPUs has the potential to provide several benefits—efficient management of larger computer systems [33], resource isolation, improved throughput and improved power utilization [11]. In the context of our work, we envision container deployments aimed at *pinned* CPUs to realize the above benefits. To achieve the benefits of pinning while executing containers inside a virtual machine, containers must be pinned to the vCPU(s) and virtual machines should be pinned to physical CPUs. Further, with over-commitment virtualization setups, dynamic scaling of vCPUs is employed to reduce scheduling overheads and address synchronization related preemptions issues [20], [34]–[36]. We envision CPU pinning to be desired feature and an available specification as part of the application provisioning service.

In a derivative setup, dynamic scaling of vCPUs of a VM, will impact the cpuset of containers and as result also the CPU pinning specification. Next, we demonstrate this adverse impact on vCPUs due to dynamic vCPU scaling in derivative setups.

2.2.1. Non-deterministic CPU sharing. We establish the non-determinism in CPU allocation in derivative setups using the following setup and experiments. A virtual machine provisioned with seven vCPUs and 8 GB memory hosted three Docker containers [27] and the desired CPU allocation across the three was 1:1:4. The CPU allocation for the containers had two configurations,

(i) **without CPU pinning**: all containers were schedulable on all six vCPUs (vCPU0 is not used) and the CPU share [37] weights were specified in the ratio 1:1:4.

(ii) **with CPU pinning**: in this configuration, the first

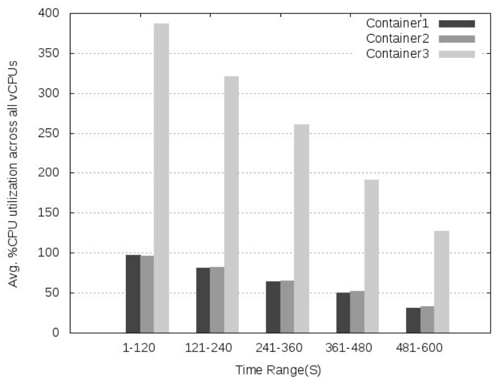


Figure 3: CPU utilization without CPU pinning.

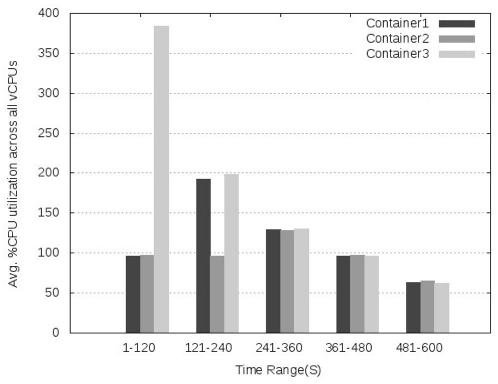


Figure 4: CPU utilization with CPU pinning.

container (Container 1) was pinned to vCPU1, the second (Container 2) pinned to vCPU2 and the third pinned to vCPUs 3, 4, 5 and 6.

We used the Sysbench [38] benchmark to a generate CPU intensive workload (generation of prime numbers) for the containers. Each container is configured with four threads and executed for 600 seconds. We executed workloads in the three containers simultaneously as a part of the hypervisor-based vCPU scaling operation reclaimed the virtual machine by one vCPU every 120 seconds and stopped scaling when the VM had three vCPUs remaining. The order of reclamation was vCPU1, vCPU2, vCPU3 and vCPU4. vCPU0, the master vCPU0 did not go offline.

Figures 3 and 4 show the average CPU utilization of each container across all vCPUs in the two CPU allocation configurations (without pinning and pinning, respectively). As can be seen in Figure 3, without CPU pinning of containers, vCPU scaling actions do affect the CPU provisioning across containers. The allocation ratio of 1:1:4 is maintained even when 4 vCPUs are reclaimed from the virtual machine. With CPU pinning to vCPUs the allocation ratios do not hold (as seen in Figure 4). The allocation ratio achieved is 2:1:2 after a single vCPU is reclaimed and then settles to a ratio of 1:1:1 for later reclamations. After the first 120 seconds, the first container has no pinned vCPUs available, as a result the Linux cgroups implementation moves the container to the

set of vCPUs of its ancestor—in this case the root docker node. The root docker has all CPUs in its set and hence the first container is schedulable on the vCPUs. Since Container 2 is pinned to vCPU2, vCPU3, vCPU4, vCPU5 and vCPU6 are shared equally by Container 1 and Container 3. After 240 seconds, when CPU set of Container 2 also becomes empty, all the three containers share the available CPUs in equal proportion.
To summarize,

- Hypervisor-based vCPU scaling is nesting agnostics and with CPU pinning of containers, CPU utilizations are not deterministic across containers. A desired feature is to enable deterministic CPU proportioning even with hypervisor initiation vCPU scaling.
- Since relative CPU proportions can be deterministically enforced without pinning CPUs, an intended goal would be to simultaneously achieve benefits of CPU pinning and deterministic CPU sharing.

