Resource Management for Running HPC Applications in Container Clouds

Stephen Herbein[1], Ayush Dusia[1], Aaron Landwehr[1], Sean McDaniel[1], Jose Monsalve[1], Yang Yang[1], Seetharami R. Seelam[2], and Michela Taufer[1](✉)

[1] University of Delaware, Newark, USA
taufer@udel.edu
[2] IBM T. J. Watson Research Center, Yorktown Heights, USA
sseelam@us.ibm.com

Abstract. Innovations in operating-system-level virtualization technologies such as resource control groups, isolated namespaces, and layered file systems have driven a new breed of virtualization solutions called containers. Applications running in containers depend on the host operating system (OS) for resource allocation, throttling, and prioritization. However, the OS is designed to provide only best-effort/fair-share resource allocation. Lack of resource management, as in virtual machine managers, constrains the use of containers and container-based clusters to a subset of workloads other than traditional high-performance computing (HPC) workflows. In this paper, we describe problems with the fair-share resource management of CPUs, network bandwidth, and I/O bandwidth on HPC workloads and present mechanisms to allocate, throttle, and prioritize each of these three critical resources in containerized HPC environments. These mechanisms enable container-based HPC clusters to host applications with different resource requirements and enforce effective resource use so that a large collection of HPC applications can benefit from the flexibility, portability, and agile characteristics of containers.

1 Introduction

While operating-system-based virtualization and containers are not new concepts, the emerging use of containers as a mechanism for operations across clusters in datacenters has the potential to change the computing landscape in HPC [26]. It also opens new research challenges in this field. Specifically, containers still need proper mechanisms for enforcing controlled resource allocation and management in containerized environments. This experimental paper describes mechanisms for the management of three key resources in containerized HPC applications: CPU, I/O, and network. The paper extends preliminary work presented in three posters at IEEE Cluster 2015 [9,11,19] by describing the mechanisms in a cohesive fashion and presenting their implementation in detail. Our implementation is based on Docker technology but can be adapted to other container technologies.

© Springer International Publishing Switzerland 2016
J.M. Kunkel et al. (Eds.): ISC High Performance 2016, LNCS 9697, pp. 261–278, 2016.
DOI: 10.1007/978-3-319-41321-1_14

Containers depend on the host OS for resource allocation and throttling. The fact that each container does not have its own kernel removes the kernel overhead from the container image and makes containers lighter weight in their memory and file system footprint as well as more easily portable than virtual machine (VM) images for HPC applications. On the other hand, the dependency on the host OS limits the containers' capability to control the allocation and management of HPC resources. Because traditional operating systems are designed to provide best-effort and fair-share resource allocation, they do not always support the workload requirements of containerized HPC applications. Thus, a containerized HPC application may, for example, experience substantial slowdowns because of the frequent context switching associated with the default fair-share scheduler, or they may suffer from high contention because of the fair-share bandwidth allocated by the OS kernel.

The contributions of this paper to the solution of resource managements problems in containerized HPC environments are threefold. First, we address the CPU allocation challenge for CPU-intensive applications running inside Docker containers that share the same compute node. By default, the host OS scheduling policy shares the node's CPUs between the containers using its fair-share quanta-based scheduling policy. In this policy each container is allowed to use a node's CPU for a predefined amount of time (typically 10 ms) before the next container is assigned to the CPU in a round-robin fashion. For applications such as LINPACK, this kind of fair sharing significantly slows the application performance because more time is spent on context switching than on real computation. To achieve better performance, we introduce a timeslicing mechanism [13] currently missing at the container level. When our mechanism is used, only a single container is scheduled at any time on shared resources for a prolonged period of time. This technique is akin to gang scheduling in HPC. With such a simple mechanism, we can improve an application's performance in a containerized environment by up to 4x.

Second, we address the challenge associated with disk I/O contention and disk I/O load imbalance in containerized applications across datacenter clusters. By default, containers are placed on nodes based only on available CPU and memory, ignoring the nodes' I/O load and capacity. This placement may result in poor I/O load balancing across the datacenter machines and ultimately in I/O hotspots for those nodes hosting containers with intensive I/O operations (e.g., frequent checkpointing). To prevent the formation of hotspots, we propose a two-tiered mechanism (i.e., at both the node and cluster levels) that extends Docker and Docker Swarm, making both capable of monitoring the containers' I/O activities and allocating containers based on I/O load balance across the datacenter nodes. We demonstrate how our two-tiered mechanism has the potential for higher bandwidth utilization without the contention effect.

Third, we address the challenge associated with network bandwidth throttling and prioritization. In order to ensure high-quality performance for critical, communication-intensive applications executed in containers, a required bandwidth level should be ensured without expanding or overprovisioning the network. By default in containers such as Docker, networks are configured to provide

the "best effort" to all the traffic. Under these conditions, parameters such as bandwidth, reliability, and packets per second for a specific HPC application cannot be guaranteed. Consequently, a communication-intensive containerized application may experience unacceptable performance when hosted on nodes with other containerized applications. We address this problem by proposing a mechanism that enables bandwidth limits and preferential delivery service for critical applications in containers. Our solution can control latency and delay while providing a way to reduce data losses.

The rest of this paper is organized as follows. Section 2 presents our mechanism for dynamic CPU resource allocation among containers. Section 3 presents methods to enforce I/O constraints among containers. Section 4 describes our mechanism to manage the network among containers. Section 5 provides background on containers and Docker as well as relevant related work. Section 6 briefly summarizes our conclusions.

