Containerizing Byte-Addressable NVM

Ellis R. Giles

Rice University
erg@rice.edu

***Abstract*—Container based applications are rapidly growing in popularity for virtualization due to the ease of deployment coupled with high-performance. Emerging byte-addressable, non-volatile memories, commonly called Storage Class Memory or SCM, technologies are promising both byte-addressability and persistence near DRAM speeds operating on the main memory bus. These new memory alternatives present a challenge for container based applications which typically access persistent data through layers of file isolation. This paper presents a high-performance containerized version of byte-addressable, non-volatile memory (SCM) for applications running inside a container.**

We created a container aware Linux loadable Kernel Module (LKM) that presents byte-addressable, persistent memory for containerized applications. We performed evaluation using micro-benchmarks, STREAMS, and Redis, and found our LKM with container support has near the same memory throughput for persistent in-memory applications as a non-containerized application and much higher throughput than persistent in-memory applications accessing SCM through Docker Storage or Volumes.

I. Introduction

Containers offer lightweight virtualization for applications and services running on the same host operating system. Containers are an alternative to full virtualization of a host operating system and services, only isolating running applications using a ”chroot” for persistent file accesses, Linux C groups for CPU and memory usage, and I/O isolation. Docker [13] is a relatively new open-source implementation of container-based virtualization technology that has been gaining in popularity for quick and easy cloud deployments and for recent work in live migration of containers using Flocker [14]. Docker creates lightweight Linux based containers with containerized applications having near the same performance as when the application is executed outside the container [6].

An exciting new memory technology with the potential of replacing hard drives and SSDs has the potential of changing traditional storage approaches. This new memory, called Storage Class Memory or SCM, is both byte-addressable and persistent and operates on the main memory bus, offering a solution between slow block based persistent storage and fast, byte-addressable volatile memory.

Figure 1 shows both traditional container based storage and Storage Class Memory. Access to SCM will be provided via a traditional mmap call to an underlying device driver or file [18]. This poses a problem for applications running inside an isolated container because an mmap of a file through an isolation layer may not gain performance benefits of SCM. Additionally, exposing a shared device to multiple containers can remove persistence isolation from containers and introduce security and portability issues.

We present a solution to this problem by introducing a Containerized SCM driver. It detects when applications are accessing the driver from within a container and presents an identical copy of the SCM.

II. Background

Virtualization is defined as having three key properties including isolation, encapsulation, and interposition [16], were isolation refers to guests not being able to affect others. Jails [9] was introduced for lightweight virtualization of environments to allow for sharing of a machine between several customers or users while still allowing for isolation of files and services of the guests on the same machine through the use of chroot and I/O constraints. Chroot changes the root of the file system to a different location for application level persistence isolation and security. This has many benefits since a system can be shared securely, but services such as CPU and memory were not isolated and could be abused by users.

Linux Containers or LXC [11] were introduced as the Linux version of Jails. The implementation added features to restrict memory and CPU usage that extended its isolation features. Docker [13] is an open-source project also for Linux which automates the deployment of applications or bundled services inside of Linux Containers. Volatile memory is handled from inside a container using regular virtual memory accesses with limits imposed by the operating system if Linux CGroups are used.

Handling access to persistent data inside a Docker Container is not a choice to be taken lightly, as there are several choices available: using a container file or traditional Docker Storage, an external or Docker Volume, or direct access to a device.

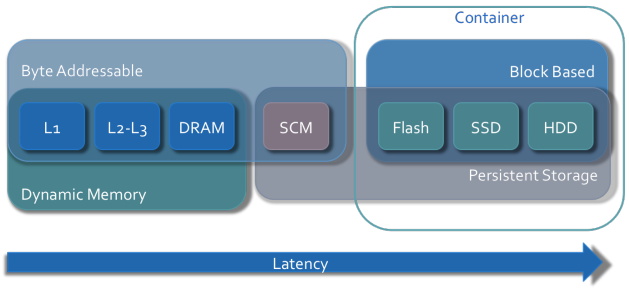

Fig. 1: Container based storage and Storage Class Memory

Each has it's own advantages and disadvantages and needs to be specified on container start if requiring a special device or volume:

1. Docker Storage: Storage to traditional files inside a container are accessed using a pluggable storage driver architecture. File accesses are layered using AUFS (or Another Union File System), Device Mapper, OverlayFS, VFS, or ZFS. A layered access requires copying of data through multiple file or persistence layers. AUFS uses Copy-on-Write and copies the entire file on first update, which could waste space but be faster if all of the contents of a file are updated. Device Mapper is a new option that performs copies at the block layer reducing space, but is slower on the writes to first blocks. Another drawback is the entire Docker service daemon must be configured for the driver and cannot be changed unless reconfigured and re-installed.

