# Adaptive Resource Views for Containers

Hang Huang[1], Jia Rao[2], Song Wu[1], Hai Jin[1], Kun Suo[2], and Xiaofeng Wu[2]
[1]National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[2]The University of Texas at Arlington, USA
Email: {huanghang, wusong, hjin}@hust.edu.cn, {jia.rao, kun.suo, xiaofeng.wu}@uta.edu

## ABSTRACT

As OS-level virtualization advances, containers have become a viable alternative to virtual machines in deploying applications in the cloud. Unlike virtual machines, which allow guest OSes to run atop virtual hardware, containers have direct access to physical hardware and share one OS kernel. While the absence of virtual hardware abstractions eliminates most virtualization overhead, it presents unique challenges for containerized applications to efficiently utilize the underlying hardware. The lack of hardware abstraction exposes the total amount of resources that are shared among all containers to each individual container. Parallel runtimes (e.g., OpenMP) and managed programming languages (e.g., Java) that rely on OS-exported information for resource management could suffer from suboptimal performance.

In this paper, we develop a per-container view of resources to export information on the actual resource allocation to containerized applications. The central design of the resource view is a per-container `sys_namespace` that calculates the effective capacity of CPU and memory in the presence of resource sharing among containers. We further create a virtual `sysfs` to seamlessly interface user space applications with `sys_namespace`. We use two case studies to demonstrate how to leverage the continuously updated resource view to enable elasticity in the HotSpot JVM and OpenMP. Experimental results show that an accurate view of resource allocation leads to more appropriate configurations and improved performance in a variety of containerized applications.

## CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Garbage collection**; • **Computer systems organization → Multicore architectures**.

## KEYWORDS

Container, Scheduling, Memory Management, Performance.


*HPDC '19, June 22–29, 2019, Phoenix, AZ, USA*

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325403

**ACM Reference Format:**
Hang Huang, Jia Rao, Song Wu, Hai Jin, Kun Suo, and Xiaofeng Wu. 2019. Adaptive Resource Views for Containers. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, 12 pages. https://doi.org/10.1145/3307681.3325403

## 1 INTRODUCTION

As a lightweight alternative to *virtual machines* (VMs), containers have been increasingly adopted to package applications for fast, cross-platform deployment. Leading cloud providers, such as Amazon AWS, Google Compute Engine, and Microsoft Azure, support a wide range of containerized applications, from long-running batch jobs to short-lived microservices, and to complicated machine learning workloads. Cluster management frameworks, including Mesos, YARN, and Kubernetes, also use containers as the basic unit for resource allocation and scheduling. Unlike virtual machines, which are isolated from each other through the abstraction of virtual hardware, containers execute directly on the physical hardware and share the same *operating system* (OS) kernel. Isolation between containers is attained through separate namespaces and control groups (cgroups). Namespaces give each container its own view of the PID space, file systems, networks, etc; cgroups allow fine-grained, precise allocation of resources, such as CPU, memory, disk, and network bandwidth, in containers. The absence of the virtual hardware abstraction in container-based virtualization eliminates most of the overhead in conventional VM-based virtualization. Studies have shown that containers achieve near-native performance on CPU, memory, and I/O in various workloads [10, 26, 27].

While the weak isolation in containers helps attain near-native performance, it can cause several issues. Besides exposing security vulnerabilities, the weak isolation creates a semantic gap in containerized applications. On the one hand, containers usually have limited access to resources. System administrators often control the amount of the resources allocated to a container through resource shares and limits. On the other hand, the absence of the virtual hardware abstraction allows containers to observe the total available resources on a host, though they can only access a subset of the resources.

The gap on the view of resources can undermine the efficiency of a wide range of applications when containerized [10]. Server programs, programming runtimes, and libraries often devise sophisticated schemes to manage resources on their own. The illusion of total resource availability deceives applications to improperly manage resources, thereby leading to suboptimal performance or even program crash. For example, the HotSpot *Java virtual machine* (JVM) automatically probes online CPUs at startup and uses the

CPU count to determine the number of JVM threads in a parallel *garbage collection* (GC); OpenMP dynamically adjusts the number of threads in a parallel region based on the availability of CPUs. The HotSpot JVM sets the maximum heap size of a Java program to one fourth of the total memory size. When such applications run in containers with resource constraints, over-threading or memory overcommitment could lead to substantial performance loss.

A few applications have recognized this issue and started to add container awareness into their designs. The Java JDK, since version 9, detects CPU and memory limits when launching a Java program from a container and configures the JVM accordingly. Specifically, it uses three configurations set by administrator, i.e., the CPU affinity, quota, and limit, to infer CPU availability and limits the JVM heap size to the hard memory limit of a container. However, there lacks a general solution in containers to export per-container resource availability to applications. Furthermore, it is challenging to accurately determine the effective resource capacity of a container in a multi-tenant environment with resource multiplexing, reservation, and capping. LXCFS [1] is an userspace file system that exports per-container resource limits through the `procfs` interface. The Linux kernel version 4.6 introduces a private control groups namespace to expose per-container resource limits to each container. However, these approaches only export the resource constraints set by the administrator but do not reflect the actual amount of resources that are allocated to a container. Since most OSes are work-conserving, allowing users to use more than their quotas if otherwise resources are left idle, the actual resource constraints seen by a container depends on not only its own resource limits but also the resource usage of colocated containers.

In this paper, we demonstrate that an isolated, per-container view of available resources is necessary for attaining high efficiency and preventing erroneous behaviors in containerized applications. To this end, we develop a `sysfs` namespace in addition to the existing namespaces in containers. The virtual `sysfs` exports the effective resources, such as the number of cores and the amount of memory that are available to a container. Effective resources are calculated based on the resource *reservation*, *share* and *limit* and are updated in real time. We further demonstrate that by leveraging the proposed adaptive container resource view, two popular runtime systems, HotSpot JVM and OpenMP, can be made elastic to the availability of resources. Experimental results with Docker using representative Java and OpenMP benchmarks show significant performance improvement.