3. Policy Driven Reactivity

In this section, we are present our solution which takes the form of user-specified policy driven reactions to hypervisor-based actions for managing the memory and CPU resources. The proposed solution design is agnostic of the underlying hypervisor or the container manager used, there by making it as generic as possible.

3.1. Memory Distribution

Enforcing memory allocation proportions across containers when no memory pressure exists is possible with the soft-limit and hard-limit configuration parameters. However, as discussed in Section 2.1, these knobs do not provide deterministic memory provisioning when the system is under memory pressure. As part of this work, we design for two new policies to provide deterministic memory provisioning in nested setups under memory pressure.

Policy 1: Proportionate memory allocation
This policy aims to ensure that all memory allocated to container is based on relative weights specified as configuration parameters. The proportional memory allocation is enforced during situations of memory pressure and during no pressure. For example, consider two containers with an intended proportionate memory allocation in the ratio 1:2. Further, assume that their soft-limit values are set to 1 GB and 2 GB, respectively, and their current usage is 2 GB and 4 GB respectively. With a memory reclamation demand of 1 GB, most of the memory will be reclaimed from the container with the larger extent of usage, since the extent of exceed is 2 GB as compared to 1 GB of the smaller container. The resulting usages after reclamation will be 2 GB and 3 GB, respectively, violating the proportionate ratio of 1:2. The proposed proportionate allocation policy aims to maintain the 1:2 ratio in all memory pressure situations.

Policy 2: Application-specific differentiated allocation
This policy provides the flexibility of providing specific rules and or categories to application containers, e.g. gold, silver, bronze etc. Containers mapped to each of these categories have different reclamation rules or the categories themselves can imply an ordering for reclamation.

3.1.1. Modification to Linux cgroups subsystem. We modify memory Linux cgroups memory management sybsustem to conform to the polices specified earlier. A new per-cgroup state variable is introduced to specify a *weight* parameter for each container and is also exposed through `sysfs` interface for every container. These weights enforce a notion of priority among containers; higher the weight for a container the higher its priority and proportion. The relative ratio of the weights, dictate the proportion of memory allocation. Cloud providers can modify these weights in the derivative setup (also applicable to native cloud providers using containers for provisioning) to enforce priority among containers executing within VMs.

3.1.2. Memory exceed estimation. When the VM in which the containers are executing comes under memory pressure, memory is reclaimed using the container specific LRU lists (as SMR) as well as the Global LRU list (as GLR) by the guest operating system. Memory reclamation depends on the extent of *exceed* in memory usage above the specified soft-limit—estimated as the difference in the two values. The container having a higher *exceed* is victimized during memory pressure first. We redefine this notion of *exceed* with one that is based on proportionality weright of each container. The proportional allocation of a container is the ratio of the container weight to the total weight across all containers in a VM multiplied by the total memory usage across all containers (Equation 3). The proportionate *exceed* value is the difference of the container's proportional memory allocation and its current usage (Equation 4). Finally, the memory reclamation policy is modified to target the container having the highest proportionate *exceed* value for each reclamation request. Similar to the default reclamation policy, the global LRU list for reclamation is used once memory usage of all containers is below soft-limit specifications.

$$T\_U = \sum_{i=1}^{n} U_i \tag{1}$$

$$T\_W = \sum_{i=1}^{n} W_i \tag{2}$$

$$PA_i = T\_U \times (\frac{W_i}{T\_W}) \tag{3}$$

$$EX_i = U_i - PA_i \tag{4}$$

U_i: Memory usage of i^{th} container
W_i: Relative weight of i^{th} container
T_U: Total memory usage by all containers
T_W: Summation of relative weights of all containers
PA_i: Proportional memory allocation of i^{th} container
EX_i: Proportionate exceed value of i^{th} container