2 CPU Allocations in Containers

Docker builds on the Linux kernel to allocate CPU resources to containers. Specifically, Docker uses Linux control groups (*CGroups* [2]) to provide portions of CPU resource pools to containers. Users specify the number of *shares* to give to a container or group of containers a priori (or statically), and the Linux OS gives each container a CPU allocation proportional to the number of shares that the container was allocated. When the specified shares saturate the CPU resource pools, the kernel allocates a fair share across all containers. Traditionally, containers are given time slices of 10 ms before context switching. The frequent context switching imposes an overhead on the system and causes cache thrashing, ultimately leading to performance degradation. Figure 1 shows an example of execution times for different numbers of containers ranging from 1 to 5, when running the LINPACK benchmark with a default 10 ms time slice. The optimal execution time has a linear behavior (ideal); the observed executing time is superlinear (10 ms time slice). The fine-grained time-slice granularity defines the interval each container is allowed to run on the CPU resources before being preempted. Every time a container's context switch is performed, the new context thrashes the content of the cache memory and overwrites the previously running container's context, requiring storing all its applications' values and execution state. The shorter the cycle in which the containers' context switching occurs, the less locality that can be exploited by the containers' applications and the more context switches that are performed.

To mitigate losses in performance associated with static resource allocation and frequent context switching, we design and implement a mechanism that allows Docker to define and deploy a dynamic, coarse-grained time slice for each container. The mechanism serializes the containers' consumption of CPU resources and is particularly suitable for high-caching, compute-intensive HPC applications in containers.

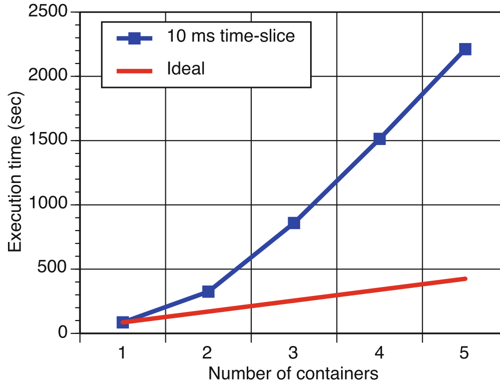


Fig. 1. Example of execution times for different numbers of containers running the LINPACK benchmark on an unmodified Linux OS with default 10 ms time slice.

2.1 CPU Allocation Mechanism

Current Docker containers support only static resource allocations. Shares are defined a priori and remain unchanged during the containers' executions. Our mechanism enables dynamic resource allocations at runtime by serializing the containers' execution for longer time-slice intervals. The serialization is obtained by increasing the shares of one container to 100 % and decreasing the shares of all other containers to 0 %. From an implementation point of view, we extend Docker with a simple but effective round-robin time-slice policy integrated into an Observe-Decide-Act (ODA) control loop [24] that schedules individual containers to be run one at a time for a defined time-slice interval. The round-robin policy first selects the container with the highest number of shares and allows it to run for the duration of the slice. The control loop then repeats, selecting containers in a round-robin fashion, each time reserving the entire pool of CPU resources for a single container.

With the ODA loop, we can collect container-specific information (e.g., priority information, application-specific information, and metadata on container requirements) that can serve as additional input for the decision-making process of the ODA loop. This is not possible if a third-party solution that runs outside Docker (e.g., in the Linux OS) is used. Moreover, the docker API and the command user interface are not changed, making the deployment of new capabilities for the user easier. Furthermore, we can leverage the Docker's client-server structure. The background Docker daemon provides the API to create, delete, or modify any container in the host OS; the Docker client provides the users with the command line interface, launches the daemon if necessary, and sends the commands to the daemon through the use of HTTP-like connections. In our implementation of the mechanism, a single Docker client can handle several Docker daemons, and a single ODA loop can handle multiple CPU resources in the distributed environment. Specifically, in our implementation, the CPU utilization is monitored at each step of the ODA loop; when overutilization is

detected, the allocation mechanism decides whether to adjust the CPU shares to control which container is using the CPU resources of the corresponding machine.

To implement the allocation mechanism, we modify the Docker 1.6.1 source files at three levels. First, we modify the Docker daemon's driver by adding a new function that modifies CGroups' parameters for a given container. Specifically, at the driver level, we create a new update driver interface that updates the containers' CGroups configuration dynamically by pulling the current configuration (i.e., `libcontainer::Container::Config`) and pushing a new configuration with updated CPU shares allocation (i.e., `libcontainer::Container ::Set`). We use the new function in the driver to make all of the necessary CGroup parameter modifications. Second, we add a new interface to the API of the server front-end that receives the `update` command for a specific container. Specifically, at the server front-end, we create a new POST handler that handles CPU share update commands sent from the client and initiates a new "update" job on the daemon. Third, we modify the Docker client by adding an additional command to the Docker command line interface. The new command handles the ODA loop, retrieves utilization information from the RESTful API, and pushes new CPU allocation information to the RESTful API. The source code of our implementation is available on GitHub at [22].

2.2 Empirical Results

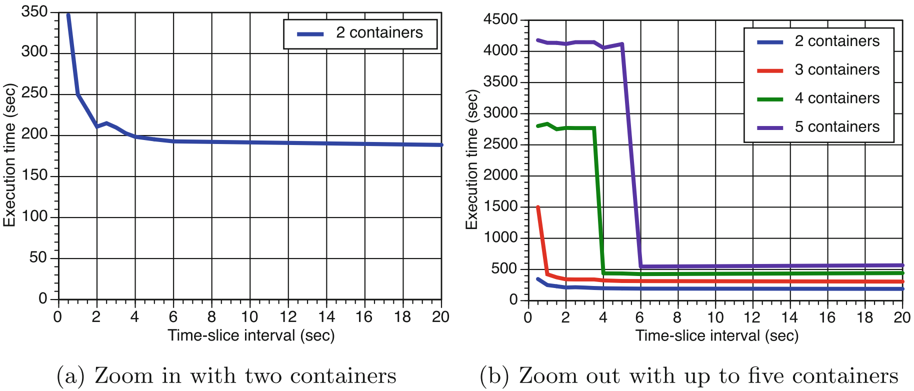
To evaluate Docker when using our mechanism, we create containers for a high-caching, compute-intensive benchmark such as LINPACK [10] with a matrix of size 1000×1000 and a block size of 100×100. This is a large-enough problem size to represent a long-term execution (i.e., longer than 60 s) on our 8-core nodes, one for each hardware thread. Each container utilizes 100 % of the CPU resources available on the host machine.