The rest of this paper is organized as follows. Section 2 introduces the background of container-based virtualization and presents motivating examples on the issue of the semantic gap. Section 3 introduces the design and implementation of our proposed per-container adaptive resource view and Section 4 presents two case studies that leverage the resource view to enable dynamic parallelism and elastic heap management. Section 5 presents experimental results and Section 6 reviews the related work. Section 7 concludes this paper.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Container-based Virtualization

Containers rely on two OS mechanisms, namespace and resource control group, to realize isolation. Without loss of generality, we describe these two mechanisms in the context of Docker and Linux. Other containers, such as Rkt and LXC, and other OSes employ similar techniques. A container is essentially a group of processes running together. Namespaces create restricted views of the host system to isolate containers from each other. Typical namespaces include the PID namespace, the user namespace, the mount namespace, the UTS namespace, and the network namespace. For example, the PID namespace allows processes in a container to have virtual PIDs starting with PID 1. The real PIDs in the host OS are mapped to the virtual PIDs but are hidden from the container. Similarly, the user namespace re-maps a privileged user from within a container to a non-privileged user in the host; the mount namespace provides isolated view of the filesystem. Each container is configured a local root filesystem, in which application binaries and system libraries are loaded for program execution. Besides the root filesystem, the `procfs` and `sysfs` are required to be mounted into a container. These filesystems provide the local view of the system, including the private PID space and user groups.

Control groups (cgroups) isolate the resource usage, such as CPU, memory, disk, and network, of containers. Processes belonging to the same container form a resource control group and share the same resource budget. Cgroups use *share* and *limit* to precisely control the resource allocation to each container. *Share* reflects the relative importance of containers when competing for shared resources; *limit* sets up an upper bound on the amount of resources a container can consume. Additional constraints can be further configured to specify the locality or affinity of resources. In what follows, we discuss the allocation of CPU and memory in containers.

Linux relies on the *completely fair scheduler* (CFS) to enforce priority and performance isolation among containers/cgroups. The tunable parameter `cpu.shares` determines the relative share of CPU time allocated to each container. If $n$ containers contend for CPU, each attempting to use 100% CPU, the CPU time a container receives is $\frac{cpu.shares}{\sum^{n} cpu.shares}$. Parameters `cfs_period_us` and `cfs_quota_us` together determine the limit of CPU allocation. cfs _period_us specifies the accounting period of CPU in microseconds, at the end of which CPU should be reallocated. `cfs_quota_us` indicates the total amount of CPU time a container can use during one period. The ratio of quota to period determines the CPU capacity of the container. If the ratio is less than one, a container can only use part of the time on a single CPU, after which it is throttled. If the ratio is larger than one, the container can run on multiple CPUs. In addition, parameter `cpuset.cpus` specifies the set of CPUs on which a container is permitted to run.

Cgroups use two types of memory limits to control memory allocation. `memory.limit_in_bytes` sets the absolute upper bound on a container's memory usage. If a container's memory exceeds this hard limit, the container either is killed or starts swapping. `memory.soft_limit_in_bytes` sets a soft limit on container memory. Unlike a hard limit, which cannot be surpassed even the system has sufficient available memory, a soft limit allows a container to use as much as memory as needed unless there is a memory shortage in the system. When Linux detects low memory, containers whose memory usage exceeds their soft limits gradually reclaim memory until either their memory usage falls below the respective soft limits or the system-wide memory shortage is resolved.

## 2.2 Resource Availability vs. Constraints

Although namespaces provide a per-container view of the host system, an important piece of information is missing in the existing namespaces. Since containers share the OS kernel, all hardware devices and resources are visible to them, though each only has access to a subset of the total resources. The illusion of total resource availability and the actual resource constraints present a semantic gap to applications running in containers.

The semantic gap can cause severe inefficiencies or even lead to program crashes. A wide range of applications automatically configure runtime parameters based on the probing of available resources. For example, HotSpot, the most widely used JVM, determines the degree of concurrency in parallel *garbage collection* (GC) based on the number of online CPUs. OpenMP dynamically adjusts the number of threads in each parallel section based on available CPU. If not explicitly configured by users, HotSpot automatically sets the maximum heap size of a Java program to a quarter of the total memory.

When the effective capacity of a container due to resource constraints is lower than the total capacity, the autoconfiguration will unfortunately over-provision threads and memory. Over-threading can cause excessive thread management overhead; over-committing heap memory can lead to significant swapping activities or *out-of-memory* (OOM) crashes. Most containerized applications detect resource availability through the OS kernel. For example, many programming runtimes, including the JVM and OpenMP, acquire information on available resources through the standard `sysconf` interface in the GNU C library (`glibc`). `sysconf` queries `sysfs` or `procfs` in order to determine the number of online CPUs. Memory size is calculated based on two memory parameters provided by the Linux kernel: _SC_PHYS_PAGES * _SC_PAGESIZE.

Unfortunately, the `sysconf` interface and the `sysfs` pseudo filesystem in the Linux kernel are not container aware and always report the total resource availability instead of the per-container resource constraints. Therefore, containerized applications may mistakenly over-commit resources. To estimate the severity of the problem, we manually examined the top 100 application images in DockerHub. Specifically, we looked into the source code of applications (e.g., httpd, tomcat, and MongoDB) and runtimes (e.g., JVM, OpenMP, and Chrome V8) to see if they rely on the Linux kernel to report resource availability for auto-configuration. We classified application images into two categories: *affected* by the semantic gap and *unaffected*. Applications are grouped by the programming language they use. As shown in Figure 1, a total number of 62 out of the top 100 applications are potentially affected by this semantic gap. Among the 7 languages we studies, all Java and PHP-based programs could suffer resource over-commitment. A majority of C++-based applications and half of C-based applications are also affected.

