Paracloud: Bringing Application Insight into Cloud Operations

Shripad Nadgowda, Sahil Suneja, Canturk Isci
IBM TJ Watson Research Center, NY USA

Abstract

Applications have commonly been oblivious to their cloud runtimes. This is primarily because they started their journey in IaaS clouds, running on a *guestOS* inside VMs. Then to increase performance, many *guestOS*es have been paravirtualized making them virtualization aware, so that they can bypass some of the virtualization layers, as in *virtio*. This approach still kept applications unmodified. Recently, we are witnessing a rapid adoption of containers due to their packaging benefits, high density, fast start-up and low overhead. Applications are increasingly being on-boarded to PaaS clouds in the form of *application containers* or *appc*, where they are run directly on a cloud substrate like Kubernetes or Docker Swarm. This shift in deployment practices present an opportunity to make applications aware of their cloud. In this paper, we present Paracloud framework for application containers and discuss the Paracloud interface (PaCI) for three cloud operations namely migration, auto-scaling and load-balancing.

1 Introduction

Traditionally applications were deployed on physical servers and they had certain assumptions about their platforms. For example, they will be long running on their platform with dedicated resources. It influenced application design, for instance many applications started allocating large buffer caches to optimize IO. Then as applications started on-boarding on *IaaS* clouds encapsulated in VM running on guestOS, certain properties about their platform changed. In cloud, they are routinely migrated for maintenance, load balancing, server consolidations; auto-scaled for elasticity in performance and cost. As a result, their operations and management on cloud became inefficient. For example, large buffer caches (although empty) created overhead during live migration. Since applications were oblivious to the cloud platform through abstraction of VMs, much of the invention was done to optimize VMs on this new cloud platform in the form of paravirtualized drivers for memory ballooning[1], migrations[2][3].

Recently containers started gaining acceptance as a lightweight alternative to virtual machines (VMs), owing to technology maturity and popularization by platforms like Docker[4], CoreOSs rocket[5], Cloud Foundry Warden[6]. Containers are being adopted as a foundational vitualization capability in building Platform-as-a-Service (PaaS) cloud solutions, e.g. Amazon Container service[7], Google Container Engine[8] and IBMs Container Servicebluemix. And applications are being ported on the PaaS cloud in the form of **application containers or appc** which are run directly on-top of the cloud substrate like *kubernetes*, *docker swarm* or *mesos* and are truly becoming cloud-native.

However, across these shifts of the deployment environments, what continues to prevail is the need of incorporating application awareness in the platform management operations [9, 10]. With containers, the communication gap between the application and the cloud management layer has improved further, with removal of the guestOS to facilitate a direct communication channel. Therefore, we believe it is the right time to make applications aware of the characteristics of their cloud platforms and revisit and revise their assumptions about them. Kubernetes, for example, has enabled Downward APIs[11] and Container hooks[12] to allow applications to introspect their runtime cluster lifecycle and become *cluster native*.

In Paracloud we are extending the notion of 'application knows best' in to container clouds. In this work we propose a uniform Paracloud interface (PaCI) to enable a bi-directional communication channel between application containers and the cloud management substrate. We highlight the benefits of PaCI in terms of incorporating application-awareness at the cloud management layer, as well as cloud-awareness at the applications level, with three use-cases namely migration, auto-scaling and load-balancing. We describe PaCI's design for one of the most popular container cloud platform i.e. kubernetes, and evaluate its benefits for an auto-scaling use case.

2 Background and Related Work

Taking example of the Linux kernel, since the early days, it has been positioned as a general-purpose operating system to host all *kinds* of application, taking responsibility for managing memory, scheduling, file-access for them all. But as different applications started posing different behavior and requirements, kernel allowed applications to share their anticipated file or memory access behaviour via system calls such as *fadvise* or *madvise* as