The first question we address in our assessment is the identification of an optimal time-slice size. To this end, we measure the execution time of the benchmark with different time-slice intervals ranging from 0.5 to 20 s and different numbers of containers ranging from 2 to 5. Figure 2a zooms in on the execution times for two containers; Fig. 2b zooms out for four configurations of 2, 3, 4, and 5 containers. For smaller time-slice intervals, we see no improvement in performance because of a delay in the reaction time for CGroups when used through Docker. Reaction time here is the amount of time elapsed between when a CGroup is changed and when the change results in altering the application performance. CGroup value changes do not instantaneously impact application performance because the OS scheduler does not pick them up as soon as they are changed. We observed an average delay time of approximately 700 ms. As the time-slice interval grows, we observe a performance sweet spot around 10 s for any number of containers considered in our tests.

The second question we address in our assessment is the impact of our mechanism on performance. To this end, we consider the optimal time-slice interval of 10 s (within the observed sweet spot window observed in Fig. 2b). Figure 3 shows the context switching of 5 containers when using our mechanism for the

LINPACK benchmark. In the test depicted in the figures, we start the adaptive scheduler after 10 s. The figure outlines the overlapping in the containers' executions before our mechanism is applied and the subsequent time-slice intervals. Within every ODA cycle, each container assumes complete control over the CPU resources for its time-slice interval. This process repeats until all the containers terminate their computation.

In Fig. 4 we compare the execution times of containers under differing levels of overallocation when using our mechanism (10 s time slice) versus using the traditional Linux implementation (10-ms time slice). For the sake of completeness we also add the ideal performance trend. As already outlined in Fig. 1, without our mechanism, as we increase the oversubscription of resources, the execution times



(a) Zoom in with two containers (b) Zoom out with up to five containers

Fig. 2. Impact of time-slice intervals on execution times for different number of containers. (Color figure online)

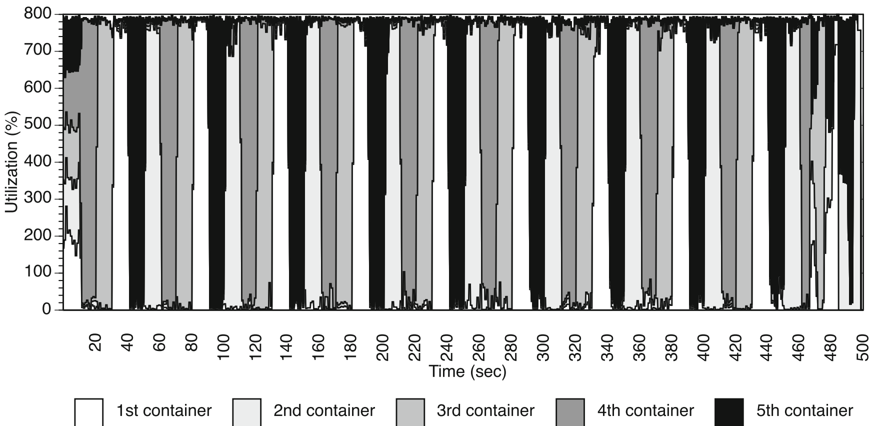


Fig. 3. Time slice of five containers and 10 s time slice.

significantly increases in a superlinear pattern because of caching and synchronization conflicts. With our mechanism, on the other hand, the performance is almost linear with the number of concurrent containers and improves by approximately a factor of 4 because of the mitigation of the aforementioned issues. With our mechanism, five containers take ∼5x the time of a single container whereas in the default model, five containers take over ∼20x times the time of a single container.

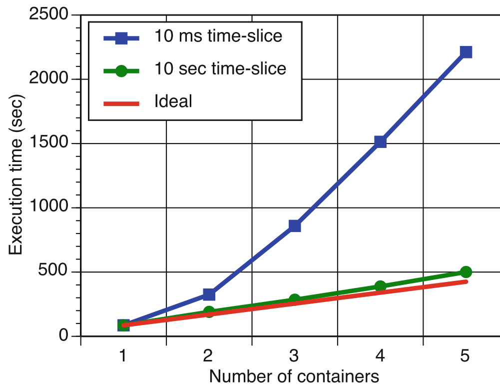


Fig. 4. Execution time for different numbers of containers with and without our mechanism with a time-slice interval of 10 s. (Color figure online)

We observe that our mechanism based on serializing applications in containers is efficient when there is an oversubscription of CPU resources and applications are highly optimized for maximal resource usage such as in LINPACK. The longer time-slice interval ultimately reduces cache thrashing and the overhead of context switching. Unlike the Linux scheduler, our time-slice intervals are on the order of seconds; performance sweet spots are observed for intervals ranging between 10 and 15 s. Under this time slice, a number of containers ranging between two and five are able to freely operate alone for long periods of time while taking full advantage of the locality during their execution. Our results demonstrate that an adaptive control scheme can positively affect the overall application performance. We are not suggesting that this mechanism is suitable for all applications; different application domains need different time slices. LINPACK is an example of compute-intensive applications where OS-based allocations is clearly not sufficient and a better CPU allocation results in a significant performance improvements when the application is executed in containerized workloads.