Note that our analysis only revealed the risk of potential container resource over-commitment, but did not quantify the impact on application performance. To this end, we selected a few representative, Java-based applications to evaluate how this semantic gap would affect performance. JVM transparently sets the number of parallel GC threads and JIT compiler threads, and the maximum

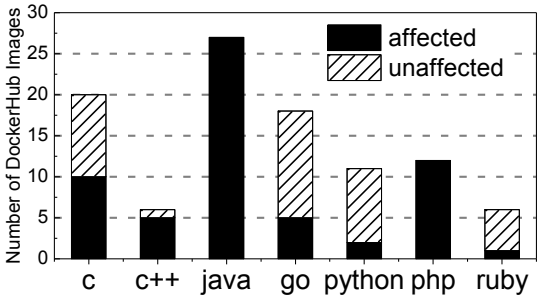
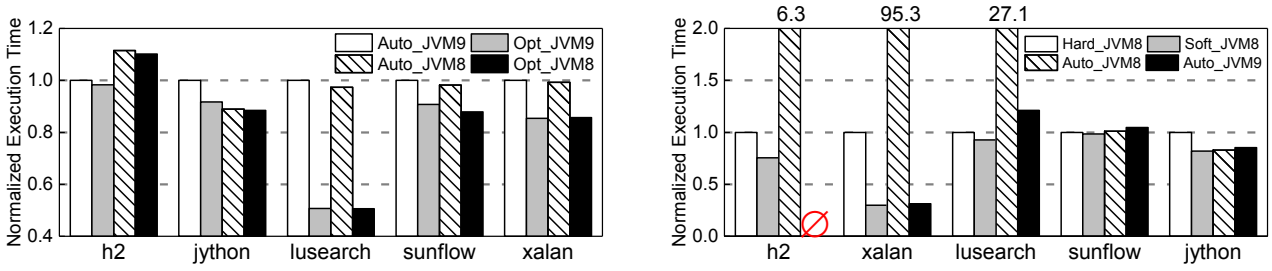



**Figure 1: Analysis of the top 100 application images on DockerHub**

Java heap size according to the host configuration of CPU and memory. The Java community has recognized this issue and starts to add container awareness since Java version 9. Specifically, JDK 9 queries the Linux kernel to obtain the CPU and memory limits, and uses them to configure the JVM.

We created two container environments to study how the configuration of GC threads and Java heap size can affect performance, respectively. For each experiment, we compared JDK 8 and JDK 9 with hand-optimized JVMs that had perfect knowledge on the resource constraints. To study the semantic gap on CPU availability, we collocated 5 docker containers on a machine with 20 cores. All containers were configured with a CPU limit of 10 cores and an equal CPU share. The five containers ran the same program from the Dacapo benchmarks [7]. Since 5 containers shared 20 cores, our hand-optimized JVMs set the number of GC threads to 4 for each container. We normalized performance to state-of-the-art JDK 9 with docker awareness. As shown in Figure 2(a), the container awareness in JDK 9 had limited impact on performance compared to JDK 8, which is oblivious to container resource constraints. The reason is that JDK 9 only detected the *static* CPU limit (i.e., 10 cores) configured by cgroups while unable to recognize the effective CPU capacity (i.e., 4 cores). In contrast, hand-optimized JDK 8 and 9 significantly outperformed the existing auto-configuration schemes in most cases, indicating the importance of detecting resource constraints during runtime and accurately determining the effective capacity.

The determination of memory capacity of a container during runtime is more challenging. We placed one container on a machine with 128GB memory and set its hard limit on memory to 1GB and the soft limit to 500MB. As such, the default JDK 8 without container awareness (denoted as *auto_JVM8*) will set the maximum heap size to one quarter of the physical memory, i.e., 32GB., while JDK 9 (denoted as *auto_JVM9*) will set to one quarter of the hard limit (1GB), i.e., 256MB. The two hand-optimized JVMs, i.e., *hard_JVM8* and *soft_JVM8*, set the maximum heap size to the hard limit and the soft limit, respectively. We also ran a memory-intensive workload in the background to cause memory shortage on the machine. One notable observation is that JDK 9 had large performance improvement over JDK 8, which mistakenly set to a too large heap based on the physical memory size and caused significant swapping. However, the heuristic used in JDK 9 does reflect the real-time usage of memory and can cause OOM errors (the missing data in Figure 2(b))




(a) Configuration of GC threads

(b) Configuration of JVM heap size

**Figure 2: The impact of container resource constraints on Java performance**

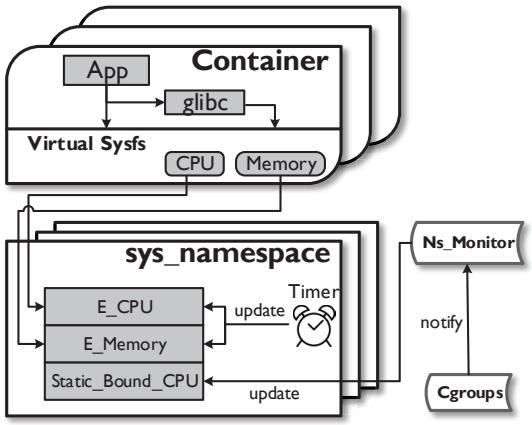



**Figure 3: System architecture**

as the working set size of *H2* cannot fit in the heap size set by JDK9 (i.e., 256MB). Overall, setting the maximum heap size to the soft limit achieved the best performance as this prevented expensive memory swapping.