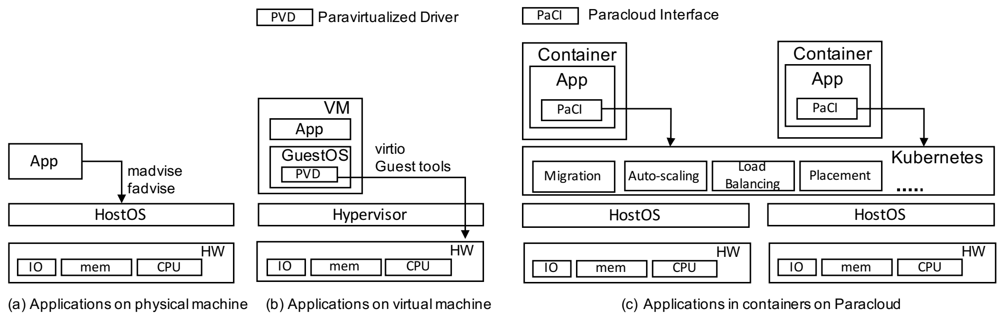


Figure 1: Application runtime evolution

shown in Fig. 1(a). Using these hints to choose the appropriate read-ahead and caching techniques enabled kernel to improve application performance and avoid redundant operations. The benefits of such hints have been reported for several applications [13, 14, 15, 16, 17, 18] and distributed filesystems [19, 20, 21]. Java runtime on the other hand has taken an approach to manage heap memory by itself in JVM to insulate from host machine's peculiarities and also because it can manage memory better by identifying and garbage collecting orphaned objects.

Applications today are increasingly being hosted directly on top of container clouds like kubernetes, docker swarm and mesos in the form of application containers. Like their traditional counterparts, these clouds are becoming a general-purpose operating platform to host various *kinds* of applications (containers). However, they have to additionally deal with the management concerns similar to those of VM clouds. The need for application awareness has long been recognized in literature, and the fact that an application is deployed in cloud makes it ever more important [22, 23, 10, 24, 25].

Moving over to higher level management tasks, benefits of application assistance has been explored in both physical and virtual systems, in the context of fault-tolerance [9, 26], physical memory management [27], checkpointing [9], migration [28], QoS adherence [29, 22], memory overcommitment [30, 31], load balancing [32] and scaling [24, 33, 34]. Others have also explored moving management functionailites into the application itself [4]. Alternatives to using applications level knowledge for management tasks have also been explored via statistical learning and prediction-based approaches [35, 36, 37].

In the past, different techiques have been employed to extract application level knowledge for aiding management operation. This includes installing paravirtualization drivers [28], in-guest controller agents [26, 38, 39, 34] (both referred to as PVD in Fig. 1(b)), leveraging or extending language runtime capabilities like apache modules or JDBC interfaces [30], or modifying the application directly [40].

With containers clouds, the principle of 'application knows best' does not change. What does change is the nature of the interface between applications and the hosting layer, which becomes closer to the traditional systems with the removal of the guestOS abstraction of VMs. Also, the scale of management operations grows, owing to an even greater flexibility and elasiticity afforded by the container abstraction.

Thus, with a similar intention of incorporating application-awareness at the cloud management layer, as well as cloud-awareness at the application level, the goal of our work is to provide a uniform interface (PaCI in Fig. 1(c)) to enable such bi-directional communication. One way to realize this channel for container-based OS virtualization is via a generic signal-and-syscall (or ioctl) based implementation. However, in this work we present an implementation for a higher-level abstraction layer for easier consumability, by targeting the popular Kubernetes container platform.

3 Paracloud Orchestration Usecases

Amongst various cloud management tasks, here we specifically target three operations - namely *migration, auto-scaling and load-balancing*, to highlight how our Paracloud interface can help make them more efficient for both the application and the cloud management layer.

3.1 Migration

Although migration may seem redundant for stateless application containers, it is still pertinent to several stateful microservice applications like databases (e.g., Mysql, Cassandra), message brokers (kafka), and coordination services (zookeeper), amongst others. This is being acknowledged and supported in standard frameworks like Kubernetes' 'StatefulSet'. Portability of stateful containers is also explored in existing solutions like ClusterHQ's Flocker[41], Virtuozzo[42] and Picocenter[43]. At the same time, it has certain inefficiencies that we would like to address below.