2. Docker Volumes: Docker Volumes are used to share data between containers or the container and the host. Usage is passed as options on startup of a container, and cannot be added later. Data is also not isolated, so changes in one environment affect other. If a container is paused and restarted elsewhere, volume data must be copied and managed by the Docker daemon.

3. Direct Device Access: Direct access to a device is given on start of a container and the same driver must be present on restart. Special privilege must also be granted to the container if it is going to write to the device; this also goes against isolation principles.

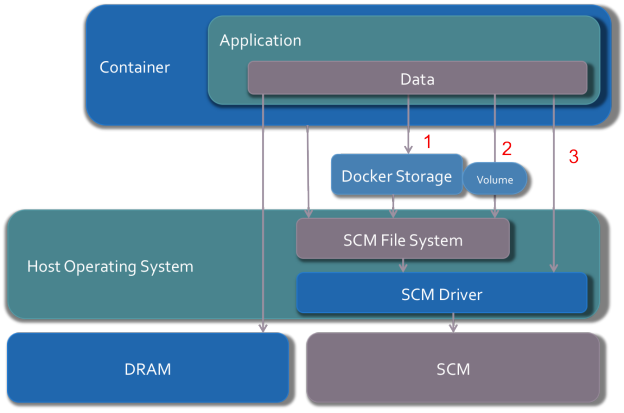

Fig. 2: Storage Class Memory exposed to Docker Container through a SCM Filesystem or Driver

Storage Class Memory presents a new situation for containers as it offers memory that is persistent. Options for SCM device access is shown in Figure 2. A privileged device access in situation 3 also is not isolated, so access to SCM from all containers to the device will not be isolated.

Combining Docker Containers with SCM poses an interesting challenge. If we expose SCM as a Docker Storage or Volume, then we might lose the performance of byte addressability. Further, exposing it as a dedicated, privileged device driver will lose application isolation and portability.

Persistence consistency guarantees impose additional problems. Consider a simple application that is writing to a variable *Z*. The value might be caught in a number of places from the processor cache to a number of write buffers. If an atomic operation is needed for an in-memory data structure, the new value of *Z* should be visible and accessible to others on the completion of a desired block or application complete. If a power failure occurs, for DRAM based variables, being located anywhere in the hardware is not problematic as the variable value is cleared on restart. However, if the value is located in SCM and a power failure occurs, the result of the computation may not be committed to persistent SCM. Even if a value is written to a memory location using a cache-line flush, *clflush*, a value may not be persisted to SCM.

The new Intel architecture specification [8] specifies new instructions such as *clwb*, or cache-line write-back, which writes a cache line out to the write buffers without invalidation and *pcommit*, or persistent memory commit, which doesn't retire until all globally visible stores are persisted to SCM. Both are weakly ordered and must be between store fences, or *sfence*.

This complicates fast reliable storage for Docker Containers, as multiple levels of synchronization, which are expensive, are required to maintain consistency through the layers. For instance, an application might be performing its own consistency guarantees through an Undo Log, which is a synchronous operation making a copy of a value before writing a new value. Before each write however, each value must be made persistent on the underlying medium. In the case of disk based persistence, this is accomplished through a disk flush. In SCM this can be accomplished through creating a log, writing the address of *Z* and value of *Z* to the log, *sfence, pcommit, sfence*, then writing the new value of *Z*. Once new values are written, the values may be flushed immediately or delayed. If the writes to the SCM locations are being performed through a Docker Storage Layer or Volume, then all of the writes to the variables, and the synchronization points are also passed through, which can be a very expensive operation. Therefore, it is advantageous to access SCM directly through the device driver. We present and evaluate a high-performance device driver that provides access to SCM while still allowing for isolation for Docker Containers.

III. Containerizing SCM

This section presents the Containerized Storage Class Memory design as a Linux loadable Kernel Module or LKM. The overall system design is shown in Figure 3.

