2017 IEEE International Conference on Cloud Engineering

Comparing Scaling Methods for Linux Containers

Shripad Nadgowda, Sahil Suneja, Ali Kanso
IBM T.J. Watson Research, NY, USA
{nadgowda, suneja, akanso}@us.ibm.com

***Abstract*—Linux containers are shaping the new era of building applications. With their low resource utilization overhead and lightweight images, they present an appealing model to package and run applications. The faster boot time of containers compared to virtual/physical machines makes them ideal for auto-scaling and on-demand provisioning. Several methods can be used to spawn new containers. In this paper we compare three different methods in terms of start time, resource consumption and post-start performance. We discuss the applicability, advantages and shortcomings of each method, and conclude with our recommendations.**

Keywords—Linux containers, checkpoint and restore, live migration, performance analysis.

I. INTRODUCTION

Linux containers have emerged as a technology with a multitude of use cases that span from sharing the infrastructure resources [47], to implementing platform-as-a-service [48], to simply using them to manage applications that are deployed in virtual machines (VMs) [49]. In this study, we focus on containers as a means to conveniently package and port applications to different platforms

In an elastic deployment model, where demand drives the number of instances an application may have, auto-scaling application instances runs into a *time-to-scale* (TTS) issue. With horizontal scaling[1], a long TTS, such as with VMs, either causes a period of suboptimal performance while waiting for new instances to become ready, or demands predictive measures to spawn new instances ahead of time. Containers have a much smaller start-time, and therefore are a better fit for on-demand application scaling with a significantly reduced TTS.

However, the start time does not necessarily mean that the application's process(es) within the container is (are) ready. In this study, we distinguish between the container start-time and its application readiness latency. We use start-time to refer to the time needed to get a container up and running, and readiness latency to also include the time it takes for the application within the container to load and initialize before it is ready to provide its intended functionality. For example, an in-memory database[2] would first have to read configuration data as well as its internal tables from disk, load them into memory, and carry out its initialization steps, before it is ready to serve any queries. In such cases, as we will show later, the readiness latency can be significantly higher than the start time.

Thus, even with low-TTS instantiations, performance scaling may lag. If, however, a new instance can be created while avoiding the application initialization costs, it instantly becomes available for service. Furthermore, if it gets access to a warm runtime cache state from the outset, without having to go through a cache warming phase, it may be able to provide peak service instantaneously, resulting in a truly low TTS.

To validate these conjectures, in this paper we compare three methods of scaling containers:

- **Cold-scale**: using the traditional method of starting containers from an original image. The application process(es) inside the container will be instantiated after the container set up.
- **Warm-scale**: using a pre-processing step in which a container's application is allowed to initialize, and then checkpointed into what we call as a *warm-image*. Any new instance of the corresponding application is then created from this warm-image.
- **Hot-scale**: based on live-cloning of a running container in order to create another instance of the same type. The newly created instance will have the exact state of the already running instance, including its caches, except it will have a new identity (instance name, network configuration, etc.)

In our experiments we observed that warm-scaled containers improve the readiness of an application, with minimum storage overhead. Hot-scaled containers better readiness along with application throughput, although they could impose performance implications at the source instance and a significant storage overhead. We also observed that the hot-scale method has worst performance if the container is already memory-thrashing. All these performance tradeoffs and overheads need to be duly considered for selecting the right scaling method.

This paper is organized as follows: in Section II we present a brief background on containers, scaling and checkpointing. In Section III we present the technical and semantic differences between our scaling methods. In Section IV we present the results of our experiment-based comparison of the scaling methods. We survey the literature for related work in Section V. Finally, we conclude and discuss our future work in Section VI.

[1] In horizontal scaling the number of instance is changed as opposed to vertical scaling where the resources allocated to a given instance are changed.

[2] For simplicity, we use the terms database and database management system (DBMS) interchangeably

978-1-5090-5817-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IC2E.2017.42

II. BACKGROUND

A. Linux Containers

Linux containers are perceived as a lightweight virtualization technology, and are often compared to virtual mahcines. In fact, containers do not emulate the physical resources. Instead, containers act like process level wrappers that isolate the application and account for its resource comsumption. Containers leverage the Linux namespaces [50] for isolation (e.g., network, process ID, mount points, etc.) and control groups (*cgroups*) [51] to limit and account for the resources (such as CPU, memory and disk I/O) consumed by its processes. A container is spawned based on an image. An image is an immutable filesystem and a set of parameters to use when instantiating the conainer at runtime. Such image may include the executable files of the process(es) (in the container's root file system once started). It can also point to volumes that are mounted to the container when it is instantated. There exist several container runtimes tor managing a container's life-cycle (creation, update and deletion) such as LCX, Docker, rkt and OpenVZ amongst others [52]. In this study, we use the Open Container Initiative specification-compliant runC [53] implementation of the popular Docker container runtime.