3 I/O Management of Container Clusters

HPC applications perform disk I/O operations (i.e., read and write) for various reasons such as reading the input parameters of a program, writing output

results of a simulations, and periodically checkpointing the simulation state. The management of containerized applications is ignorant of the applications' load and nodes' I/O capacity. In other words, containers are placed on machines based only on available CPU and memory knowledge. When dealing with I/O-intensive applications, this naive placement of containers ultimately results in poor I/O load balancing across datacenter machines. Figure 5a shows an example of naive allocation of containers and its impact on the I/O bandwidth. The results in the figure refer to 90 containers executed in 14 virtual machines, each container hosting one I/O-intensive application. Each application continuously streams data to disk with different I/O rates: of 2 MB/s, 4 MB/s, or 8 MB/s. The I/O rate is randomly assigned at the container's launch time. The number of containers per VM depends on what containers are selected to be scheduled on the specific VM. In the figure we observe that after all the applications have been launched, the I/O bandwidth imbalance between nodes is significant, on the order of 50 MB/s. While the example in the figure depicts a load-imbalanced scenario across VMs, a similar behavior is expected across nodes in datacenter clusters.

To balance the load across nodes, we define a two-tier solution that combines a node-level mechanism and a cluster-level mechanism. At the node level, we change the Docker daemon to allow monitoring each container's I/O load, and we set absolute upper bounds on the container's I/O bandwidth. At the cluster level, we enable the scheduling system to perform load balancing by placing new containers across nodes based on the node's system I/O capability and utilized I/O bandwidth. Together these two mechanisms allow I/O-intensive applications to effectively execute with available I/O resources in HPC containerized environments.

3.1 Allocation Mechanisms

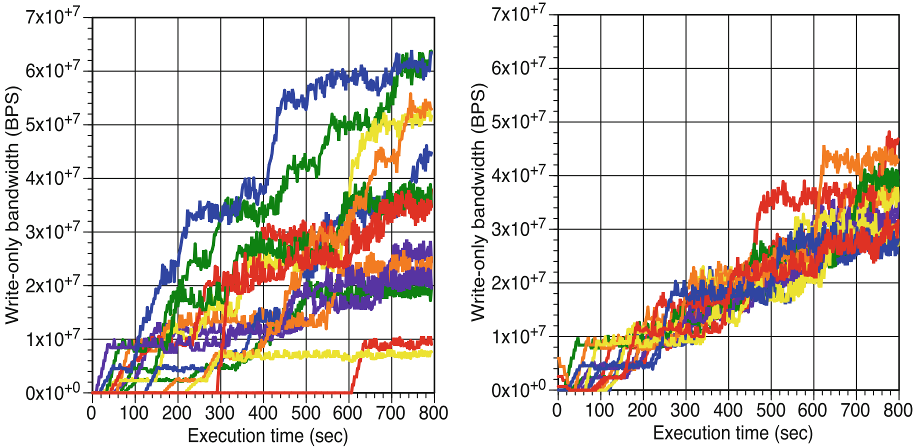
While kernels provide statistics on the I/O activity of each node and the `Blkio` CGroup has various parameters that allow us to control allocations such as I/O operations per second and bandwidth per second, container managers such as Docker do not provide API functions to leverage these capabilities [1]. At the node level, we extend the Docker API to capture the I/O device status for each container; the status includes disk utilization in terms of how much time the storage device has outstanding work (i.e., is busy), write bandwidth in terms of the number of bytes written to the device per second, read bandwidth in terms of the number of bytes read from the device per second, and wait time in terms of the average time (milliseconds) for I/O requests issued to the device to be served. To capture the I/O activities and to modify bandwidth limits for reads and writes in containers, we augment the client-side API of the Docker daemon in three ways. In our first modification, we add the ability for the Docker daemon to determine the maximum I/O bandwidth of the node it is running on. When a Docker daemon starts, it continuously performs unbuffered writes of 4 MB for 30 s and then calculates the effective bandwidth over that time period. In our second modification, we add the ability for the Docker daemon to determine the

current I/O bandwidth utilization on each of the nodes. The Docker daemon parses the node's kernel information present at `/proc/diskstats`. Among the many statistics stored in `/proc/diskstats` is the number of sectors read and written to disk since boot. Every two seconds, the modified Docker daemon parses the number of sectors read and written, comparing the values between samples, and computes a running average of bytes accessed per second over 15 samples. We add this information, along with the maximum bandwidth, to the response for the `/info` Docker API call. For the third modification, we enable the Docker daemon to set the `blkio.throttle_write_bps_device` flag and the `blkio.throttle_read_bps_device flag` in the `Blkio` controller. Setting these flags under the `CGroup` directory named after the container ID allows us to limit the I/O bandwidth of a container.

At the cluster level, we ensure that the I/O load is being balanced across entire clusters. This task involves ensuring that the I/O load is distributed across the nodes of a cluster and that no node in the cluster is overburdened with I/O operations. To this end we monitor the average load of each node (i.e., CPU, memory, and I/O loads); the I/O load is monitored, and the I/O device status is stored with the modifications described above. When a new container is scheduled, it is allocated to the node with the lowest total load, where the weight of each load type is user configurable. We implement our solution on top of Docker Swarm. Note, however, that our method is generic to any other scheduling solution and can be easily extended to other schedulers such as Mesos and SLURM. While Docker Swarm includes the capability to monitor CPU and memory as well as to schedule containers across a cluster, it lacks knowledge of the I/O capacity and utilization of the containers. The node-level mechanism described above extends the Docker daemon to make the I/O activity of each node and the I/O throttling of containers available through the `/info` Docker API call. The information is made available to Docker Swarm by augmenting the internal data structures of Swarm to store the I/O information for each node in the cluster and by increasing the frequency at which Swarm makes the `/info` API call to each Docker daemon so that Swarm always has an up-to-date view of the cluster's I/O state. We include the collected I/O information in Swarm's scheduling strategies and adapt the node weighting function that Swarm uses to determine the current load on a node to integrate the I/O weight together with the existing CPU and memory weights. These modifications to Docker Swarm allow us to determine when the I/O of a node is saturated and consequently stop scheduling containers on that specific node. The modifications also allow Swarm to better load balance container allocations across the cluster when using the *spread* scheduling strategy since it can include I/O when calculating the load on a node. The source code of the implementation of our node-level [20] and cluster-level [15] mechanisms is available on GitHub.