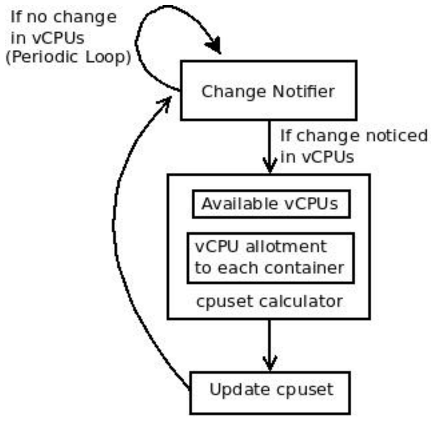


Figure 5: Flow of cpuset update

3.1.3. Reclamation granularity. The current soft-memory reclamation policy (SMR), targets one container and reclaims a non-deterministic amount of memory from it for every request. The extent of reclamation depends on the anonymous region (from which a fixed size is reclaimed) and the disk page cache (from which a large portion is reclaimed). Since the page cache size is non-deterministic, reclamation extent can also be non-deterministic. This may lead to a container being targeted exclusively for a particular reclamation request, leading to large deviations from weighted allocations. To overcome this issue, we have capped reclamation chunk size to a maximum of 50 MB.

3.1.4. Maximizing deterministic reclamation. As part of the Linux cgroups memory reclamation process, soft-reclamation (SMR) reclaims memory from each container till memory usage is above the soft-limitspecification. Simultaneously, a small amount of memory is also reclaimed from the system-wide pool of pages (GLR). In our solution, to maximize deterministic reclamation we perform proportionate reclamation across containers till the usage of containers reduces to zero. Beyond this situation, the system-wide global LRU list is used for further reclamations. Since the SMR policy is container-aware, this strategy provides us a better handle over memory provisioning for containers.

3.2. CPU Reallocation

In Section 2.2, we demonstrated the Linux cgroups CPU management process with and without pinning containers to vCPUs. With hypervisor-based vCPU scaling policies for mamangement actions in nested environments, the aim is to maximize provision pinned vCPUs to containers.
To recap, the Linux cgroups subsystem provides two knobs for CPU allocation—cpuset (the set of vCPUs assigned to a container) and cpushare (the relative share of a container on shared vCPUs). With no CPU over-commitment CPU allocation is not an issue, with over-commitment allocation has to be managed carefully to meet allocation proportions. Our solution aims to maximize the number of pinned vCPUs to each container while maintaining the user-specified

Algorithm 1: Procedure for CPU set and share estimation.

```
Data: $S_i$:cpu.share of $i^{th}$ container, L: List of available
      CPUs for containers, N: Available CPU capacity
Result: CPU set mapping for each container
1  L_S = null (List of containers requiring shared CPUs.)
2  T_S = $\sum_{i=1}^{n} S_i$: summation of cpu.share of all containers.
3  foreach Container do
4      $share_i = \frac{N \times S_i}{T\_S}$
5      $X_i = \lfloor share_i \rfloor$; $Y_i = share_i - X_i$
6      $cpuset_i = X_i$ CPUs from L (remove CPUs from L)
7      if $Y_i \neq 0$ then
8          L_S.insert($Y_i$)
9      end
10 end
11 Assign remaining CPUs from list L to containers in the list L_S
   and update corresponding $cpuset_i$ entries.
```

proportionality constraints. Since the set of vCPUs assigned to a VM is dynamic, maintaining these proportions needs a revised CPU allocation process. Our solution is presented as Algorithm 1. The input to our algorithm is the *share* of the CPU capacity to be allocated to each container and the set of CPUs mapped to each of them. The share is used as a weight to estimate relative CPU allocation among the containers and the CPU set is the mapping of CPUs (when the CPUs are available and not reclaimed by the hypervisor). The algorithm tries to allocate as many dedicated CPUs as possible to the container from its CPU set and then provisions shared CPUs while maintaining the proportional allocation constraints derived from the CPU share specifications. The number of vCPUs are periodically check and if a change is noticed, the CPU set for each container is updated (refer Figure 5). The dynamic configuration of the CPU set for each container is the output of Algorithm 1.
Referring to Algorithm 1, $share_i$, is the number of proportionate number of vCPUs to be assigned to a container based on the available capacity and its relative proportion compared to all contaniers. If $share_i$ is a non-fraction, then the procedure allocates that many vCPUs and updates the number of available vCPUs. If the value is fraction, the procedure uses the floor of the $share_i$ value to allocate number of vCPUs. Containers whose proportionate request is not satisfied are added to a temporary list. After iterating through all the containers to estimate their mappings $cpuset_i$, the remaining vCPUs are allocated to all the containers stored in the temporary list. These containers formed the set of containers to share the set of CPUs available and are unallocated. The shared CPUs are shared in the proportions based on the CPU share specification (S_i) of each container.
Special case: Algorithm 1 maximizes CPU pinning for all containers in a VM while trying to maintain allocation ratios. However, there can a boundary case where CPU pinning may not be requested for a subset of containers