B. Horizontal Scaling

Horizontal scaling is a response to workload variation by scaling in/out the number of instances of a given application [54]. The process of scaling is significantly simplified with the use of virtual machines and containers, since the application and its environment is captured in an image that is easily restored. Automated scaling (auto-scaling) features are typically expressed by rules and policies defined by the application's owner. The monitoring features of a system will detect the violation of those rules and trigger an auto-scaling event. Auto-scaling typically involves adding/removing new instances as well as auto-configuring the load-balancer to reflect those changes. In this study, we achieve 'warm' and 'hot' container scaling, as introduced in Section I, via (container's) process checkpoint and restore as described next.

C. Checkpoint and Restore

Amongst the various existing process checkpoint and restore techniques in Linux [55], we use a popular user-mode[3] implementation in CRIU (Checkpoint and Restore in User-Space) [56]. CRIU enables checkpointing a running process as a collection of files (pagemap, open files, open sockets, etc.), which can later be used to restore and resume the process' operations from the point in time it was frozen at. The checkpoint procedure relies heavily on */proc* file system where CRIU fetches the information it needs concerning files descriptors, pipes parameters, memory maps etc. For the restore step, CRIU reads the dump files generated in the checkpoint phase, forks the process, and then restores its resources (memory, sockets, credentials, timers, etc.).

We use CRIU with Docker-runc to enable checkpointing a container's processes, which then forms the basis for warm-scale and hot-scale provisioning, described in detail in Section III. We use the base stop-and-checkpoint approach in our experiments. Other flavors supported by CRIU include incremental, iterative, and post-copy checkpointing.

III. CONTAINER SCALING METHODS

This Section illustrates how we orchestrate a new instance initialization based on three different scaling methods. We use the Docker container runtime and CRIU-based container checkpointing.

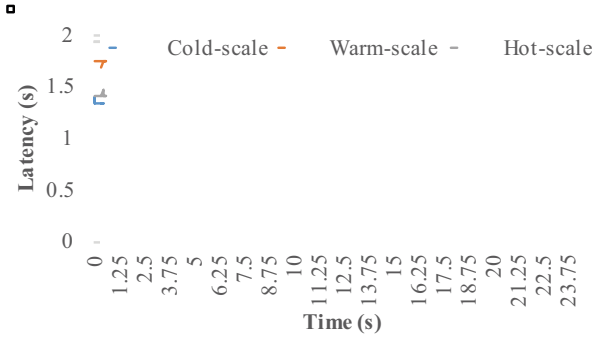
i) **Cold-scale**: In this method, we instantiate an application container from its base image using the default *Docker run* command. Although the first execution will fetch several base image layers from the Docker hub [57], in our experiments we assume the base image exists on the target node. The application initialization latency is then the amount of time it takes for the application to get ready following its container instantiation.

ii) **Warm-scale**: In this method, as a pre-processing step, we first start an application container as in Cold-scale. Once it is initialized and the application is ready, we use CRIU to take a memory checkpoint of the container. CRIU persists this checkpoint in a set of files together comprising a *mem-dump*. Any filesystem changes made by the application itself during initialization (e.g. creation of database init files, configuration file changes, etc.) are persisted in the container's *rootfs* directory. Thus, for consistency, we also capture the container's *rootfs* state, quiesced right after the checkpoint. This, together with the *mem-dump* collectively constitutes a warm-image[4], which is used to provision a new application container instance using CRIU-restore. This then allows a new instance in Warm-scale to avoid the application initialization latency.