During initialization, many applications, like databases, tend to allocate large memory buffers to optimize their IO operations. Similarly, application runtimes, like JVM, allocate large operating heap memory which they manage themselves such as via custom garbage collection policies. This becomes a problem when such application containers are to be migrated [1]. In stop-and-copy based migration, this increases application downtime by also migrating non-dirty or unused, speculatively cached or to-be-garbage-collected pages. Although the downtime is lower for pre-copy or post-copy migration, unnecessary page transfer processing on an already troubled host (that influenced application offloading in the first place), possibly also coupled with a congested network outflow, can still slow down the readiness of a migrated application.

Although it may be possible to add certain memory filtration heuristics to the migration process, but the application can do a far better job of minimizing its state for checkpointing or transfer. Therefore, in Paracloud we propose signaling an application container to be migrated, and allowing a short migration grace-period for 'preparation'. This allows application to minimize its memory footprint, for example by running its garbage collector to release heap-memory, flushing its IO buffers, releasing all temporary resources like temp files. It can even exercise aggressive eviction on its cache by flushing less common objects, or optionally settle at a consistent state pre-transfer. Similarly, on restoration at the target, the application container is signaled to re-configure itself to the new environment. During restore grace-period, application can perform sanity checks, re-acquire its memory share for caching, as well as any temporary or lost resources like network connections, register its service, and re-discover other services.

3.2 Auto-scaling

It is one of the core capability for any cloud platform to enable elasticity for their workloads. Policy-based scaling techniques, which are common in today's clouds, scale out instances when certain resource-use metrics grow beyond a particular threshold. For instance, a sample policy could be to increase container instances when the average memory utilization of the container is greater than 70% for a duration of 1 minute. We argue that such externally monitored metrics may not be *true* indicators of demand to derive auto-scaling policies. For example, most database applications(e.g. Mysql, ElasticSearch) tend to perform various auxiliary functions besides storing and accessing data, like periodic log rotation, data compaction, data pruning, auditing and consistency checks etc. These functions are commonly designed to *exploit slack*. They consume resources over the actual workload processing when there are free cycles, and can cause false-positive auto-scaling triggers. Another issue with black-box triggers is that several modern applications are designed to respond to their operating environment constraints. For example, ElasticSearch (ES) aims to keep as much of its index in memory. Therefore, a memory-based auto-scaling trigger, oblivious to this behavior, can start scaling number of ES instances once memory grows beyond a certain threshold, as the new instances start to grow their memory footprint, it continues to further scale up the instances, resulting in a chain reaction.

In contrast, Paracloud provides a way for an application to indicate whether it really desires new instances to be forked, given its current operational state. The auto-scaler on sensing an increased usage beyond the set threshold, can signal the container indicating an upcoming scaling operation, allowing it to optionally dictate its intention. The application can validate the trigger against its actual state, and try to minimize usage of some resources if its operational state allows, for example by running its garbage collector or flushing some caches and buffers. This gives cloud users a trade-off knob, where they can intelligently balance performance and operational cost.

Another use of Paracloud interface is for the application to itself proactively hint the auto-scaler for a more prompt service, based on its key performance indicators. Furthermore, Paracloud also enables an application container to specify the *type* of sibling instances it needs. Auto-scaling typically forks a brand-new instance which then has to be initialized, configured and cache-warmed, thereby delaying the achievement of steady-state performance for the application as a whole. Instead, it may be advantageous to hot-scale the instance with a pre-initialized state and a warm cache [45].

3.3 Load Balancing

A cloud host's resources (CPU, memory, IO) are usually over-committed to save costs by increasing density (i.e. number of colocated instances existing at any given time). During high resource contention, some of the instances (containers) are either killed or migrated. These may be selected based upon their priorities or current resource allocations. For example, one common practice is to kill containers consuming most resources. But intuitively, those could be amongst the most active and/or stateful containers on the host, which could have been kept running by killing other less-active and/or stateless applications' instances (a stateful application gets hurt worse, having to re-warm its cache on reinstanitaition [45]).