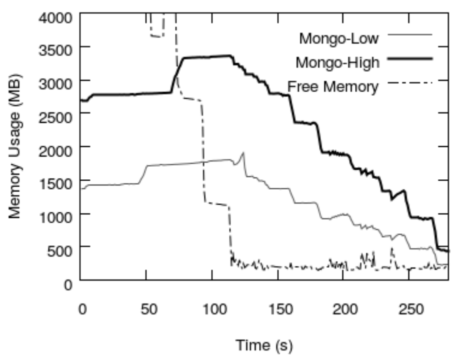


Figure 6: Memory usage by Mongodb containers when SL=HL, using modified controller

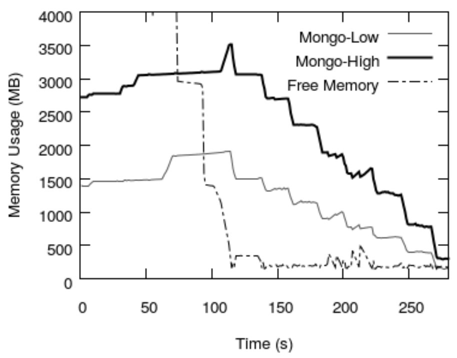


Figure 7: Memory usage by Mongodb containers when SL!=HL, using modified controller

running in the system (specified by an empty cgroups *cpuset* specification). To account for this, we add all such containers to the list L_S before the algorithm starts estimation of vCPU mappings for pinning. This ensures that these containers are only offered shared vCPUs and which are proportionally shared using the CPU share specification.

4. Evaluation

We will show the effectiveness of our approach empirically in this section. Experimental set-up for the most part remains same as that mentioned in Section 2. Any changes to the set-up is mentioned in the respective subsections.

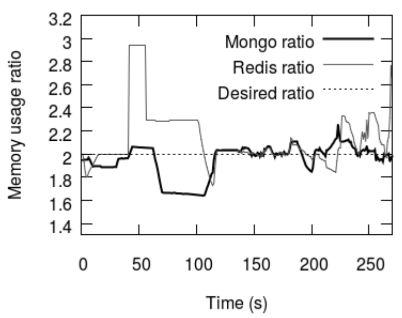
4.1. Impact on memory distribution

Experiments are performed with the same workload and set-up except for the size of VM-1. At the beginning of the experiment, size of VM-1 is 20 GB which is gradually reduced to 3.5 GB, at the rate of 1.5 GB every 20s (started at 50s). Relative memory weights for the MongoDB containers are in the ratio of 1:2 (1 for low and 2 for high priority containers) as are the relative weights for the Redis containers. These weights are added using the newly added *weight* parameter in the memory *Cgroup*.