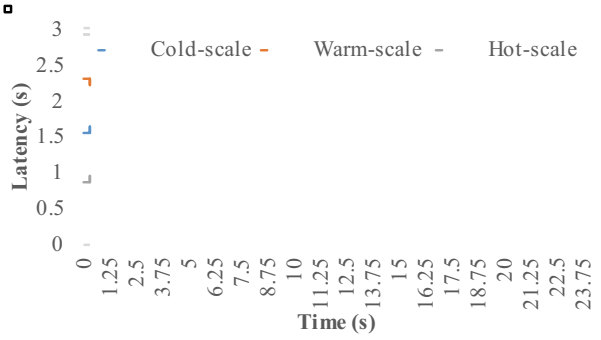
iii) **Hot-scale**: Instead of booting from a post-init checkpoint as in Warm-scale, a new instance in Hot-scale is created on-demand as a live clone of an already running container (the *source*). With hot-scale, the new instance also gets access to the runtime caches of the source container, which may help it attain peak performance quickly (see Section IV-3). In this mode, we use CRIU's remote page-server model to generate a *mem-dump* of the source container directly at the target host's memory (using *tmpfs* as an in-memory mounted file system). The *rootfs* disk state would also need to be checkpointed for access by the new instance, after which the source container can be resumed. In our experiments, we used a data federation framework from Cargo[45] to enable consistent *rootfs* access for both container instances (which is sufficient due to the read-only nature of our test workload, i.e. the instances are not modifying the state on disk). The *rootfs* together with the *mem-dump* then constitutes the hot-image, which is what the new instance gets provisioned with using CRIU-restore.

[3] The changes needed in the Linux kernel to support kernel-mode CR where not included in upstream Linux. Therefore, we opted not to use them.

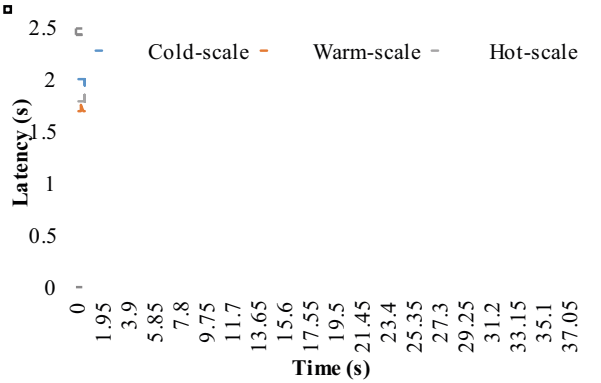
[4] Alternatively, a data cache could also be incorporated as part of the warm-image, after an anticipatory identification and preloading of important data.



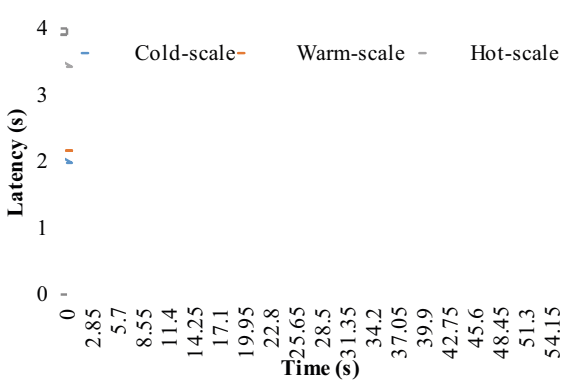
(a) Memory-resident Zipfian distribution



(b) Memory-resident Uniform distribution



(c) Memory-thrashing Zipfian distribution



(d) Memory-thrashing Uniform distribution

Figure 1. Runtime performance comparison for Cold-scale, Warm-scale and Hot-scale methods

IV. Experiments and Comparison

We compare the three scaling methods along the following three dimensions:

(i) *Instance size*- total size of the base image including both disk and memory states

(ii) *Readiness latency*- total time to get a new application instance up and running

(iii) *Post-init performance*- runtime performance of a newly created application instance

We used MySQL and ElasticSearch as our test applications, due to similar performance tradeoffs for the different scaling methods for both, and the lack of space we present results for only the former here onwards.

A. Testbed Setup

We conducted our experiments on Ubuntu 16.04 LTS machines, each configured with 4 CPUs, 4 GB memory and 25GB disk with ext4 filesystem. The machines were configured with the Docker container runtime v1.12.5, and CRIU v2.8 for checkpoint-restore. As one of our test applications, we selected one of the most popular database applications from Docker hub- MySQL 5.7.15. We used the standard Yahoo Cloud Serving Benchmark (YCSB)[46] benchmarking tool to measure MySQL performance under different workload patterns.

The test workload consisted of 400K database read operations, generated using two request distributions, namely (i) *Zipfian*- for popularity-based long tail access, and (ii) *Uniform*- for uniformly random access pattern. The workload was run against a 200MB database table size, consisting of 50K records, each 4K in size. To evaluate the behavior of scaling methods under different operating states of an application, we explicitly enabled the *query cache* configuration for the MySQL engine. We defined two extreme operating states for an application namely (i) *memory-resident*- when all workload data fits in memory (*query cache = 256MB*), and (ii) *memory-thrashing* - when only part of the data can reside in memory at any given point (*query cache = 128MB*). For each workload-type, we recorded the average latency and throughput observed by the YCSB client. Each experiment was performed atleast 3 times to ensure consistency in the results.