3.2 Empirical Results

To demonstrate the benefits of I/O knowledge once integrated in Docker Swarm, we repeat the test in Fig. 5a but with our modified Swarm. As described above,



(a) Example of I/O load imbalance obtained with the original Docker and Docker Swarm.

(b) Example of balanced I/O load obtained with augmented Docker and Docker Swarm.

Fig. 5. Examples of imbalanced and balanced I/O load obtained with and without our augmented Docker and Docker Swarm. Each line represents the I/O used by one of 14 VMs running multiple I/O-intensive containerized applications. (Color figure online)

our test is performed on 14 VMs running 90 containers on the modified Docker daemons. Each VM had a dedicated core and dedicated hard drive to minimize contention. Figure 5b shows the results for the augmented Docker and Docker Swarm with full knowledge of the system's I/O. Contrary to the results in Fig. 5a, the I/O load across the 14 VMs is well balanced, and each VM is roughly using the same I/O bandwidth. More important, no single VM has maxed out its disk bandwidth. Under this balancing scheme, I/O-intensive containerized applications get their desired bandwidth easily and without contention. More generally, Fig. 5b provides a proof of concept to support the claim that the additional I/O knowledge allows Swarm to make better decisions at schedule time, which ultimately result in better load balancing and higher resource utilization. When applied to containers on HPC clusters, the same mechanism allows containerized applications to satisfy the I/O requirements and balance the load across the nodes of a datacenter.

4 Network Allocations in Containers

The Docker networking interface offers limited options to configure the network usage of containers. Docker networks are currently configured to provide the "best effort" to the network traffic. When Docker boots up, a single virtual Ethernet bridge, called `docker0`, is generated. By default, all the containers are configured to be in the same subnet and use `docker0` to enable the communication with one another. Thus, no throttling or traffic prioritization is

available. Without throttling, communication-intensive applications can starve other applications. Moreover, the lack in network traffic prioritization can cause poor performance in time-sensitive containerized applications. For example, a container hosting a real-time HPC application (e.g., a streaming application with fast Fourier transformations requiring iterative manipulations) may require more bandwidth than does a container hosting a batch or analytics applications. Figure 7a shows an example of a scenario with four containers with different traffic priorities: the first container has high network traffic priority, the second medium priority, and the last two low priority. Each container hosts an application that downloads a 450 MB file over FTP from an FTP server. Although we assign different priorities to the four containers a priori and the four download jobs start simultaneously (with one job in each container), the network bandwidths do not reflect the desired traffic prioritization with the default Docker networking.

To support different bandwidths and priorities for a pool of Docker containers with different network requirements, we implement two mechanisms that allow Docker to provide prioritization and throttling based on a network priority or rate limit assigned to containers by the user a priori. The two mechanisms can be used separately or in concert, since they complement each other.

4.1 Throttling and Priority-Based Allocation

Our priority-based mechanism extends Docker to include a priority scheme for network traffic. The priority scheme is enforced by using a packet classifier and scheduler. When using the packet classifier, packets are classified and added to one of three available priority queues (i.e., high, medium, or low). The medium level is the default level that is assigned to any container when no priority is defined by the user. The scheduler dequeues the packets and sends each packet to a container according to the queue's priority. Figure 6 provides a high-level overview of the mechanism.

We use the Linux traffic control (TC) utility to classify and then schedule the network traffic in and out of the `docker0` interface [6]. Using the TC utility, we configure a network scheduling algorithm, also called a queuing discipline (`qdisc`), for both the ingress and egress directions [5]. Specifically, we use the PRIO `qdisc`, a scheduling algorithm that contains an arbitrary number of classes with priorities. We configured the PRIO `qdisc` to utilize three classes, where Class 1 has the highest priority, Class 2 (the default class) has a medium priority, and Class 3 has the lowest priority. In order for the PRIO `qdisc` to schedule packets based on priority, the packets must first be classified. We use TC filters to classify incoming and outgoing packets based on their IP address. A priority is assigned to containers when they are created. Since each container has an unique IP address, packets coming from or going to a container's IP address take on the priority of that container. For example, all the packets that have destination IP address of a container with high priority are enqueued to Class 1.