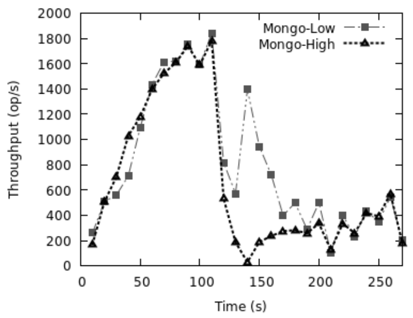
4.1.1. Memory usage and throughput. We have tested the effectiveness of our approach in both cases: (i) Soft Limit is equal to Hard Limit (SL=HL) and (ii) Soft Limit is not equal to Hard Limit (SL!=HL), because memory reclamation methods for both cases are different as mentioned in Section 2.1.1.
Figure 6 and Figure 7 depicts the memory usage patterns for MongoDB containers for cases (i) and (ii) respectively. We observe a gradual decrease in memory allocated to the low and high priority containers in both cases once the system runs out of free memory at about 120 seconds. The ratios of memory allocated to containers can be seen to be in accordance with their relative weights, unlike the earlier case when there is no modified controller. We observed similar pattern in case of Redis container. This pattern of relative memory allocations can be observed more prominently when comparing Figure 2c and Figure 8a which shows memory usage ratios between low and high priority containers (closer to 2, the better) using existing controller and our modified controller respectively.
The effect of better memory allocation is notable in the application throughputs. The existing controller produces better throughputs for the lower priority MongoDB containers in the interval 100-200s as seen in Figure 8b. With our modified controller, both the low priority and high priority containers produce similar throughput as seen in Figure 8c. The similar throughputs observed are strongly connected to the fact that low and high priority containers are provisioned with 1:2 memory allocations and so are the ratio between their usage patterns (Provisioning is done to avoid contention due to any other resource). Similar throughput impacts are also observed with Redis containers. Hence credit share not only produces better memory allocations, but also enhances application performance for containers.

4.1.2. Differential QOS containers. Consider the application specific differentiated allocation policy specified in Section 3.1 where we would like to prioritize a container from being victimized. Let's call this container as a gold container. We have used same set-up (SL!=HL) to test this. We have prioritized the gold container using the higher weights (Weights of the order of 10^3 relatively higher) to avoid reclamation from it unless it's necessary. There are also other containers running inside the VM with lower QOS and let's call them silver containers.
In Figure 9 we have plotted only the memory usage for the gold and one silver container for ease of analysis. We observe that at about 120 seconds free memory drops and the reclamation kicks in. Victimization of gold container is much lesser than other containers until the other containers usage near zero. This shows how we could use our weighted controller to enforce differential service QOS containers.

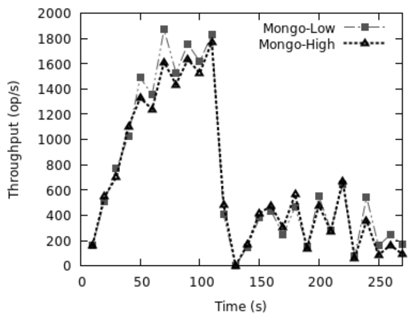
4.1.3. Impact of reclamation chunk size. The native reclamation method has no notion of limit on the amount of memory that can be reclaimed from a targeted container. This leads to larger reclamation amounts from a container thereby deviating it from the desired relative allocations.



(a) Memory usage ratios between MongoDB and Redis containers



(b) Application throughputs with existing memory controller



(c) Application throughputs with modified memory controller

Figure 8: Impact of modified controller on low and high priority containers

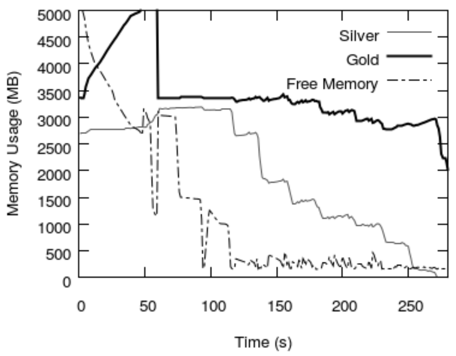


Figure 9: Memory usage plot showing implication of relative weights provide differential QOS guarantees

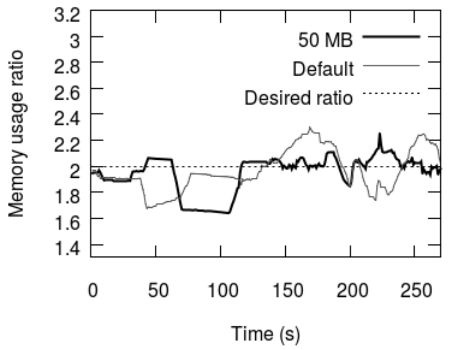


Figure 10: Memory usage ratio with different reclamation chunk size

Figure 10 shows the ratio of memory usage between MongoDB containers for default (native) case and reclamation chunk size of 50 MB. We consider observation after 120 seconds, as that is when the pressure kicks in and triggers reclamation. Although the 50 MB capping of reclamation chunk provides better memory ratios than the default case most of the time, it still deviates from desired behavior in the interval 180-220 seconds as seen in Figure 10.