B. Results

We present the comparison results here, and discuss our observations on these results later in section IV-C, while considering all performance implications together.

1. *Instance Size*

We define instance size to refer to the base image size for the application container. For cold-scale instantiation, this refers to the regular base image hosted and pulled from the Docker hub, which for MySQL is around 400MB. For warm-scale booting, the size of the warm-image increases to 520MB. And for hot-scale live cloning, the size of the hot-image was recorded as 920MB for *memory-resident* application configuration, and 790MB for the *memory-thrashing* state (the difference is consistent with the cache size configuration: 256 versus 128 MB).

2. *Readiness Latency*

We define readiness latency as the time from container instantiation until an application is ready to provide its service, after having completed its initialization operations. We measured the readiness latency in both cases −when the instance image resided on disk, or in memory (*tmpfs*). For cold-scale instantiation, the average readiness latency for MySQL was measured to be 7.5 seconds. The image location (on disk or in-memory) had a negligible impact. For warm-scale provisioning, the application readiness latency is the same as the container restore time, since the application is restored to its initialized state. This was measured to be 300ms and 400ms on average, for in-memory and on-disk hosting respectively. The readiness latency for the hot-scale method includes both −the checkpoint time at the source host, and the restore-time at target host. As shown in Table I. it ranges between 900ms to ~1.3s for the different combinations of image hosting locations (network bandwidth was ensured not to be a bottleneck)

TABLE I. CHECKPOINT-RESTORE TIMES FOR HOT-SCALING

	Local on-disk	Local in-memory	Remote on-disk	Remote in-memory
Checkpoint	0.58s	0.45s	0.91s	0.9s
Restore	0.7s	0.47s	-	-

3. *Post-init Performance*

Once initialized and ready for service, we ran the YCSB test workloads, as described in Section IV-A, against our database application. Figures 1 (a) - (d) show the performance of the newly instantiated MySQL container, for the different operating states and access patterns. The plots show time series of the response latency for incoming requests (the relative trends for throughput curves are similar to the latency plots, and are thus omitted for brevity). The data points also include the readiness latency, most easily observed for the cold-scale method by its *null* performance for the first few seconds.

The first general observation we make is that the runtime performance for cold-scale and warm-scale methods is similar for all workloads and operating states, except for the higher readiness latency for the former.

The benefits of hot-scale provisioning can be seen in Figures 1(a) and 1(b), where for a *memory-resident* application state, a newly created instance can immediately start serving requests with low latencies. This is due to its already-warm cache at instantiation, a hot-scale instance does not have to go through a cache warming phase that causes higher service latency for its competitors. Furthermore, the total execution time for the YCSB run is also lower for hot-scale instance-14.4s vs. 18.1s, indicating a 25% higher throughput. In the case of *memory-resident* setting, similar results were observed for both zipfian and uniform access distributions.

The *memory-resident* setting is only one facet of hot-scale provisioning. When an application operates under a *memory-thrashing* state, we observed that it performs worse than its competitors as shown in the Figures 1(c) and 1(d). The initial service latency is higher for cold-scale and warm-scale provisioned containers during the cache warming phase (between 0-2s). Then, relatively low latencies are recorded until the cache becomes full (between 2-10s). Thereafter, increasingly high latencies are noticed due to memory thrashing. On the other hand, for hot-scaling, this thrashing state gets captured in its *mem-dump*, and the new instance directly gets restored with such memory thrashing. As a result, we notice high latencies from the beginning of its instantiation. The total execution time and throughput for the YCSB run against this hot-scale instance is worse by 46% for Zipfian and 64% for uniform access pattern, as compared to the other scaling techniques. To further substantiate the thrashing overhead, we collected the actual *query-cache* stats from the MySQL engine.

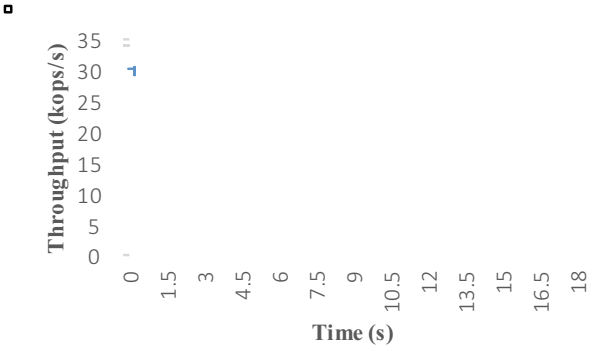
TABLE II. MEMORY THRASHING OVERHEAD

query-cache stats	**Zipfian**		**Uniform**	
	Cold-scale	**Hot-scale**	**Cold-scale**	**Hot-scale**
Cache inserts	137K	143K	219K	231K
Cache hits	262K	256K	180K	168K
Cache prunes	114K	**145K**	197K	**232K**