**Summary** We have used two examples to demonstrate that it is important to make applications aware of the container resource constraints. The challenges lie in 1) how to accurately determine the effective capacity of a container in real time in a multi-tenant environment with resource affinity, share, and limit; 2) how to adapt applications to the possible changing capacity of a container, without extensive changes to the application source code. In the next few sections, we elaborate on the design of an adaptive resource view for containers and how to build elastic apps based on the resource view.

## 3 DESIGN

To bridge the semantic gap between the illusion of total resource availability and the actual constrained resource allocation in containers, we develop a per-container view of real-time resource allocation. The adaptive resource view is continuously updated to reflect the effective capacity of a container. Figure 3 illustrates the system architecture that enables the resource view. It consists of three components: a virtual `sysfs`, a new `sys_namespace`, and a system-wide daemon `Ns_Monitor`. The virtual `sysfs` provides an interface connecting the `sys_namespace` with user space applications. It intercepts application queries to the `sysfs` and re-directs them to the new `sys_namespace`, from where per-container view of resources is returned. `sys_namespace` maintains the effective capacity of CPU and memory allocated to a container. `Ns_Monitor` tracks changes in cgroups settings, including container creation/termination and adjustments to resource constraints, and updates the corresponding `sys_namespaces`.

### 3.1 Sys_namespace

While the existing namespaces, such as PID and network namespaces, isolate containers by limiting their access to system resources, `sys_namespace` is designed to provide a per-container view of resource allocation of containers. It calculates the effective capacity of resources allocated to a container and exposes it to applications running in the container through the `sysfs` interface. In this paper, we focus on the effective capacity of CPU and memory. Besides being updated by `ns_monitor`, `sys_namespace` is equipped with a timer that periodically updates effective CPU and memory based on container resource usage.

**Effective CPU** is the maximum amount of CPU time that can be utilized by a container, given its resource limit and share. To interface with `sysfs`, `sys_namespace` exports the information on effective CPU capacity in a way similar to that in a physical system, in which CPU capacity is measured by the number of online CPUs. Specifically, for each container, `sys_namespace` exports a finite number of CPUs, whose aggregate capacity equals the amount of CPU time that can be used by the container. Expressing effective CPU as discrete CPU count offers two advantages. First, existing studies [9, 23] have shown that, for a variety of workloads, it is more efficient to execute threads on a few stronger, dedicated CPUs than running them on a large number of shared CPUs, on which each thread only gets a slice of CPU time. Exporting effective CPU in the form of an equivalent number of dedicated CPUs allows user space applications to appropriately determine thread-level parallelism (as will be discussed in Section 4.1). Second, this design is compatible with applications that probe system resources based on CPU count.

Algorithm 1 shows the calculation of effective CPU for each container. First, we determine the lower and upper bounds of effective CPU, which refer to the *reservation* and *limit* of a container's CPU allocation, respectively. The *share w* determines the portion of the total CPU capacity ($|\mathbb{P}|$ CPUs) that is guaranteed to be allocated to

**Algorithm 1** The calculation of effective CPU

1: **Variables:** CPU share of the $i^{th}$ container $w_i$; CPU limit of the $i^{th}$ container $l_i$; CPU mask of the $i^{th}$ container $\mathbb{M}_i$; effective CPU updating period $t$; CPU usage of container $i$ during the updating period $u_i$; system-wide unused CPU capacity $p_{slack}$; online CPU set $\mathbb{P}$;
2: **Output:** The number of effective CPU $E\_CPU_i$ for container $i$.
3: `/* Calculate effective CPU based on limit, affinity and share*/`
4: $LOWER\_CPU_i = \min(\frac{l_i}{t}, |\mathbb{M}_i|, \lceil \frac{w_i}{\sum w_j} \cdot |\mathbb{P}| \rceil)$
5: $UPPER\_CPU_i = \min(\frac{l_i}{t}, |\mathbb{M}_i|)$
6: $E\_CPU_i = LOWER\_CPU_i$
7: `/* Adjust effective CPU based on container CPU usage*/`
8: **repeat**
9: **if** $p_{slack} > 0$ **then**
10: **if** $\frac{u_i}{E\_CPU_i \cdot t} > 95\%$ and $E\_CPU_i < UPPER\_CPU_i$ **then**
11: $E\_CPU_i++$
12: **end if**
13: **else**
14: **if** $E\_CPU_i > LOWER\_CPU_i$ **then**
15: $E\_CPU_i--$
16: **end if**
17: **end if**
18: Reset update timer and sleep for $t$ period
19: **until** Container $i$ is terminated

**Algorithm 2** The calculation of effective memory

1: **Variables:** Container $i$'s hard memory limit $L_i^{hard}$ and soft memory limit $L_i^{soft}$; system-wide current free memory $c_{free}$; system-wide free memory in previous update interval $p_{free}$; container $i$'s current memory usage $c_i^{mem}$ and previous memory usage $p_i^{mem}$.
2: **Output:** The effective memory size $E\_MEM_i$.
3: $E\_MEM_i = L_i^{soft}$
4: `/* Expand effective memory until there is a system-wide memory shortage */`
5: **if** $c_{free} > LOW\_MARK$ **then**
6: **if** $\frac{c_i^{mem}}{E\_MEM_i} > 90\%$ and $E\_MEM_i < L_i^{hard}$ **then**
7: $\Delta_i^{mem} = (L_i^{hard} - E\_MEM_i) \cdot 10\%$
8: $\Delta_{predict}^{mem} = \frac{p_{free} - c_{free}}{c_i^{mem} - p_i^{mem}} \cdot \Delta_i^{mem}$
9: **if** $(c_{free} - \Delta_{predict}^{mem}) > HIGH\_MARK$ **then**
10: $E\_MEM_i += \Delta_i^{mem}$
11: **end if**
12: **end if**
13: **else** `/* Reset effective memory if reclaiming memory */`
14: $E\_MEM_i = L_i^{soft}$
15: **end if**