Such scenarios can be handled better by tying Paracloud with cloud load balancers. Cloud applications are typically deployed as "replica sets", with multiple identical instances available at any given time, which are load-balanced by an ingress controller like haproxy [46]. With Paracloud, we propose to expose an *appYield* interface

[1]Several container runtimes today support migration; Docker, for example, achieves this via the popular CRIU [44] checkpoint-restore Linux utility.

between containers and the cloud platform, enabling applications to yield themselves temporarily during high resource contentions, similar to Linux' *sched_yield*-based co-operative scheduling [47]. For a host under pressure, its containers get notified with a yield request notification. Such notification would contain information about the resource under contention, and the health status of the container's replica/peer instances. An application can then decide to yield, releasing the contended resource, or risk termination. The load-balancer would temporarily tag the instance out-of-service, bringing it back into action once the resource contention smoothes out. Essentially, the application stays alive but dormant. Similarly, the application can also advertise its statefulness over Paracloud so that the host can accordingly target a 'less critical' application if possible. It is important to acknowledge that such co-operative scheduling is helpful only when the resource contention is transient, and work better with the existence of some 'incentive' schemes for applications to yield, than perhaps with the autocratic *yield-or-die* regime.

4 System Design

We design Paracloud interfaces to ensure consumability and simplicity. Therefore PaCIs are implement on the *Kubernetes* container platform. Kubernetes advocates building applications that are cluster-aware and has basic machinery in place in the form of *Container Lifecycle Hooks* [12] to inform containers about management lifecycle events. Currently two container hooks are facilitated, *PostStart* and *PreStop*. PostStart hook is called immediately after container is created and PreStop hook is called before it is terminated. These hooks are captured and processed inside container in *hook handlers*. Kubernetes supports two handler types, *Exec* to execute a command or script in container namespace, and *HTTP* to invoke a specified http endpoint of container. Application containers that participate in cluster lifecycle are termed as *cluster-native*.

In Paracloud we motivate the extension of the scope of *cluster-native* applications beyond just lifecycle events to critical cloud operations like auto-scaling, migration and load-balancing. We have currently designed six new interfaces for Kubernetes substrate as summarized in Table 1. We envision addition of new interfaces in support of other Paracloud operations like placement, compliance, security and monitoring.

4.1 PaCI Definitions

PaCIs are completely optional for applications. Therefore hook handler invocations are not retried on cloud platform and all PaCIs are designed as asynchronous so they do not block any cloud operation.

PaCI	Delivery guarantees
preMigrate↑	at-most-once
postRestore↑	at-most-once
reqYield↑	at-least-once
appYield↓	-
chkAutoscale↑	at-least-once
hotScale↓	-

Table 1: Paracloud Interfaces; ↑ = interface implementation as a container *hook*; ↓ = extension to Kubernetes API set.

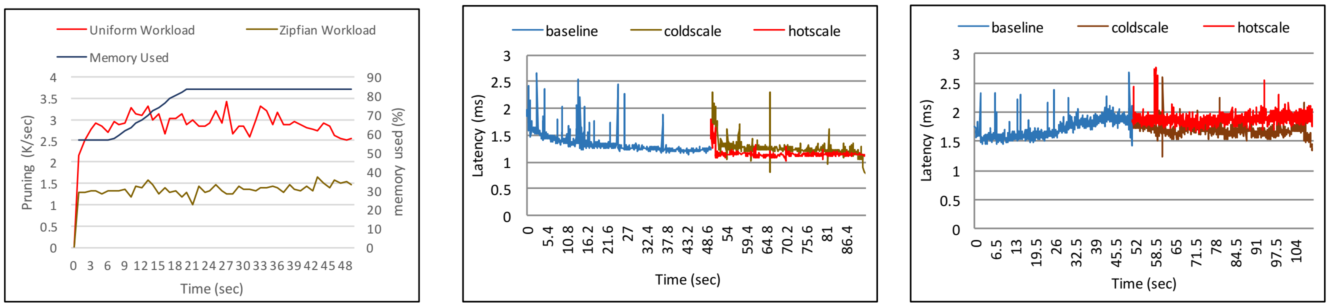
Migration: The *preMigrate* hook is sent after container is scheduled for migration but before it is checkpointed. *postRestore* hook is sent after container is restored to running state and before it is reported as available. Both these hooks follow *at-most-once* delivery guarantees since the hook handler might not be idempotent. Applications can have different grace-period requirements to process hook handlers. Therefore, we allow applications to configure these grace-periods as container labels.