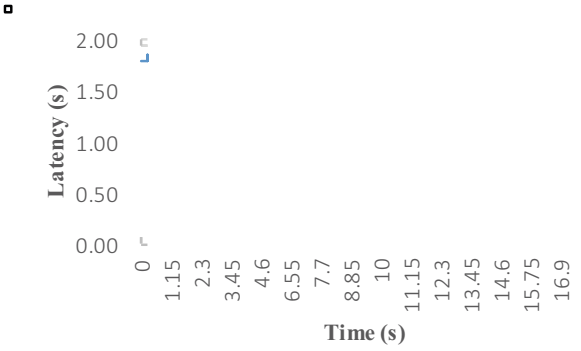
As shown in Table II, more cache pruning occurs for the hot-scale method in a *memory-thrashing* application state, which deteriorates runtime performance.

3.1 *Impact on Source*

Unlike, cold-scale and warm-scale approaches, an instance created via hot-scaling is dependent on its source container's state at the time of its instantiation. The source application may observe operational disruption during a hot-scale instantiation, with the performance impact depending upon the cloning efficiency. During checkpointing, CRIU freezes the complete process tree of the container for consistency. Thus, as seen from the Figures 2(a) and 2(b), during checkpointing the source's throughput drops to zero and latency increases by 300%. This is for base CRIU stop-and-checkpoint approach, and a lower impact may be observed with its other checkpointing flavors such as incremental or iterative clone that adds additional overhead depending on the frequency of



(a) Throughput Overhead



(a) Latency Overhead

Figure 2. Runtime performance overhead for source container during hot-scale live cloning

iterations. Another option is to use post-copy, which may impact the performance on the destination due to memory misses caused by accessing pages that have not been copied yet to the destination.

C. Observations and Recommendations

In this section, we consider all the evaluated performance parameters together to draw general conclusions. For warm-scale provisioning, the size of the image was about 25% larger than the base image in cold-scale's case, and it delivers the same steady-state runtime performance as the latter. But its value comes from hiding the application initialization delay and a near-constant readiness latency that is less than 1 second compared to 7.5 seconds with cold-scale. Therefore, especially for short-lived containers where readiness latency is critical, warm-scale provisioning is more suitable. For example, on serverless platforms [43,44] where user actions are required to be executed in real-time in response to an event, they can be hosted in warm-scaled containers. Similarly, for IoT (Internet of Things) workloads [58] warm-scale containers are better match.

For the hot-scale method, the image size is a function of the amount of memory being used by a container. In our experiments, we measured it to be almost 100% larger than the base image. And it has about 2x readiness latency as compared to warm-scaling. But at runtime it delivers 25% higher throughput than its competitor. Thus, for containers where attaining a steady-state throughput quickly is critical, hot-scaling should be employed. For instance, while scaling an application for load-balancing, it is desirable to have a newly added instance operating at highest performance immediately. Similarly for scaling a Hadoop cluster running data intensive map-reduce task, or an HTTP web server hosting static pages.

In hot-scale provisioning, the running state of the source application is shared with the target instance, along with any data. Thus, from a security aspect hot-scaling should be limited to intra-tenant container scaling, while warm-scale can be used (cautiously) in a multi-tenant environment (especially for applications that do not require authentication and authorization). Also, as discussed in Section 3.1, another aspect, which needs to be considered, is the performance implication on the source application instance. This may vary depending on the CRIU checkpoint mode used during hot-scaling. This scaling method is ideal for migrating an application, where the source instance is to be decommissioned post-scaling, while the application keeps running at scale with minimal disruption. E.g., when migrating a container to a different physical machine with better performing hardware.

Finally, the most important take-away is: *"hot-scale is not always as hot"*. If you hot-scale a container while it is thrashing for low memory, then the new instance enters the thrashing state sooner. Thus, it is important to monitor and understand the operating state of a container for hot-scaling.

V. RELATED WORK

Existing container scaling techniques employ a start-from-scratch / cold-scale approach [1,2,3,5,7], while we also explore warm-scaling so as to avoid the application initialization latency [36]. While warm-scaling uses checkpointing for obtaining a container snapshot to base a new instance from, checkpoints have traditionally been used to provide fault tolerance / high availability at various abstraction levels- application runtimes [27-33], containers [4], VMs [9, 10, 18-21] and others [34,35]. In addition, checkpointing has also been employed as a means to suspend or consolidate idle systems [23, 24, 25, 37].