a container, if other constraints (CPU affinity $\mathbb{M}$ and limit $l$) permit. The lower bound on effective CPU count is rounded up to the closest integer (line 4). The upper bound is the minimum of $|\mathbb{M}|$ and $\frac{l}{t}$, with the latter equivalent to $\frac{cpu.cfs\_quota\_us}{cpu.cfs\_period\_us}$ (line 5). The lower and upper bounds are updated by the `ns_monitor` upon container creation/deletion or changes to cgroups settings. Otherwise, they remain unchanged throughout the life time of a container.

Second, we dynamically adjust effective CPU in the range of [LOWR_CPU,UPPER_CPU] according to the CPU usage of a container. As most modern OSes are work-conserving, a container is allowed to use more than its fair share of CPU if the system has slack CPU ($p_{slack}$) that would otherwise be left idle. We gradually increase the number of effective CPU (line 9-11) if the actual CPU usage of a container $u_i$ (in terms of CPU cycles) is close to or higher than the capacity of effective CPU. Given each updating period $t$, the total available CPU time on effective CPU is $E\_CPU_i \cdot t$. If the ratio of a container's CPU usage to the capacity of effective CPU is larger than a threshold ($UTIL\_THRSHD$), effective CPU is incremented by 1. We empirically set $UTIL\_THRSHD$ to 95%. In contrast, if the system has no idle CPU, decrement effective CPU by 1 at each `sys_namespace` update period until reaching the lower bound (line 14-15). Dynamically adjusting effective CPU to reflect the actual CPU allocation of a container is important for user space applications to control their concurrency. For example, decrementing effective CPU until there is slack CPU allows containers to individually reduce concurrency and agree on an optimal degree of concurrency. Changes to effective CPU are limited to 1 per update to prevent abrupt fluctuations.

**Effective memory** reflects the actual memory usage of a container as well as the memory limits enforced by cgroups. `Sys_namespace` records the soft ($L^{soft}$) and hard limits ($L^{hard}$) of memory, i.e., `soft_limit_in_bytes` and `limit_in_bytes` and periodically updates effective memory when the update timer fires. A container can use more memory than the soft limit if there is free memory in the system but will be killed or trigger swapping when exceeding the hard limit. Whenever there is a system memory shortage, the portion of memory beyond the soft limit will be reclaimed, causing swapping. Unlike managing effective CPU, in which the CFS scheduler enforces fair sharing among containers and over-subscription usually results in graceful slowdowns, over-committing memory can cause memory thrashing and performance collapse. Therefore, individual containers' effective memory should be carefully managed to avoid swapping or thrashing.

`Kswapd` is a kernel daemon responsible for managing swap space in response to memory pressure. It uses three watermarks to track memory pressure. When the number of free pages in the system falls below `low_watermark`, `kswapd` reclaims memory from containers that exceed their soft limits until system-wide free memory reaches `high_watermark`. If free memory further drops below `min_watermark`, `kswapd` performs direct reclaiming, which indiscriminately frees memory from any containers, including those do not exceed their soft limits. As shown in Algorithm 2, effective memory is initialized to a container's soft memory limit (line 3). If there is no memory shortage (line 5) and a container uses more than 90% of its effective memory, the container is allowed to gradually expand its effective memory, at an increment of 10% towards its hard limit (line 7). To prevent memory thrashing or swapping, we need to ensure that the increment of individual containers' effective memory does not trigger `kswapd` for memory swapping. We use the ratio of system-wide change of free memory to the change of memory usage of a container in a previous round to estimate

the change in system-wide free memory given the change in container effective memory $\Delta_i^{mem}$ in this round (line 8). Note that this could be an over-estimation of the change of system-wide free memory as the container may not use up all the additional effective memory. If the predicted change of free memory does not bring system-wide free memory below `high_watermark`, the container's effective memory is incremented by $\Delta_i^{mem}$ (line 10). Otherwise, the container's effective memory remains unchanged. This algorithm aligns with the design of `kswapd` and treats `high_watermark` as the threshold for sufficient free memory. Whenever, system memory is in shortage and `kswapd` is reclaiming memory, reset a container's effective memory to its soft limit (line 14).

## 3.2 Implementation

In a conventional OS, applications probe system resources through system-wide `procfs` or `sysfs`. To display effective resources in containerized applications, we intercept system calls to `procfs` or `sysfs` and test if the calls are from an ordinary process or from within a container. Containerized processes are within their own namespaces and thus have links from their `task_struct` to these namespaces. In contrast, ordinary Linux processes are in the namespaces of the `init` process, the parent of all processes. Therefore, when a process probes system resources and is linked to its own namespaces other than the `init` namespaces, a virtual `sysfs` is created for this process. Future queries issued by this process to the `sysfs` will be redirected to the newly created virtual `sysfs` and the calculated effective resources will be returned.