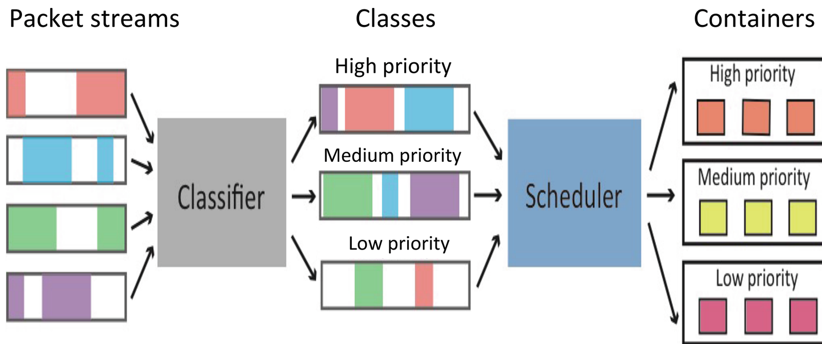


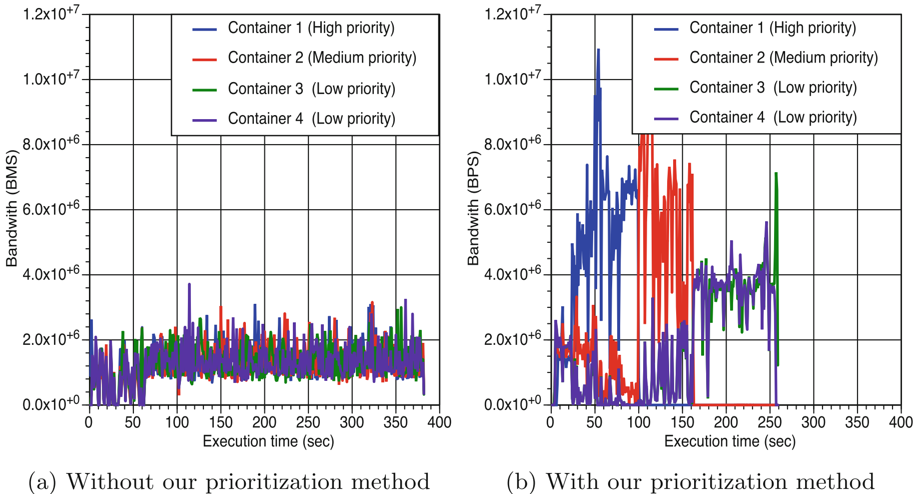
Fig. 6. Overview of the packet classifier and scheduler workflow.

Similarly, the packets with destination IP address of a container with default and low priority are added to Classes 2 and 3, respectively. Once the packets are properly classified and populate the appropriate class queues, the PRIO `qdisc` can begin scheduling the packets. The PRIO `qdisc` scheduler first checks for packets in the queue of Class 1; if no packets are available to dequeue, then the queue of Class 2 is checked; and if no packets are available to dequeue in the queue of Class 2, the queue of Class 3 is checked. The dequeuing of packets from the queue of different classes enforces the scheduling policy and priorities of containers. The problem with using just a PRIO `qdisc` is that individual connections, or "flows," within the same class can contend with one another and degrade performance. To avoid this scenario, we add a stochastic fairness queuing (SFQ) `qdisc` to each class. SFQ ensures fairness within each class by scheduling flows of the same class in a round-robin fashion, thus preventing any single flow from drowning out the rest of the flows.

Our throttle-based mechanism gives Docker the capability to throttle the rate at which the packets are sent or received by a given container. To throttle each container to its specified limit, we use TC to apply an independently configurable token bucket filter (TBF) `qdisc` to each container's packet queue. Each TBF has two required parameters. The first parameter is the traffic rate limit, specified by the users and assigned during the containers' initialization. The second parameter is the burst size, which determines the size of the buffer used by the TBF to queue packets when traffic is being throttled. If the buffer size is too small, packets may be dropped because more packets arrive than can be accommodated in the buffer. These dropped packets will cause an overhead to our throttling mechanism. We reduce this overhead by configuring the `qdisc` parameters for optimal performance—specifically, by setting the burst size to 10 % of the user-defined throttle rate. The source code of the implementation of both our network prioritization and throttling mechanisms is available on GitHub [12].

4.2 Empirical Results

We assessed our priority-based mechanism by repeating the test in Fig. 7a with the same four containers uploading a 200 MB file to the same FTP server but this time with the property-based mechanism in place. Figure 7b shows the network throughput observed for the four containers. In Fig. 7a we use the default Docker network, and the network throughput remains the same for all three containers despite the user-defined priorities. On the other hand, in Fig. 7b we observe different network throughput for the four containers because of our mechanism. Initially, the high-priority container (#1) has the highest share of the total throughput. When the file download completes for the high-priority container, the medium-priority container (#2) gets the highest share of the total throughput. Similarly, when the file download completes for the medium-priority container, the low-priority containers (#3 and #4) get all the bandwidth. The results also show that the low-priority containers get an equal share of the total throughput. These results prove that our mechanism implements priorities in containers and that the containers with equal priority get an equal share of the available bandwidth.

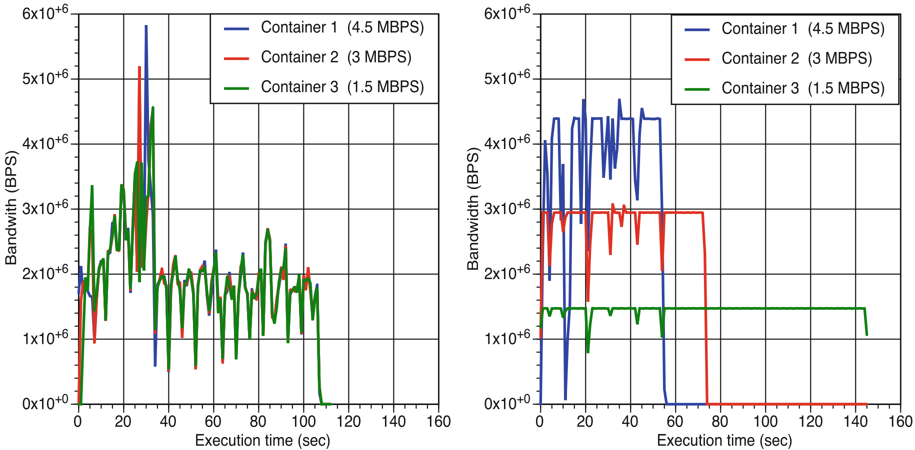


(a) Without our prioritization method (b) With our prioritization method

Fig. 7. Network throughput of 4 containers downloading 450 MB over FTP with and without our network prioritization mechanism. Container 1 is assigned high priority, Container 2 medium priority, and Containers 3 and 4 low priority. (Color figure online)

To assess our throttle-based mechanism, we assign different bandwidth limits to three containers running on the same node and then monitor the network throughput experienced by each container. The containers' limits are 4.5, 3, and 1.5 MBps. The containers are configured to execute only a single job of uploading a file of size 200 MB to an FTP server. We use an FTP server to receive files from

all the containers. The network throughput of all three containers is monitored for the duration of the uploads, first without throttle-based mechanism and then with the mechanism in place. Figure 8a shows the network throughput obtained by the three containers without our mechanism; Fig. 8b shows the throughput for the same test with our mechanism.