In this study, we also compare a hot-scaling methodology, where new container instances are created by live-cloning running containers. In the VM domain, such cloning-based instance creation has been employed for dynamic server scaling [41, 42, 17], parallel worker forking [12] and speeding up system testing [38]. Fast scaling can also be achieved by image-based techniques such as caching [15], p2p streaming [13], and lazy fetching [26].

While instance creation from snapshot also exists in the VM domain [39], they do not focus on warm-scale's motivation of avoiding application initialization latency. A similar approach is also proposed in [14], but to minimize OS

re-initialization latency during VM restart. Scaling via replication is also hinted upon in [6] for Tomcat application server containers. Other related work in the VM domain that comes closest to our motivation includes the proposal in [22], Twinkle [11], Dolly [8], and Jump-Start Cloud [16]. In [22], the authors propose using VM image caches in order to speed up concurrently booting VMs. However, they do not consider the application ready time that this study focuses on. Twinkle [11] speeds up VM instantiation by minimizing the VM snapshot size to consist of the working set of memory pages. In our case, the container snapshot size is significantly smaller than for VMs (application-only vs. application + OS), hence we focus on minimizing the application startup time. Unlike Twinkle, we also examine the effect of a warm cache on a new instance's performance. Dolly [8] proposes a database provisioning system based on VM cloning, where new instances are provisioned based on snapshots and paused VMs. Again, our analysis is different in that we do not see the need to use paused containers, since the restore time of a container is not significant. Moreover, we demonstrate that depending on the workload and the application parameters, starting from a clone may not always be beneficial for performance. In Jump-Start Cloud [16], an adaptive snapshot replication technique is proposed for high availability while considering the available resources. Whereas, we are not biased towards a single technique such as snapshotting, and objectively compare three different options.

It should be noted that in our experiments we use MySQL only as an example of a stateful application. By running a read-only workload against it, we bypass database consistency concerns which is not a focus of this paper. Live cloning a database node or restoring from checkpoint is not trivial, and may involve database resynchronization and transaction rollbacks for consistency [5, 8, 40].

VI. CONCLUSION AND FUTURE WORK

With the context of auto-scaling and on-demand provisioning, we compared three different techniques for container scaling-cold, warm and hot scaling. While differentiating the container start time from its application readiness, we compared the three approaches in terms of their readiness latency, image size and post-init performance. We discussed the applicability, advantages and shortcomings of each method, and highlighted performance tradeoffs and overheads which need to be considered for selecting the right scaling method. Going forward, we wish to further validate our observations with other applications, in addition to the MySQL and ElasticSearch containers. Finally, we believe if the Unikernel technology gains more maturity and adoption in building microservices, then Unikernel VMs should be compared to containers in terms of the performance of auto-scaling.

REFERENCES

[1] Hoenisch, Philipp, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. "Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers." In International Conference on Service-Oriented Computing, pp. 316-323. Springer Berlin Heidelberg, 2015.

[2] Steinholt, Ravn Kristinesønn. "A study of Linux Containers and their ability to quickly offer scalability for web services." Master's thesis, Univeristy of Oslo, 2015.

[3] Kukade, Priyanka P., and Geetanjali Kale. "Auto-Scaling of Micro-Services Using Containerization." International Journal of Science and Research (IJSR), Volume 4 Issue 9, September 2015.

[4] Li, Wubin, Ali Kanso, and Abdelouahed Gherbi. "Leveraging linux containers to achieve high availability for cloud services." In Cloud Engineering (IC2E), 2015 IEEE International Conference on, pp. 76-83. IEEE, 2015.

[5] Kang, Hui, Michael Le, and Shu Tao. "Container and microservice driven design for cloud infrastructure devops." In Cloud Engineering (IC2E), 2016 IEEE International Conference on, pp. 202-211. IEEE, 2016.

[6] He, Sijin, Li Guo, Yike Guo, Chao Wu, Moustafa Ghanem, and Rui Han. "Elastic application container: A lightweight approach for cloud resource provisioning." In 2012 IEEE 26th International Conference on Advanced Information Networking and Applications, pp. 15-22. IEEE, 2012.

[7] de Abranches, Marcelo Cerqueira, and Priscila Solis. "An algorithm based on response time and traffic demands to scale containers on a Cloud Computing system." In Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on, pp. 343-350. IEEE, 2016.