Containers, such as docker, are ephemeral by design. At launch, a container is assigned with a per-container `init` process to set up the namespaces, including the added new `sys_namespace`. After the environment is set up, the `init` process calls `exec` to start the process specified by command `docker run`, after which the original `init` process is terminated and the process started by the `exec` system call becomes the new `init` process of the container. Any processes later forked in the container will inherit the namespaces, but the namespaces can only be accessed from within the container because the owner, i.e., the original `init` process, has been terminated. As `sys_namespace` needs to be periodically updated by the OS, a key issue is to access the per-container `sys_namespace` from outside of a container. To address this issue, we transfer the ownership of `sys_namespace` to the new `init` process, which will be alive throughout the life of a container. Specifically, we modify the `execve` system call to change the ownership of `sys_namespace` to the *current* task when the state of the original `init` process changes to `TASK_DEAD`. This ensures that the new owner is the new `init` process of the container and all tasks created later share the same cgroups settings.

`Ns_monitor` is implemented as a system-wide kernel thread. We modify the source code of cgroups to invoke `ns_monitor` if a `sys_namespace` exists for a control group and there is a change to the cgroups settings. The `sys_namespace` update timer is an original low-resolution timer and its update interval is set to the scheduling period in Linux, during which all tasks are guaranteed to run at least once. This ensures that any changes to the CPU allocation of containers are immediately reflected in `sys_namespace`. In Linux CFS, the length of a scheduling period depends on the number of ready tasks. When there are no more than 8 tasks, the scheduling period is set to 24*ms*. Otherwise, the period is set to 3*ms*×num _of_tasks. Since the change of memory usage is less frequent than that of CPU allocation, we use the same update interval of effective CPU in calculating effective memory.

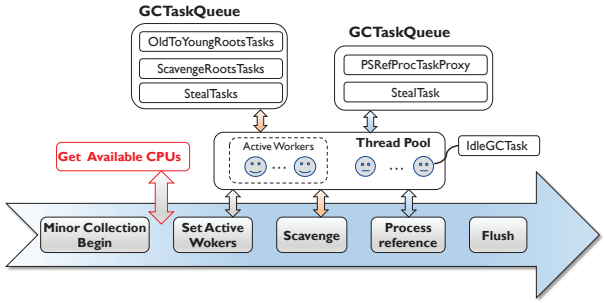



**Figure 4: Parallel GC with dynamic parallelism**

# 4 CASE STUDIES: BUILDING ELASTIC APPS

In this section, we demonstrate how to leverage the per-container resource view to enable elasticity in some representative applications. Although requiring changes to application source code, enabling elasticity offers two advantages: 1) applications are able to use additional resources when they are available; 2) applications can more efficiently utilize resources when sharing with others.

## 4.1 Dynamic Parallelism

We begin with dynamically controlling the degree of parallelism in multi-threaded programs in response to varying CPU allocation. In general, launching a variable number of threads during runtime is very challenging. Therefore, we focus on applications that employ a thread pool to dynamically assign work to threads. As such, altering the number of worker threads does not affect the completion of the program.

**HotSpot** is a widely adopted JVM that use multiple threads to perform parallel *garbage collection* (GC) on the heap space. The number of GC threads created when launching the JVM is determined by the number of CPUs in the system. HotSpot uses a fairly sophisticated design for parallel GC. It implements a centralized `GCTaskQueue`, from where individual GC threads fetch GC tasks. This design is key to enabling dynamic work assignment, which allows faster GC threads to fetch more tasks. `GCTaskQueue` is protected by `GCTaskManager`, a monitor construct that not only enforces mutual exclusive access to the queue but also provides a condition variable to synchronize GC threads. When the `GCTaskQueue` is empty, i.e., the completion of a GC, all GC threads are put to sleep. They are woken up when the next GC starts and `GCTaskQueue` is refilled with GC tasks. This design allows a variable number of GC threads to be activated/woken up at each GC.

Figure 4 shows how to integrate effective CPU into the process of parallel GC. There are two types of GC, minor and major GC, which perform a partial and full heap scan, respectively. For illustration, we only show the process of a minor GC while major GCs can be modified in a similar way to use dynamic parallelism. As shown in

the figure, at the beginning of each GC, the parallel garbage collector, i.e., *parallel scavenge* (PS), first determines the number of GC threads to activate. The existing design, called *dynamic GC threads*, calculates the number of active threads based on the number of Java application threads (i.e., mutators) and the size of the heap. Assume the total number of GC threads created when launching the JVM is $N$ and the number of calculated active threads is $N_{active}$, the actual threads woken up at each GC is $N_{gc} = \min(N, N_{active})$. As discussed in Section 2.2, the Java community has recognized the issue, and starting JDK 9, sets $N$ according to a container's CPU limit rather than the number of online CPUs in the host. While this design incorporates container awareness into JDK, it has two limitations that hamper elasticity. First, the JVM cannot launch more GC threads if the container's CPU limit is lifted and more CPUs are available. Second, static CPU limit and share do not accurately reflect a container's CPU allocation, thereby unable to fully exploit elasticity.

To truly enable elasticity, we make two changes to the PS collector in HotSpot. First, we launch as many GC threads as possible according to the number of online CPUs, retaining the potential to expand the JVM with more CPUs. Second, we instrument the PS collector to query the number of effective CPUs to dynamically adjust the number of GC threads. Specifically, we use formula $N_{gc} = \min(N, N\_active, E\_CPU)$ to adjust GC threads, where $E\_CPU$ is the effective CPU count read from the `sys_namespace`.

**OpenMP** provides a mechanism to alter thread-level parallelism at runtime. If a user enables dynamic threads in OpenMP, a variable number of threads will be created for parallel constructs, depending on the number of online CPUs and system load. `gomp_dynamic_max_threads` calculates the number of threads using $n\_onln - loadavg$, where $n\_onln$ is the number of online CPUs and $load\_avg$ is the average load, in terms of running tasks, of the system in a 15-minute period. Our change to OpenMP is straightforward. We substitute $n\_onln$ with $E\_CPU$ and remove the second term of the formula as effective CPU already includes load information at a much finer granularity.