Load-balancing: When a compute node is heavily loaded, cloud platform invokes *reqYield* hook on all containers on that node. It follows *at-least-once* guarantee since a hook might be called multiple times during the lifetime of application container. The hook handler in response evaluates yield chances and calls *appYield* API on Kubernetes if it decides to yield reseources. Multiple *appYield* calls are processed in the same order they are received. When the node load drops to an acceptable level, the remaining yield calls are ignored. Containers for which yield call is accepted are temporarily removed from load-balancer.

Auto-scaling: For every auto-scale triggered on the platform, the *chkAutoscale* hook is called on the corresponding containers to validate the scaling request. This gives the applications a chance to mitigate the auto-scaling conditions in the case of false positives. In the handler, containers can release the heavily-used resources to clear the trigger. For example, when high memory usage triggers an auto-scale, the application can employ garbage collection or flush caches to reduce its footprint. This hook also allows a configurable grace-period for handler processing and mitigation actions. If the auto-scale condition still holds true after grace-period, application scaling is implemented by the platform. If the application wants to hot scale by creating a clone of one of its running instance, it calls *hotScale* API on Kubernetes.

4.2 PaCI Sidecar Implementation

In Paracloud we implement hook handlers and the Kubernetes client for PaCI calls in *sidecar containers*. Sidecar containers are a common microservice composite container pattern for providing supporting services, such



(a) Cache Pruning and Memory Monitor (b) Zipfian access workload (c) Uniform access workload

Figure 2: Paracloud Autoscaling Usecase

as service discovery and logging, to main application containers by creating new containers that share namespaces with them. The important benefit of this pattern is that the sidecar containers can be developed independently without changing applications. Most PaCIs actions, like garbage collection, cache flush, post-restore service registration, evaluating auto-scale criteria can be implemented in sidecar containers with minimum disruption and high consumability by most applications.

5 Evaluation

In this paper we focus on the auto-scaling use-case for Paracloud. As described in the previous section, Paracloud uses the *chkAutoscale* hook to inform the container of an autoscale trigger and the *hotScale* API to actuate a hot scale if requested by the container. Traditionally, auto-scaling uses *cold scaling*, wherein new container instances are created from a pristine image. In *hot scale* new instances are created by live-cloning the running state of the container. In the below experimental evaluation we show how an application can optimize its performance by judiciously employing hot scaling when advantageous.

We use `MySQL 5.7.15` from Docker Hub as our test application and YCSB [48] as the benchmark. The container is configured with 512MB memory. The workload consists of two runs of 400K read operations against a 200MB database table consisting of 50K records of 4KB size each. The Kubernetes cluster triggers an autoscale when container used memory $>80\%$ for more than 30 seconds. We use `Haproxy` to arbitrate requests among multiple instances using the round-robin policy. We also enable query cache for MySQL engine and monitor its rate of low memory pruning.

We run two sets of experiments with different access patterns, $zipfian$ for popularity-based long tail access and $uniform$ for random access pattern. The application exhibits similar memory usage in both experiments, as shown in Fig. 2(a). This usage pattern triggers an autoscale at around 50s, which is communicated to the container via *chkAutoscale*. In each experiment the application can respond in one of three ways: (i) do nothing, which triggers a cold autoscale; (ii) take mitigating actions, such as releasing memory resources, to revert the autoscale trigger; and (iii) request a *hotScale*. In our evaluation we exercise the first and third options. This shows that requesting a *hotScale* for the $zipfian$ access leads to significant performance improvements, while employing cold scaling performs better for $uniform$. Fig. 2(b) shows the results for $zipfian$. Container-triggered hot scaling helps reduce the latency of application by $\tilde{2}0\%$ immediately. In comparison, cold scaling requires a ramp-up time for cache warming to attain the same performance. 2(c) shows the contrasting results for $uniform$. In this case, the container does not initiate a *hotScale*. As a result cold scaling is applied, which improves performance by reducing latency by 20%. The figure also shows the latency if hot scaling were applied, where it actually deteriorates performance. These results demonstrate the substantial advantages with Paracloud providing application insight in cloud operations.