With smaller maximum reclamation size, we will have finer control over the memory allocation ratios, but this comes at the cost of frequent updation. Further investigation has to be done to fix upon a reasonable size to maintain balance between performance and precision here. For now, the reclamation chunk size is parameterized in our implementation and can be easily modified using a kernel module to alter the maximum reclamation chunk size. We have used reclamation chunk size of 50 MB in our implementation and this applies to all the experiments that we have run with our modified controller.

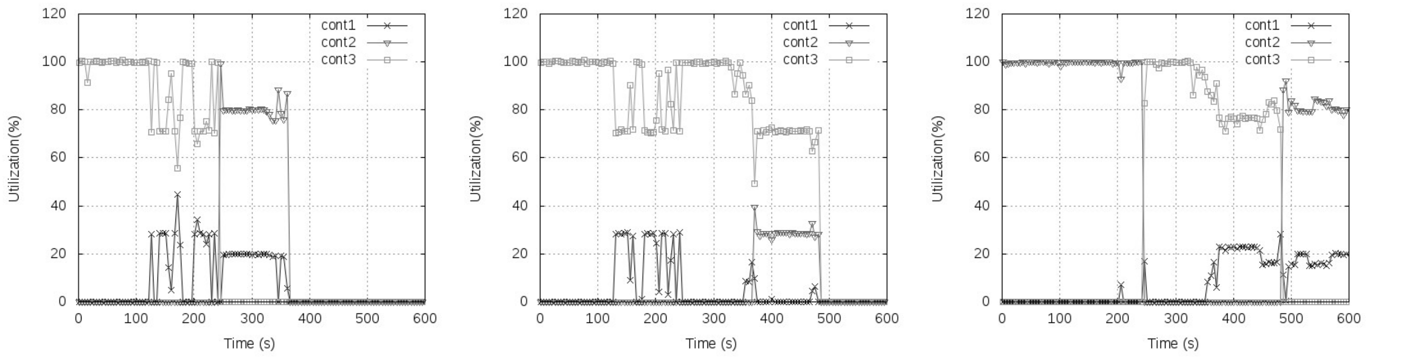
4.2. vCPUs shareability while pinning

Experimental set-up to evaluate vCPUs shareability among containers is similar to the set-up discussed in Section 2.2. However in this case we have maintained the ratio of CPU shares among C1:C2:C3 as 1:4:5.

4.2.1. Relative share of vCPU. We have performed several experiments to verify the proposed Algorithm 1 to find the CPU allocations for each container while maintaining CPU shares and maximizing *cpuset* pinning. We evaluated the CPU utilization for each container by using Sysbench benchmark [38] (prime number calculation) for 600 seconds. In this experiment we reduced the number of vCPUs available to VM by 1 every 120 seconds.

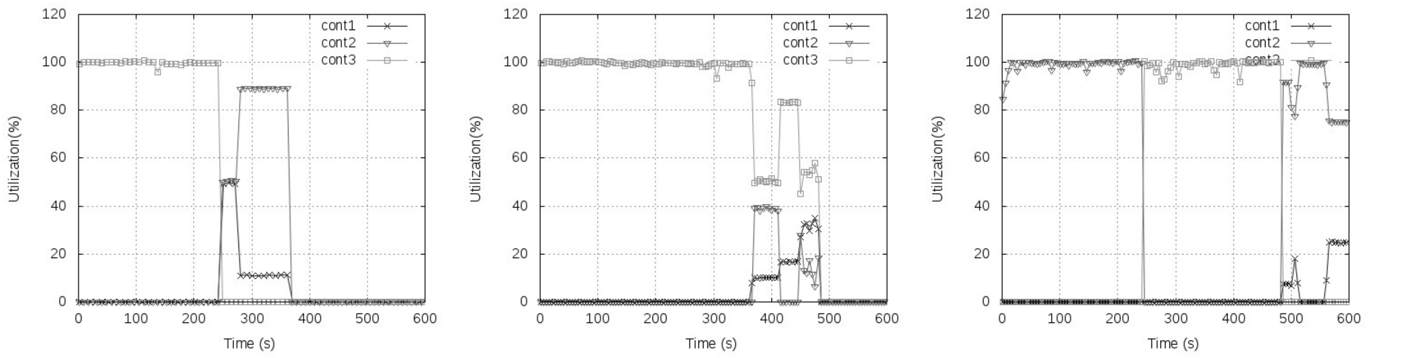
Average CPU utilization for every 120 seconds using our controller is shown in Figure 12. We observe that ratio of CPU utilization maintained is nearly equivalent to the ratio of their respective CPU shares even after vCPU offlining is done. Our updated controller produces much more desirable results as compared to outputs produced using the existing controller illustrated in the Figure 4. Table 2 lists the vCPUs in *cpuset* of each container for every 120s interval. It shows how our controller tries to maximize CPU pinning while maintaining CPU shares.