## 4.2 Elastic Heap

Making memory allocation elastic is more challenging. Therefore, we focus on enabling elastic memory for runtime systems that are already equipped with dynamic memory allocation. Heap management in the HotSpot JVM employs an adaptive sizing algorithm to dynamically change the size of the JVM heap. Users specify an initial heap size (`-Xms`) and a maximum heap size (`-Xmx`) at JVM launch time. The adaptive sizing algorithm automatically determines an appropriate heap size based on feedbacks from completed GCs. However, the sizing algorithm cannot expand the heap beyond the maximum heap size set at JVM launch time or shrink the heap in response to memory pressure in a container.

To enable elastic heap, it is necessary to understand the structure of JVM heap to properly expand or shrink the heap while still retaining the properties of the original design. We follow the discussions on Java heap in [6] to divide heap memory into three states: *used*, *committed*, and *reserved*. Used space is where objects (live or dead) are stored; committed space is a super set of used memory and may include memory that is allocated to the JVM but is currently free;

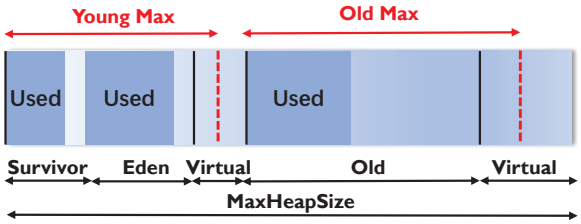


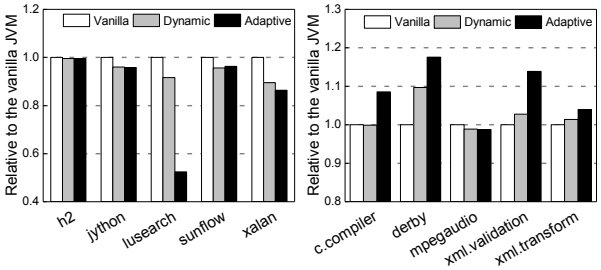

**Figure 5: Realizing elastic heap in HotSpot**

reserved space, similar to a virtual address space, is the maximum expandable heap size, in which memory beyond the committed space is not yet allocated. Scaling JVM heap is essentially to adjust the committed space. When additional memory is available, the committed space should be able to expand beyond the limit of a previous reserved space while able to free memory and shrink itself if the reserved size falls below the committed size.

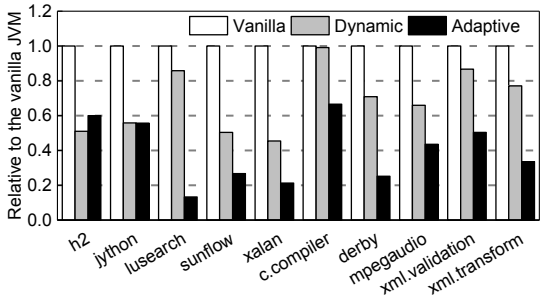
The *parallel scavenge* (PS) collector, the default garbage collector in JDK 8, is a generational collector that divides the heap into *young*, *old*, and *permanent* generations. The young generation is where new objects are first allocated and a memory allocation failure in this area triggers a minor GC. If objects survive a predefined number of minor GCs, they are promoted to the old generation, which is larger than the young generation and holds long-living objects. The collection of the old generation, which is called a major GC, is less frequently performed. The sizes of the young and old generations are determined by the adaptive sizing algorithm. The JVM maintains a fixed ratio of 1:2 between the sizes of the young and old generations. The permanent generation contains JVM metadata and thus does not participate in size adjustment.

The sizing algorithm maintains two invariants: 1) the expansion of the heap should not exceed the reserved size, a static limit set at JVM launch time; 2) the shrinkage of the heap should preserve the size ratio of generations. The sizing algorithm validates a proposed new size to be smaller than the reserved size. Inspired by Bruno et al.,'s work on vertical heap scaling [6], we decouple the sizing algorithm from the static reserved size by adding a new, dynamic size limit `VirtualMax` on the heap. By setting the original reserved size `MaxHeapSize` to a sufficiently large value, close to the size of physical memory, and dynamically adjusting `VirtualMax` to realize elasticity. In addition, we impose two new limits `YoungMax` and `OldMax` on the young and old generation, respectively, to retain the size ratio.

Intuitively, we use effective memory from the `sys_namespace` as `VirtualMax` and set `YoungMax` and `OldMax` accordingly. Heap expansion is straightforward. We simply set `VirtualMax` to a larger value and rely on the adaptive sizing algorithm to gradually expand the heap. In contrast, heap shrinkage is challenging. Figure 5 shows the structure of JVM heap. `MaxHeapSize` is the original static upper bound, close to physical memory size. The spaces between solid vertical black lines are the committed spaces of the young and old generations. The dotted vertical red lines indicate the current `YoungMax` and `OldMax`. When effective memory drops, the dotted lines need to be moved to the right and there exist three scenarios: 1) If no dotted red lines cross solid black lines, i.e., the change in




(a) DaCapo execution time (b) SPECjvm2008 throughput




(c) DaCapo and SPECjvm2008 GC time

**Figure 6: Performance comparison on DaCapo execution time (lower is better) and SPECjvm2008 throughput (higher is better)**

effective memory not affecting the committed sizes of young and old generations, only the values of `YoungMax` and `OldMax` need to be changed; 2) if red lines cross black lines but do not touch used spaces, the sizing algorithm is instructed to shrink the committed sizes in addition to changing `YoungMax` and `OldMax`; 3) if red lines cross used spaces, the corresponding type of GCs are invoked to free the used spaces. If a single GC may not be able to free enough space, we invoke GCs every 10s until success. We query `sys_namespace` every 10s and perform the adjustment if needed. Note that the elastic heap management only deals with the size limits and is independent from the original sizing algorithm, thereby applicable to other dynamic Java heap management schemes.