While the advantage of hot scaling in $zipfian$ is expected, its negative effect for $uniform$ is interesting. This is because with with hot scaling applications can inherit bad state as well as good state. Only judicious application of scaling techniques, driven by the applications themselves, can lead to net benefits. In the $uniform$ case, the hot scaled container inherits the thrashing state of the MySQL query cache which dampens the initial scale-out gain. This can be observed from the high pruning rate of $uniform$ in Fig. 2(a). Cold scaling in such scenarios performs fundamentally better since it removes cache pruning overheads and only incurs cold cache miss latencies. Thus, the containerized application can simply use this additional pruning rate signal to decide on when to request *hotScale*, while the container runtime is oblivious to these characteristics.

6 Conclusion

We presented our Paracloud idea for container clouds, wherein application insights are brought into the cloud platform thorough a generic Paracloud interface (PaCI). The critical inference here is that applications have intimate knowledge of their own operating state parameters, which are typically different for every application. The cloud platform cannot gauge these parameters reliably for all applications. Paracloud can help bridge this gap and provide a shared-responsibility model, where all the cloud functions like auto-scaling, migration, load-balancing, etc. are implemented by the platform with application insight and cooperation through PaCIs. We are planning to extend the scope of PaCI to cover many more cloud operations and establish them as a standard design choice during application development.

7 Discussion

First, we would like to establish that Paracloud framework is beneficial for both application developers and cloud providers. For developers there's motivation to make their applications more agile, cost-efficient, consistent and highly elastic on clouds, while for providers it helps ease their operational overheads for different *kinds* of applications and satisfy SLAs.

How disruptive is this model ? As illustrated above for auto-scaling use case, few common PaCIs can be implemented as a sidecar containers for applications. But, some PaCIs requires support from the applications with revision in their design and implementation, for example, to release resources during migration, acquiring resources for vertical auto-scaling, identifying and book-keeping of relevant operational states etc. But, we believe this is an incremental feature addition as-oppose to the complete application re-modeling imposed by event-driven serverless architectures or single address space unikernels.

How secure are these interfaces? Although not limited by design, currently we are regulating applicability of Paracloud to a single tenant platform wherein different hosted applications have trust amongst their peers. But, at the same time we are doing security profiling of PaCIs by evaluating their security implications and any possible exploitation in multi-tenant environments, so they can be hardened.

Open Issues: (i) Should PaCI be vendor-agnostic, perhaps via a signal-and-syscall implementation? (ii) Better incentives than yield-or-die may be needed for less crude, escalation inhibitive, and more harmonious existence in the load balancing scenario. (iii) Whether PaCIs are applicable for non-containerized deployments ?

References

[1] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[2] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60. ACM, 2009.

[3] Kai-Yuan Hou, Kang G Shin, and Jan-Lung Sung. Application-assisted live migration of virtual machines with java applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 15. ACM, 2015.

[4] Docker. `http://www.docker.com/`.

[5] CoreOS. `https://coreos.com/rkt/`.

[6] CloudFoundry. `https://github.com/cloudfoundry/warden`.

[7] Amazon EC2 Container Service. `https://aws.amazon.com/ecs/`.

[8] Container Engine. `https://cloud.google.com/container-engine/`.

[9] Yennun Huang and Chandra Kintala. Software implemented fault tolerance: Technologies and experience. In *FTCS*, volume 23, pages 2–9. IEEE COMPUTER SOCIETY PRESS, 1993.

[10] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 2016.

[11] Kubernetes. Downward api. `https://kubernetes.io/docs/user-guide/downward-api/`.

[12] Kubernetes. Container lifecycle hooks. `https://kubernetes.io/docs/user-guide/container-environment/`.

[13] Michael McCandless . Lucene and fadvise/madvise. `http://blog.mikemccandless.com/2010/06/lucene-and-fadvisemadvise.html`.

[14] Greg Bowyer. Solr / Lucene madvise performance ? `http://people.apache.org/~gbowyer/madvise-perf/`.

[15] google-perftools@googlegroups.com. Google Performance Tools system_alloc in MoongoDB. `https://github.com/mongodb/mongo/tree/master/src/third_party/gperftools-2.5`.