The system implementation is comprised of three main components:

- Docker Command Line Client Integrations
- User Level Library
- Container Aware Linux loadable Kernel Module

Due to the lack of readily available SCM DIMMS for Docker testing, we used Ramfs to simulate SCM data. Then for a file system that does persistence consistency, we used an ext2 file system in the SCM data space. The file system

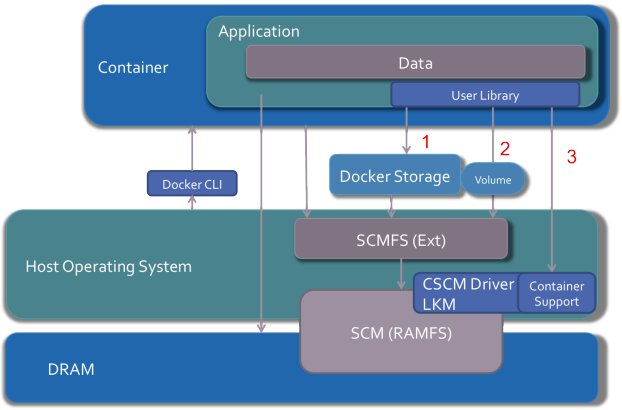

Fig. 3: Containerized Storage Class Memory System Design

contents for Docker and supporting libraries are installed into the SCM FS space. The file system layout is as follows:

Listing 1: Example SCM Data File Structure

```
SCM Driver (RAMFS) - /scmdata
|->hostdata
|->hostscmfs
        ext file sys mount /scmfs
        with Docker /var/lib/docker/
        /dfscmdata volumes
|
|->containers/
|->images/
```

The Docker container is executed using a privileged device driver to our CSCM LKM driver, which gives privileged access to the container to read and write to the device driver /dev/scm.

docker run –privileged -ti –device=/dev/scm –name=test1 rhel:7

Docker Client Integration

Docker 1.10.3 and API for Client Version 1.22 is built in a language called the Go Programming Language. The Docker API client allows for simple extensions to the Docker interface. It communicates to the Docker server daemon and responses from the daemon can be handled through the Go language. On container start, the command line to launch an image is:

docker run image id

We modified the docker api client run code to initialize the data in a container on startup. The image id is obtained from the return value from the Docker daemon after the container create image and before a container run. The data in the /scmdata image for the id is copied if an image exists for the new container being launched. If no image id is present, it need not be copied since the CSCM driver can create it on the first access. Likewise for a commit, the reverse procedure is followed.

User Library

The user library provides several functions, namely scmalloc and scmsync. The scmalloc call examines the environment and can be configured to provide access to any one of the three configurations described previously: Docker Volumes, Storage, our CSCM driver, or volatile DRAM. For volume or storage access, a named file can be specified and is then opened on first access and mmaped into user address space. The resulting pointer from the mmap call is returned and saved statically in the function and incremented by the requested size. Subsequent calls use the incremented pointer and increment the next available location as well.

The scmsync function call simply calls sync on the file handle. If the file handle is for /dev/scm, then it calls *sfence, pcommit, sfence*. The *pcommit* is emulated by CPUID.

CSCM Kernel Module

The Containerized SCM Linux loadable Kernel Module creates an SCM data area for the container, sets up the node and major version number. It registers our container supported /dev/scm device appropriately and registers the mmap handler. On container restart it is attached through the active device to the persistent SCM data.

On an mmap the following pseudo-code is executed:

Listing 2: CSCM LKM mmap flow

```
static int scm_mmap(filp, vma){
...
//  Get the file system root
//  Detect chroot (is fs root /)
//  Get SCM location
    //  If root, /scmdata/hostdata
    //  else examine /proc/1/cgroup
//  Lock process
//  Open SCM File
//  Unlock process
//  Setup VMA generic_file_mmap
}
```

One challenge after getting the generic VMA set was determining not just if an application is in a container but the container id itself. The container id dictates what underlying file to open. The file /proc/1/cgroup has different values depending on the OS.

Once the correct SCM data file is opened, the file can be mapped since it's using Ramfs underneath and the resulting generic file mmap call value can be returned to the scmalloc call. If the SCM data file isn't present, it is be created on the first access.

One area of consideration is security for the privileged CSCM LKM. For security, our CSCM LKM on application initialization, locks the file system for the current process so

it can't open a file when the LKM has chrooted itself to open the appropriate SCM file in the parent system.