[8] Cecchet, Emmanuel, Rahul Singh, Upendra Sharma, and Prashant Shenoy. "Dolly: virtualization-driven database provisioning for the cloud." In ACM SIGPLAN Notices, vol. 46, no. 7, pp. 51-62. ACM, 2011.

[9] Chan, Hoi, and Trieu Chieu. "An approach to high availability for cloud servers with snapshot mechanism." In Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference, p. 6. ACM, 2012.

[10] Nicolae, Bogdan, and Franck Cappello. "BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots." In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 34. ACM, 2011.

[11] Zhu, Jun, Zhefu Jiang, and Zhen Xiao. "Twinkle: A fast resource provisioning mechanism for internet services." In INFOCOM, 2011 Proceedings IEEE, pp. 802-810. IEEE, 2011.

[12] Lagar-Cavilla, Horacio Andrés, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. "SnowFlock: rapid virtual machine cloning for cloud computing." In Proceedings of the 4th ACM European conference on Computer systems, pp. 1-12. ACM, 2009.

[13] Zhang, Zhaoning, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. "VMThunder: fast provisioning of large-scale virtual machine clusters." IEEE Transactions on Parallel and Distributed Systems 25, no. 12 (2014): 3328-3338.

[14] Goodson, Garth Richard, Sai Susarla, and Kiran Srinivasan. "System and method for fast restart of a guest operating system in a virtual machine environment." U.S. Patent 8,006,079, issued August 23, 2011.

[15] Razavi, Kaveh, and Thilo Kielmann. "Scalable virtual machine deployment using VM image caches." In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 65. ACM, 2013.

[16] Wu, Xiaoxin, Zhiming Shen, Ryan Wu, and Yunfeng Lin. "Jump‐start cloud: efficient deployment framework for large‐scale cloud applications." Concurrency and Computation: Practice and Experience 24, no. 17 (2012): 2120-2137.

[17] Mior, Michael J., and Eyal de Lara. "Flurrydb: a dynamically scalable relational database with virtual machine cloning." In Proceedings of the 4th Annual International Conference on Systems and Storage, p. 1. ACM, 2011.

[18] Cui, Lei, Zhiyu Hao, Chonghua Wang, Haiqiang Fei, and Zhenquan Ding. "Piccolo: A Fast and Efficient Rollback System for Virtual Machine Clusters." In Parallel Processing (ICPP), 2016 45th International Conference on, pp. 87-92. IEEE, 2016.

[19] Hou, Kai-Yuan, Mustafa Uysal, Arif Merchant, Kang G. Shin, and Sharad Singhal. Hydravm: Low-cost, transparent high availability for virtual machines. HP Laboratories, Tech. Rep, 2011.

[20] Cully, Brendan, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. "Remus: High availability via asynchronous virtual machine replication." In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pp. 161-174. 2008.

[21] Tamura, Yoshi. "Kemari: Virtual machine synchronization for fault tolerance using domt." Xen Summit (2008).

[22] Warszawski, Eduardo, and Muli Ben-Yehuda. "Fast initiation of workloads using memory-resident post-boot snapshots." U.S. Patent Application 14/930,674, filed November 3, 2015.

[23] Zhang, Liang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments." In Proceedings of the Eleventh European Conference on Computer Systems, p. 37. ACM, 2016.

[24] T. Knauth and C. Fetzer. DreamServer: Truly on-demand cloud services. In Proceedings of International Conference on Systems and Storage (SYSTOR'14), 2014.

[25] T. Knauth, P. Kiruvale, M. Hiltunen, and C. Fetzer. Sloth: SDN-enabled activity-based virtual machine deployment. In Proceedings of the Third workshop on Hot Topics in Software Defined Networking (HotSDN'14), 2014

[26] Harter, Tyler, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Slacker: fast distribution with lazy Docker containers." In 14th USENIX Conference on File and Storage Technologies (FAST 16), pp. 181-195. 2016.

[27] Cao, Tuan, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. "Fast checkpoint recovery algorithms for frequently consistent applications." In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 265-276. ACM, 2011.

[28] Stellner, Georg. "CoCheck: Checkpointing and process migration for MPI." In Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International, pp. 526-531. IEEE, 1996.

[29] Bronevetsky, Greg, Daniel Marques, Keshav Pingali, and Paul Stodghill. "Automated application-level checkpointing of MPI programs." In ACM Sigplan Notices, vol. 38, no. 10, pp. 84-94. ACM, 2003.