[16] Don Burleson. Using direct I/O with Oracle. `http://www.dba-oracle.com/art_orafaq_oracle_direct_io.htm`.

[17] MySQL. Optimizing InnoDB Disk I/O. `https://dev.mysql.com/doc/refman/5.5/en/optimizing-innodb-diskio.html`.

[18] Tobias Oetiker. Tuning RRDtool for performance. `http://oss.oetiker.ch/rrdtool-trac/wiki/TuningRRD`.

[19] Hadoop HDFS. Add HDFS support for fadvise readahead and drop-behind. `https://issues.apache.org/jira/browse/HDFS-2465`.

[20] NFSv4 Working Group: D. Hildebrand, T. Myklebust, S. Falkner. Support for posix_fadvise. `http://www.potaroo.net/ietf/idref/draft-hildebrand-nfsv4-fadvise/`.

[21] James Coomer. Lustre File System Acceleration Using Server or Storage-Side Caching. http://www.opensfs.org/wp-content/uploads/2014/04/D2_S27_LustreFileSystemAccelerationUsingServerorStorageSide-Caching.pdf.

[22] Khalid Alhamazani, Rajiv Ranjan, Fethi Rabhi, Lizhe Wang, and Karan Mitra. Cloud monitoring for optimizing the qos of hosted applications. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 765–770. IEEE, 2012.

[23] Qinghua Lu, Xiwei Xu, Liming Zhu, Len Bass, Zhanwen Li, Sherif Sakr, Paul L Bannerman, and Anna Liu. Incorporating uncertainty into in-cloud application deployment decisions for availability. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 454–461. IEEE, 2013.

[24] J Yang, T Yu, LR Jian, J Qiu, and Y Li. An extreme automation framework for scaling cloud applications. *IBM Journal of Research and Development*, 55(6):8–1, 2011.

[25] Hanieh Alipour, Yan Liu, and Abdelwahab Hamou-Lhadj. Analyzing auto-scaling issues in cloud environments. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 75–89. IBM Corp., 2014.

[26] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[27] Sitaram Iyer, Juan Navarro, and Peter Druschel. Application-assisted physical memory management. Technical report, Citeseer, 2004.

[28] Kai-Yuan Hou, Kang G Shin, and Jan-Lung Sung. Application-assisted live migration of virtual machines with java applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 15. ACM, 2015.

[29] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.

[30] Michael R Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 130–137. IEEE, 2011.

[31] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 630–637. IEEE, 2009.

[32] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 231–242. ACM, 2013.

[33] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[34] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, 2014.

[35] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. Ieee, 2010.

[36] Alexandru-Florian Antonescu and Torsten Braun. Improving management of distributed services using correlations and predictions in sla-driven cloud computing systems. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014.

[37] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[38] Zhikui Wang, Yuan Chen, Daniel Gmach, Sharad Singhal, Brian J Watson, Wilson Rivera, Xiaoyun Zhu, and Chris D Hyser. Appraise: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4), 2009.

[39] Gregory Katsaros, George Kousiouris, Spyridon V Gogouvitis, Dimosthenis Kyriazis, Andreas Menychtas, and Theodora Varvarigou. A self-adaptive hierarchical monitoring mechanism for clouds. *Journal of Systems and Software*, 85(5):1029–1041, 2012.

[40] José E Moreira and Vijay K Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303–330, 1997.

[41] ClusterHQ. Flocker. `https://docs.clusterhq.com/en/1.0.3/`.

[42] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, pages 85–92, 2008.

[43] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *EuroSys'16*, page 37. ACM, 2016.

[44] CRIU. `www.criu.org/`.

[45] Shripad Nadgowda, Sahil Suneja, and Ali Kanso. Comparing scaling methods for linux containers. In *To appear in the IEEE Third International Workshop on Container Technologies and Container Clouds (WoC)*, 2017.

[46] HAProxy. The Reliable, High Performance TCP/HTTP Load Balancer. `http://www.haproxy.org/`.

[47] Linux Programmer's Manual. sched_yield - yield the processor. `http://man7.org/linux/man-pages/man2/sched_yield.2.html`.

[48] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.