IV. Evaluation

The SCM Containers core library and Linux loadable Kernel Module were built using gcc 4.8.2. Testing was performed on a machine equipped with an Intel(R) Xeon(R) CPU E5-2697 v2 processor running at 2.70GHz. The processor has 12 cores, each of which supports 2 threads, for a total of 24 hardware threads. The processor is equipped with 32GB of memory organized with four DDR3 8GB DIMMS, clocked at 1.867GHz. The host operating system and guest containers are identical versions of Red Hat Enterprise Linux Server release 7.2 (Maipo) running Linux kernel 3.10. In evaluation, SCM is exposed as 200MB chunks on the /scmdata file system. A Docker container is executed with:

docker run –privileged -ti –device=/dev/scm –name=test1 -v /scmfs:/vol -v /scmdata:/hostscm testa

We tested two micro-benchmarks followed by tests using STREAM [12] and Redis [17] using each configuration of Host, Docker Device (CSCM), Docker Storage and Docker Volumes.

The array update micro-benchmark creates an array in SCM using the CSCM user library scmalloc for 20 million 4-byte integers and randomly updates elements in the array. We record the average throughput of updates per second for each configuration. Figure 4 shows how the Docker CSCM device has near the same memory throughput as when the test is running on the host and higher throughput over storage and volumes, due to data having to go through multiple storage layers or the volume driver.

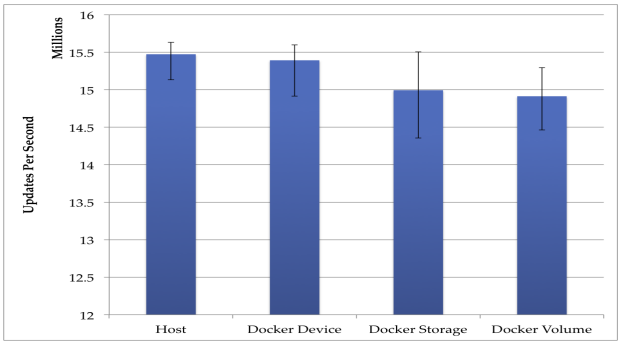

Fig. 4: Single Updates Per Second on Array of 4-Byte Integers

We then modified the Memory STREAM Benchmark [12] to allocate memory using CSCM scmalloc and linked with the CSCM user library. We tested the Copy function which sequentially copies data from one 90MB chunk to another location. Figure 5 shows how the Docker CSCM device has near the same memory throughput as when the test is running on the host. The sequential nature of the benchmark allows for the caching in the Docker Storage driver to have high performance; however, there are no consistency guarantees in this memory benchmark.

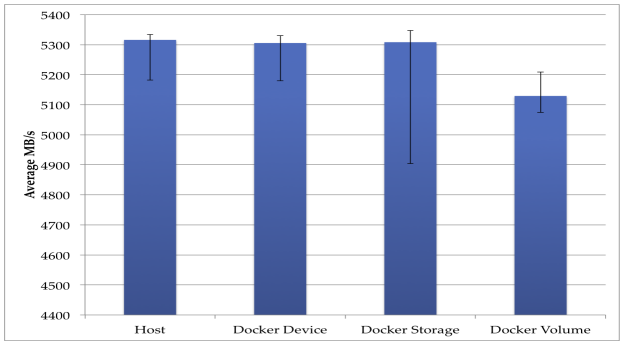

Fig. 5: Memory Copy Throughput in MB/s for STREAM

We tested a micro-benchmark that adds elements to an in-memory B-Tree, initialized with 200k elements. Figure 6 shows the average time required to insert additional elements. Docker Storage had the lowest value recorded. The equal performance is due to many of the top level elements in the tree are in the processor cache, resulting in a constant cost. Consistency guarantees in the following experiments show the higher performance of the CSCM Device Driver.