[30] Zheng, Gengbin, Lixia Shi, and Laxmikant V. Kalé. "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI." In Cluster Computing, 2004 IEEE International Conference on, pp. 93-103. IEEE, 2004.

[31] Lahiri, Tirthankar, Amit Ganesh, Ron Weiss, and Ashok Joshi. "Fast-Start: quick fault recovery in oracle." In ACM SIGMOD Record, vol. 30, no. 2, pp. 593-598. ACM, 2001.

[32] Ni, Xiang, Esteban Meneses, and Laxmikant V. Kalé. "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm." In 2012 IEEE International Conference on Cluster Computing, pp. 364-372. IEEE, 2012.

[33] Rezaei, Arash, Giuseppe Coviello, Cheng-Hong Li, Srimat Chakradhar, and Frank Mueller. "Snapify: capturing snapshots of offload applications on Xeon Phi manycore processors." In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, pp. 1-12. ACM, 2014.

[34] Laadan, Oren, Dan Phung, and Jason Nieh. "Transparent checkpoint-restart of distributed applications on commodity clusters." In 2005 IEEE International Conference on Cluster Computing, pp. 1-13. IEEE, 2005.

[35] Russinovich, Mark, and Zary Segall. "Fault-tolerance for off-the-shelf applications and hardware." In Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on, pp. 67-71. IEEE, 1995.

[36] SPEC Cloud™ IaaS 2016 Benchmark. http://spec.org/cloud_iaas2016/docs/designoverview.pdf. (Accessed January, 2017)

[37] Bila, Nilton, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. "Jettison: efficient idle desktop consolidation with partial VM migration." In Proceedings of the 7th ACM european conference on Computer Systems, pp. 211-224. ACM, 2012.

[38] J. Zhi, S. Suneja, and E. De Lara. The case for system testing with swift hierarchical vm fork. In Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing, HotCloud'14, pages 19–19, 2014.

[39] VMware. Understanding Clones. https://www.vmware.com/support/ws5/doc/ws_clone_overview.html. (Accessed January, 2017)

[40] Depoutovitch, Alex, and Michael Stumm. "Otherworld: giving applications a chance to survive OS kernel crashes." In Proceedings of the 5th European conference on Computer systems, pp. 181-194. ACM, 2010.

[41] Vrable, Michael, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. "Scalability, fidelity, and containment in the potemkin virtual honeyfarm." In ACM SIGOPS Operating Systems Review, vol. 39, no. 5, pp. 148-162. ACM, 2005.

[42] Bryant, Roy, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal De Lara. "Kaleidoscope: cloud micro-elasticity via VM state coloring." In Proceedings of the sixth conference on Computer systems, pp. 273-286. ACM, 2011.

[43] Amazon Lambda https://aws.amazon.com/lambda/. (Accessed January, 2017)

[44] IBM Bluemix OpenWhisk https://www.ibm.com/cloud-computing/bluemix/openwhisk. (Accessed January, 2017)

[45] Cargo: Container storage migration https://developer.ibm.com/open/openprojects/cargo/. (Accessed January, 2017)

[46] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (pp. 143-154). ACM

[47] IBM Bluemix Container Service. https://www.ibm.com/cloud-computing/bluemix/containers. (Accessed January, 2017)

[48] Red Hat OpenShift Container Platform. https://www.openshift.com/container-platform/features.html

[49] Google Cloud Platform. Container-Optimized Google Compute Engine Images. https://cloud.google.com/compute/docs/containers/container_vms. (Accessed January, 2017)

[50] Namespaces. Linux Programmer's Manual. man7.org/linux/man-pages/man7/namespaces.7.html. (Accessed January, 2017)

[51] Linux control groups. Linux Programmer's Manual. man7.org/linux/man-pages/man7/cgroups.7.html.

[52] CoreOS. Rkt vs. other projects. https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html. (Accessed January, 2017)

[53] Linux Foundation. runC by Open Container Initiative. https://runc.io. (Accessed January, 2017)

[54] M. Michael, J. E. Moreira, D. Shiloach and R. W. Wisniewski, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007, pp. 1-8.

[55] CRIU: Comparison to other CR projects. https://criu.org/Comparison_to_other_CR_projects. (Accessed January, 2017)

[56] CRIU. Main Page. https://criu.org. (Accessed January, 2017)

[57] Docker Hub. https://hub.Docker.com. (Accessed January, 2017)

[58] IBM. IBM Watson IoT Platform. https://www.ibm.com/internet-of-things/iot-solutions/watson-iot-platform. (Accessed January, 2017)