(a) Without our throttle-based mechanism (b) With our throttle-based mechanism

Fig. 8. Network throughput of three containers without and with throttle-based mechanism. When the throttle-based mechanism is in place, the three bandwidth limits of 4.5, 3, and 1.5 MBps are observed for the three containers' uploads. (Color figure online)

Without the throttle-based mechanism, the network throughput is almost the same for all the containers. On the other hand, with limits, the network throughput is throttled to 4.5 MBps for Container 1, 3 MBps for Container 2, and 1.5 MBps for Container 3. The dips in the network throughput are due to the congestion control mechanism of TCP. In particular, if a packet loss occurs, multiple duplicate ACK signals are received, and the congestion window (`cwnd`) is reduced by the sender. The congestion window size estimates the congestion between sender and receiver and avoids overloading the link between sender and receiver with too much traffic.

The results of our empirical tests show that our Docker implementation with the throttle-based and priority-based mechanisms efficiently provides resource allocations to containers based on priorities and requirements. Specifically, our extension to Docker networking guarantees that containers' network bandwidth matches assigned priorities. Providing priority-based network allocation to containers has three advantages. First, container hosting bandwidth-intensive applications can be assigned a limit to prevent them from contending with other containers sharing the network. Second, operating costs can be reduced by using

existing network resources more efficiently and thus delaying or reducing the need for expansion or upgrades. Third, time-sensitive and critical applications hosted in high-priority containers can now be assigned higher priority to get a higher share of network bandwidth, without starving other containers. Moreover, when containers host applications using UDP, which is not sensitive to network congestion, our mechanisms allow the associated containers to be throttled appropriately to achieve the desired level of bandwidth sharing.

5 Background and Related Work

This section provides background on container technologies, including container clouds. Also discussed in related work on Docker.

5.1 Containers, Docker, and Container Clouds

OS-based virtualization has supported containerization since the early 1980s, enabling users to isolate and customize their processes' environments (e.g., each container can have its own set of libraries while sharing resources of the system). FreeBSD Jails [18], Solaris Zones, AIX WPARs (Workload PARtitions), and HP UX containers are all examples of container technology used in various application domains.

Linux OS introduced resource control groups (i.e., *CGroups*) in 2007. *CGroups* are a set of features to measure, control, and isolate CPU, memory, disk I/O, network, namespace, and devices to a set of processes. Namespaces in particular allow the creation of containers with their own identify in terms of hostname, process identifiers, and network devices. *CGroups* are used to build isolated execution contexts called containers. Docker [21], Warden [3], Kubernetes [4], LMCTFY, LXC, LXD, and rkt are only a few examples of container technologies built on top of Linux *CGroups*. As of this writing, Docker is by far the most popular container technology. Figure 9 shows a typical system with Docker technology. The host operating system consists of the Docker Container Manager (i.e., Docker Engine). This engine can create, modify, and delete containers. Containers are created from file system packages called Docker images. A Docker image is a file system that contains all of the software necessary to run an application. The software includes the application code, dependent libraries, tools, and system libraries. There can be one or more docker containers per application on a given host, as shown in the figure. Other container technologies such as Warden and rkt are built on these same concepts. Although each container can have its own set of application codes and libraries, all the containers running on a host share the host kernel. Thus, the Container Manager depends on the host kernel for resource isolation, management, and enforcement. Linux kernels provide only fair sharing of resources across processes; advanced sharing policies are not normally implemented as part of the base kernel.

Container clouds are technologies that enable a cluster of Docker Engines to run on multiple hosts and offer the cluster resources as a service to multiple users

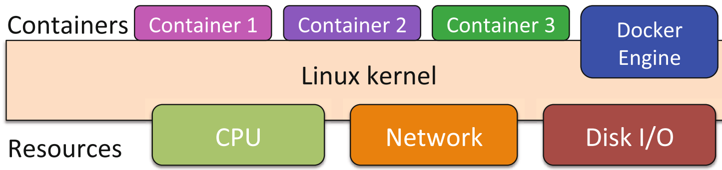


Fig. 9. Illustration of a typical container system. Docker Engine creates and manages the containers. The host OS provides the resource management functions.

with supporting tools for application composition, deployment, and operations. Docker Swarm is a clustering solution to group a set of Docker Engines. When an application composed of multiple containers is deployed to a Swarm cluster, different containers of the application may run on different nodes. In this context, efficient resource management has to be enforced across the multiple containers. Such efficient resource management is still missing for container-based clusters hosting HPC applications.

5.2 Related Work

The National Energy Research Scientific Computing Center (NERSC) developed Shifter [17], a system that allows for HPC centers and users to utilize Docker images in their normal workflows. It makes code deployment easier for users and software stack management easier for center administrators because application dependencies are integrated into the Docker images. Their system integrates with many existing HPC resources such as high-speed interconnects, parallel filesystems, batch job schedulers, resource managers, and HPC-specific operating systems (Cray Linux environment). Their work shows that Docker can be a valuable addition to the HPC workflow, but it lacks extensive testing of the overhead associated with containers.