Table 3 lists the vCPU(s) in *cpuset* of each container at the intervals of 120 seconds for the special case as discussed in Section 3.2. In this case we are trying to allocate dedicated vCPUs to only C3 while maintaining the ratio 1:4:5 across containers C1,C2, and C3.



(a) vCPU3 (b) vCPU4 (c) vCPU5

Figure 11: vCPUs shared by different containers excluding vCPU0



(a) vCPU3 (b) vCPU4 (c) vCPU5

Figure 13: vCPUs shared by different containers including vCPU0

Time-interval	C1	C2	C3
1-120	1	1,5,6	2,3,4
121-240	2	5,6	2,3,4
241-360	3	3,6	4,5
361-480	4	4,6	4,5
481-600	5	5	6

TABLE 2: vCPU(s) for each container at 120 seconds interval

Time-interval	C1	C2	C3
1-120	1,5,6	1,5,6	2,3,4
121-240	2,5,6	2,5,6	2,3,4
241-360	3,6	3,6	4,5
361-480	4,6	4,6	4,5
481-600	5	5	6

TABLE 3: vCPU(s) for each container at 120 seconds interval for the special case, pinning only occurs for C3

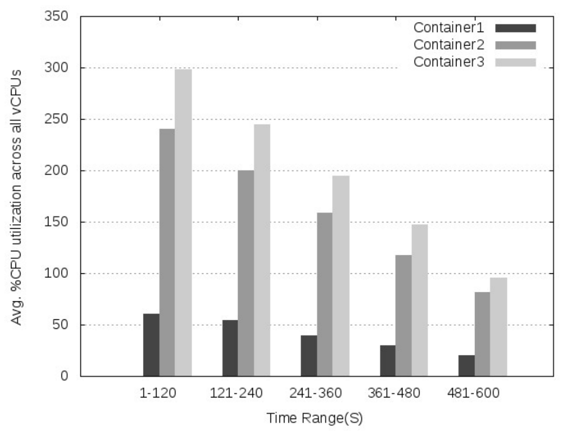


Figure 12: With scaling down of the vCPUs, 1 vCPU every 120 seconds

4.2.2. cpuset anomaly. We have come across an unexpected behavior during our experimentation. According to *cpuset* implementation in linux kernel (cgroup), processes running inside a container can only be scheduled on the CPUs of parent (node) *cpuset* only when its own *cpuset* goes empty state (due to our solution) this still allows processes in that container to use CPUs in its parent *cpuset*. With our solution, the change in state of *cpuset* from empty (due to vCPU offlining) to non-empty (due to our solution) still allows processes in that container to use CPUs in its parent *cpuset*. Table 2 shows the cpusets for all three containers at every 120 seconds intervals We see that state of *cpuset* of C1 will switch from empty to non-empty because vCPUs goes offline in a numerically contiguous way (1,2,3,4,...). In the

time interval of 121-240, C1 should only use vCPU2 but processes inside C1 get scheduled on vCPU3 and vCPU4. Similarly, in the time interval of 360-480, C1 should only use vCPU4 but it uses vCPU5 mostly. Both situation can be seen in Figure 11.
To solve this problem, one of the simple approach we have used is to not let this transition from empty to non-empty *cpuset* occur, by assigning a master vCPU(CPU) that is shared among all containers which will not go off-line (e.g vCPU0). Figure 13 highlights the effectiveness of this approach in which all the containers are respecting their *cpusets*. But this solution will impact the benefits of CPU pinning negatively because that master vCPU is shared among all containers.