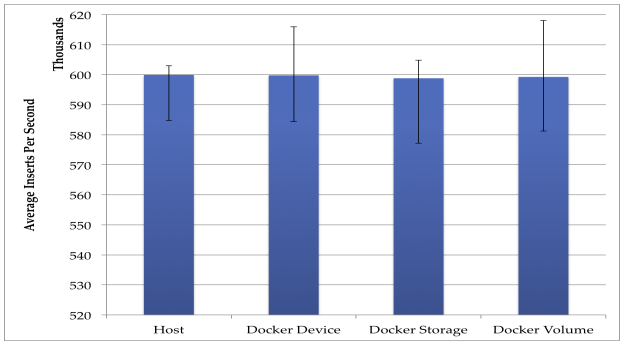

Fig. 6: Single Element Inserts per Second into B-Tree

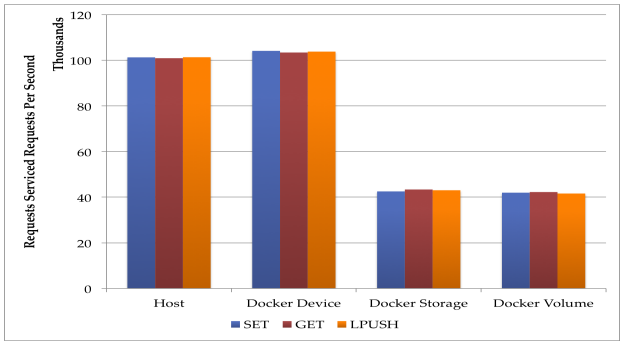

Fig. 7: Redis Benchmark of Service Requests Per Second

Next we tested several benchmarks in Redis [17], an in-memory data structure store, that offers lists and hash table values. We configured Redis to operate in memory and integrated it with the CSCM user library by modifying the memory allocator Jemalloc that ships with Redis to use our CSCM scmalloc routines. We executed the Redis benchmarks

for set, get, and list push times and recorded the average requests serviced per second. Figure 7 shows the CSCM driver in a Docker container has equal requests per second throughput as the same benchmark running on the host through the driver. These have twice the throughput as the Docker Storage or Volume options as requests do not have to flow through the Docker Storage or Volume drivers.

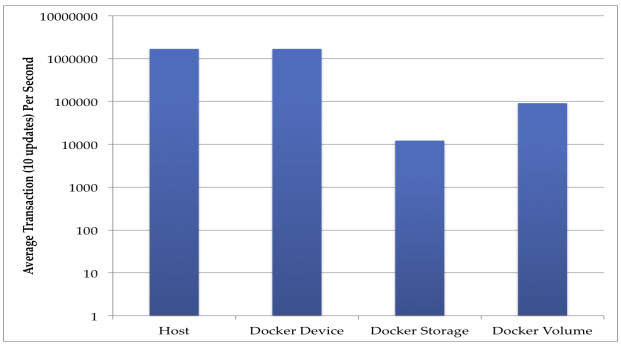

Fig. 8: Average Transactions Per Second (10 Element Updates)

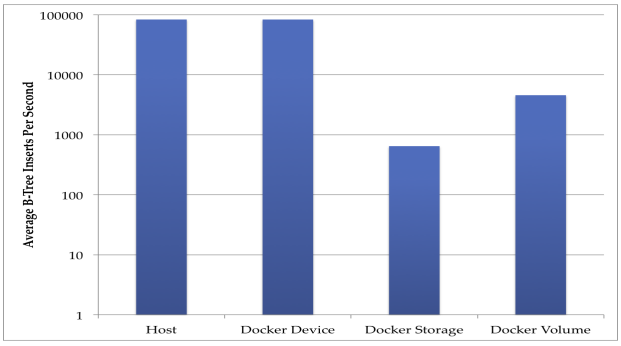

Fig. 9: Average B-Tree Insert Transactions Per Second

Finally, we tested persistence consistency by adding synchronization using scmsync in the CSCM library to the array and B-Tree tests. Figure 8 shows the array throughput and Figure 9 shows the B-Tree throughput. In both cases the CSCM driver running in the Docker Container has the same throughput as when running on the host. When the SCM is accessed through volumes however, performance degrades by a factor of 10. This is due to the additional level of synchronization required to the volume. Even slower is the Docker Storage driver which is another factor of 10 slower than the volume driver, making it two orders of magnitude slower than the host or device. This is due to the storage driver going through an additional layer. Future work includes exploring changes in the transaction size.

V. Related and Forward Work

Docker [13] has a number of supported file systems and volumes from AUFS, a layered union file system that uses Copy-on-Write when files are modified, to the Device Mapper which uses thin provisioning to implement the layers. Recent work on Flocker [14] manages Docker containers themselves and integrates with a Docker Swam manager allowing for Docker Data Volumes to follow migrated Docker Containers. However, this support is for block based devices and not byte-addressable, non-volatile memory which faces persistent memory consistency problems.