Extensive analyses have been made of the overheads associated with containers compared with other virtualization methods and "bare metal" execution [8, 14, 23, 27]. These analyses are usually limited to a single machine, however, and rarely study the effects of multiple containers running simultaneously. Soltesz et al. do study the contention caused by multiple containers on an overallocated system, but their work targets Linux-VServer, a virtualization technology that predates Linux containers and is limited to the CPU and I/O resources of a single machine [25].

Others have developed two-tiered systems for resource management of containers. Hong et al. developed a node-level system that monitors the CPU and memory usage of the containers running on each node in the cluster and a cluster-level scheduler that places new containers on nodes with the lightest load [16]. This is similar to how Docker Swarm schedules containers. Our two-tiered method is similar, with the crucial difference being that our method includes disk I/O in the load calculation. Blagodurov et al. developed a node-level system that pins memory-intensive applications to separate NUMA nodes

in order to minimize contention and a cluster-level manager that migrates heavily contended applications to less contended nodes [7]. Their method does not directly integrate with the container scheduler, however, and thus they make decisions only at runtime, after the contention has occurred. Our method seeks to prevent contention before it happens, by integrating with the container scheduler and improving the scheduling decisions.

6 Conclusion

The resource management provided by the operating system is not sufficient for containerized workloads in HPC. We discuss resource management challenges in CPU, network, and I/O and describe solutions that achieve better application performance and system utilization. Our CPU management mechanism allows users to mitigate the performance slowdown due to frequent context switching and fair share. Our I/O management mechanism deals with I/O contention by allowing Docker to set up I/O bandwidth limits and to perform cluster-wide I/O load balancing. Our network management mechanism enables application-specific bandwidth priorities and bandwidth limits. Our results demonstrate that advanced resource management technologies are necessary to leverage containers for a broad set of HPC applications.

Acknowledgment. This work is supported by NSF grant #312259 and #312236. We also thank IBM for providing access to their Softlayer (http://www.softlayer.com) and Supervessel (https://ptopenlab.com) cloud platforms and for providing guidance on container technologies. Our code can be found on GitHub at [12,15,20,22].

References

1. Block I/O Controller. https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt
2. CGroups. https://www.kernel.org/doc/Documentation/cgroups/
3. Cloud Foundry Warden. https://github.com/cloudfoundry/warden
4. Kubernets by Google. http://kubernetes.io
5. Linux Advanced Traffic Control. http://lartc.org/howto/
6. Network Classifier CGroup. https://www.kernel.org/doc/Documentation/cgroups/net_cls.txt
7. Blagodurov, S., Fedorova, A.: Towards the contention-aware scheduling in HPC cluster environment. J. Phys. Conf. Ser. **385**(1), 012010 (2012)
8. Dandapanthula, N., Stanfield, J.: High Performance Computing - Containers, Docker, Virtual Machines and HPC. http://en.community.dell.com/techcenter/high-performance-computing/b/general_hpc/archive/2014/11/04/containers-docker-virtual-machines-and-hpc
9. Diaz, J.M.M., Landwehr, A., Taufer, M.: Poster: resource management layers for dynamic CPU resource allocation in containerized cloud environments. In: Proceedings of the IEEE Cluster 2015 Conference, pp. 1–2, September 2015
10. Dongarra, J.J.: Performance of various computers using standard linear equations software. SIGARCH Comput. Archit. News **20**(3), 22–44 (1992)

11. Dusia, A., Yang, Y., Taufer, M.: Poster: network quality of service in Docker containers. In: Proceedings of the IEEE Cluster 2015 Conference, pp. 1–2, September 2015
12. Dusia, A., Yang, Y.: Nework QoS Mechanism - Docker, May 2015. https://github.com/adusia/docker
13. Feitelson, D.G., Rudolph, L.: Parallel job scheduling: issues and approaches. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 1–18. Springer, Heidelberg (1995)
14. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2015)
15. Herbein, S.: I/O QoS Mechanism - Docker Swarm, May 2015. https://github.com/SteVwonder/swarm
16. Hong, J., Balaji, P., Wen, G., Tu, B., Yan, J., Xu, C., Feng, S.: Container-based job management for fair resource sharing. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 290–301. Springer, Heidelberg (2013)
17. Jacobsen, D., Canon, R.: Contain this, unleashing Docker for HPC. In: Cray User Group (CUG 2015), Chicago, IL, April 2015
18. Kamp, P.H., Watson, R.N.M.: Jails: confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference (2000)
19. McDaniel, S., Herbein, S., Taufer, M.: Poster: a two-tiered approach to I/O quality of service in Linux. In: Proceedings of the IEEE Cluster 2015 Conference, pp. 1–2, September 2015
20. McDaniel, S.: I/O QoS Mechanism - Docker, May 2015. https://github.com/seanmcdaniel/docker/
21. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
22. Monsalve, J., Landwehr, A.: CPU QoS Mechanism - Docker, May 2015. https://github.com/josemonsalve2/docker
23. Ruiz, C., Jeanvoine, E., Nussbaum, L.: Performance evaluation of containers for HPC. In: Hunold, S., et al. (eds.) Euro-Par 2015 Workshops. LNCS, vol. 9523, pp. 813–824. Springer, Heidelberg (2015). doi:10.1007/978-3-319-27308-2_65
24. Sironi, F., Bartolini, D., Campanoni, S., Cancare, F., Hoffmann, H., Sciuto, D., Santambrogio, M.: Metronome: operating system level performance management via self-adaptive computing. In: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pp. 856–865, June 2012
25. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 275–287 (2007)
26. Vaughan-Nichols, S.: New approach to virtualization is a lightweight. Computer **39**(11), 12–14 (2006)
27. Xavier, M., Neves, M., Rossi, F., Ferreto, T., Lange, T., De Rose, C.: Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 233–240, February 2013