5. Conclusion

We demonstrated the adverse impact of hypervisor-based management actions in derivatie cloud setups for the memory and CPU resources. Towards providing deterministic control of resource provisioning of the nesting-agnostic actions of the hypervisor, we devised additional controls in hosted virtual machines to enable user-defined policies for the nested containers. To overcome these nesting-agnostic actions, we have designed a weighted share based memory controller to support a set of user-defined policies. We also devised mechanism to enforce proportionate CPU sharing across containers while maximizing dedicated mapping of CPUs to containers with dynamic CPU capacity changes. We empirically evaluated our solution and the supported policies and demonstrated their applicability. For example, for a desired memory sharing of 1:2 between nested containers, while the default Linux cgroups subsystem could not maintain the sharing proportion during memory pressure, our solution enforced these limits correctly. We also demonstrared that the user-specified CPU proportions are met while maximizing pinned CPUs with dynamic changes in CPU capacity. We believe that changes to the Linux cgroups subsystem, will enable derivative PaaS providers to control the effects of hypervisor-based actions in a much more fine grained manner than what is possible today.
As part of future work, we would like to extend the memory controller proposed in this work to enfore soft-limit constraints. Currently, we ignore the soft-limit specificaiton in favour of proportionate provisioning. We also noticed a possible bug when the CPU set for a container changes from empty to non-empty. In this case, the CPU set of the parent node are still mapped to the container under consideration. This erroneous behanviour needs fixing.

References

[1] (2016) Google cloud platform. [Online]. Available: https://cloud.google.com/

[2] (2016) Amazon aws services. [Online]. Available: https://aws.amazon.com/

[3] (2016) Ravello systems: Virtual labs using nested virtualization. [Online]. Available: https://www.ravellosystems.com/

[4] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, "Spotcheck: Designing a derivative iaas cloud on the spot market." ACM, 2015.

[5] (2016) Heroku derivative cloud hosting provider. [Online]. Available: https://www.heroku.com/

[6] D. Williams, H. Jamjoom, and H. Weatherspoon, "The xen-blanket: virtualize once, run everywhere." ACM, 2012.

[7] (2016) Nested virtualization in xen. [Online]. Available: http://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," vol. 1, 2007.

[9] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010.

[10] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," vol. 41, no. 3. ACM, 2007.

[11] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency." IEEE, 2015.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers." IEEE, 2015.

[13] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments." IEEE, 2013.

[14] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, 2014.

[15] (2016) Intel container virtualization white paper. [Online]. Available: http://www.intel.com/content/www/us/en/communications/linux-containers-hypervisor-based-vms-paper.html

[16] I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian, "Memory overcommitment in the esx server," *VMware technical journal (VMTJ)*, 2013.

[17] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, 2009.

[18] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Toward energy-efficient cloud computing: Prediction, consolidation, and over-commitment," *IEEE network*, vol. 29, no. 2, 2015.

[19] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, 2002.

[20] L. Cheng, J. Rao, and F. Lau, "vscale: automatic and efficient processor scaling for smp virtual machines," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.

[21] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule processes, not vcpus," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 2013.

[22] Y. Bai, C. Xu, and Z. Li, "Task-aware based co-scheduling for virtual machine system." ACM, 2010.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," vol. 37, no. 5. ACM, 2003.

[24] A. Muller and S. Wilson, *Virtualization with VMware ESX server*. Syngress Publishing, 2005.

[25] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the linux symposium*, 2009.

[26] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, “Memory buddies: exploiting page sharing for smart colocation in virtualized data centers.” ACM, 2009.

[27] (2016) Docker inc. [Online]. Available: https://www.docker.com/

[28] (2016) Redis: In-memory key value store. [Online]. Available: http://redis.io/

[29] (2016) Mongodb:nosql database program. [Online]. Available: https://docs.mongodb.com/v3.2/

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb.” ACM, 2010, pp. 143–154.

[31] (2016) Xen credit scheduler. [Online]. Available: http://wiki.xen.org/wiki/Credit_Scheduler

[32] K. Raghavendra, “Virtual cpu scheduling techniques for kernel based virtual machine (kvm).” IEEE, 2013.

[33] (2016) cpuset kernel documentation. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt

[34] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia, “I/o scheduling model of virtual machine based on multi-core dynamic partitioning,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010.

[35] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, “vturbo: accelerating virtual machine i/o processing using designated turbo-sliced core,” in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.

[36] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu, “vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing,” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012.

[37] (2016) Redhat resource management guide. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-cpu.html

[38] A. Kopytov, “Sysbench: a system performance benchmark,” *URL: http://sysbench. sourceforge. net*, 2004.