Consistency models for persistent memory was considered in [15]. For SCM atomic consistency, numerous software logging approaches and new hardware-software methods have been proposed [1], [2], [4], [7], [20], [21], [23]. These all rely atomic 8-64 byte writes to SCM and new instructions such as *pcommit*. NVM programming models will include use of mmap to access SCM through loads and stores [18]. However, these models do not address virtual machine containers for accessing SCM in isolation.

Several general purpose persistent memory file systems built on SCM have been proposed that can allow quick adoption of application use of SCM. Since these are general purpose file systems, they could support full virtual machines running on top of them such as KVM [10]. BPFS, or Block-Persistent File System, [3] uses copy-on-write techniques for a ordering of cache evictions but requires changes to the hardware.

The Persistent Memory File System [5], or PMFS, is a complete file-system implementation built for SCM but requires a dedicated kernel and doesn't address containers. SCMFS uses sequences of **mfence** and **clflush** operations to perform ordering and flushing of load and store instructions and requires garbage collection [22]. These file systems have to be accessed using regular Docker Storage or through Docker Volume management from Docker suffering from isolation and performance.

Research into Non-Volatile Memory allocators such as nvmalloc could be accessed from containers but do not support any sort of container isolation other than regular virtual memory isolation. Recent work to virtualize Non-Volatile Ram [19] was presented to support Xen hypervisor, but doesn't address the consistency guarantees that might be needed by host applications or support containers.

VI. Summary

Running applications inside containers using Docker is growing in popularity as it presents a low-cost, high performance method for isolating applications and services. Emerging byte-addressable non-volatile memory, commonly called Storage Class Memory, presents an interesting challenging for in-memory persistent applications running inside a container.

This paper investigated tradeoffs for presenting the SCM persistence to a container based application through a memory-mapped file inside a container, mounted as a volume, and as a container-aware Linux loadable Kernel Module. We presented and evaluated CSCM, for a containerized LKM with an mmap interface for in-memory applications and integration with Docker.

We found the container aware LKM to have the highest in-memory application throughput with orders of magnitude higher throughput for persistence to volumes while still achieving container persistence isolation for SCM.

REFERENCES

[1] CHAKRABARTI, D. R., BOEHM, H.-J., AND BHANDARI, K. ATLAS: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 433–452.

[2] CHATZISTERGIOU, A., CINTRA, M., AND VIGLAS, S. D. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment 8*, 5 (2015), 497–508.

[3] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of 22nd ACM SOSP* (2009), ACM Press.

[4] DOSHI, K., GILES, E., AND VARMAN, P. Atomic persistence for scm with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 77–89.

[5] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 15:1–15:15.

[6] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On* (2015), IEEE, pp. 171–172.

[7] GILES, E., DOSHI, K., AND VARMAN, P. Softwrap: A lightweight framework for transactional support of storage class memory. In *Proc. 31st Symposium on Mass Storage Systems and Technologies* (2015), MSST '15, IEEE.

[8] INTEL CORPORATION. Intel Architecture Instruction Set Extensions Programming Reference, October 2014. http://software.intel.com/.

[9] KAMP, P.-H., AND WATSON, R. N. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (2000), vol. 43, p. 116.

[10] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[11] LINUX. Linux containers, 2012.

[12] MCCALPIN, J. D. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter* (1995), 19–25.

[13] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[14] PEINL, R., HOLZSCHUHER, F., AND PFITZER, F. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing* (2016), 1–18.

[15] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *ISCA'14* (2014), pp. 265–276.

[16] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[17] REDIS.IO. Redis, a data structure store, 2016.

[18] RUDOFF, A. Programming models for emerging non-volatile memory technologies. *Login 83*, 3 (June 2013).

[19] RUIA, A. *Virtualization of Non-Volatile Ram*. PhD thesis, Texas A&M University, 2015.

[20] VENKATRAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte addressable memory. In *Proceedings of 9th Usenix Conference on File and Storage Technologies* (2011), ACM Press, pp. 61–76.

[21] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 91–104.

[22] WU, X., AND REDDY, A. L. N. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 39:1–39:11.

[23] ZHAO, J., MUTLU, O., AND XIE, Y. FIRM: Fair and high-performance memory control for persistent memory systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 153–165.