# 5 EVALUATION

In this section, we present an evaluation of the adaptive resource view in containers. Because we did not observe measurable overhead associated with creating and updating `sys_namespace`, our focus is on evaluating the performance improvement due to dynamic parallelism and elastic heap.

## 5.1 Experimental Settings

**Hardware.** Our experiments were performed on a PowerEdge R730 server, which was equipped with dual 10-core Intel Xeon 2.30 GHz processors, 128GB memory, and a 1TB SATA hard drive.

**Software.** We used CentOS 7 64bit and Linux kernel version 4.12.3 as the host OS. Docker 17.06.1 was used as the containers technology. Experiments were conducted on OpenJDK with Parallel Scavenge as the garbage collector and OpenMP in gcc 4.8.

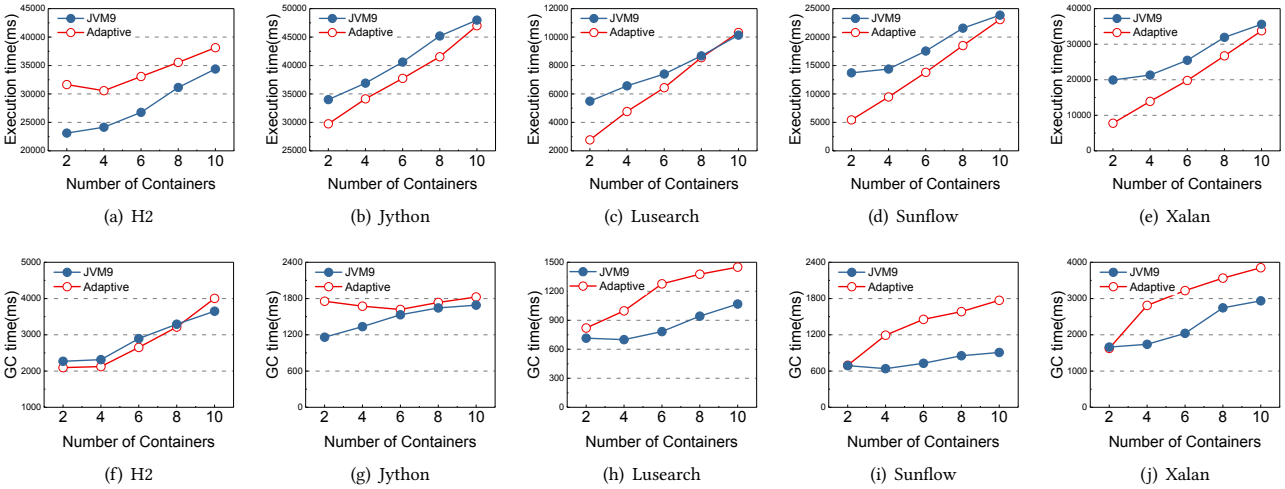
**Benchmarks and methodology.** We selected Dacapo [7], SPECjvm2008 [28], and HiBench [14] as the workloads to evaluate the optimized JVM, and used *NAS parallel benchmarks* (NPB) [21] to evaluate OpenMP with dynamic parallelism. The heap sizes of Java-based benchmarks were set to 3x of their respective minimum heap sizes. Each result was the average of 10 runs. Performance comparison was made between the original JVM and OpenMP (denoted as *vanilla*) and the optimized versions that use the adaptive resource view (denoted as *adaptive*). In addition, we evaluate the effectiveness of the existing dynamic parallelism schemes in HotSpot and OpenMP (denoted as *dynamic*) and the recently released JVMs with container awareness (e.g., JVM9 and JVM10).

## 5.2 Results on Dynamic Parallelism

We begin with a well-tuned environment with five containers running five copies of the same Java benchmark. We empirically determined that five benchmarks sharing a total number of 20 cores, each with four GC threads, achieved the best performance. We are interested in if effective CPU can lead to an optimal number of GC threads and how is its performance compared to the vanilla JVM. We used OpenJDK 1.8 as the JVM as it is not incorporated with container awareness. Figure 6 shows the performance of *DaCapo* and *SPECjvm2008* benchmarks due to different JVMs. The adaptive JVM with effective CPU clearly outperformed the vanilla JVM by as much as 49% in *DaCapo* and by 18% in *SPECjvm2008*, respectively. The existing dynamic GC threads scheme in JVM improved performance over the vanilla JVM with static GC threads. Since it imposes a minimum amount of work for a GC thread to process, it effectively mitigated over-threading. However, as the number of containers increases, its performance gain would diminish because the increasing competition from other containers does not affect the heap size to GC thread ratio. As shown in Figure 6(c), most performance gain was due to improved GC time.

JDK 9 is the first container-aware Java runtime. Before launching a JVM, it detects if there is a CPU mask associated with the Java process, a typical way to limit CPU allocation to a container using CPU affinity. If CPU affinity is found, the JDK calculates the number of CPUs the JVM is permitted to access and uses this count to determine the number of GC threads. We configured the CPU mask to access two cores in each container and varied the number of co-running containers from 2 to 10. Dynamic threads was enabled in the JVM. As shown in Figure 7, our adaptive approach based on effective resources outperformed the container-aware JVM9 in all benchmarks. Note that the overall performance gain (Figure 7(a) - (e)) diminished as the number of containers increased. However, Figure 7 (f) - (j) show that GC performance due to *adaptive* was worse than JVM9 with a fixed number of GC threads and the gap widened as more containers were added except in *Jython*.

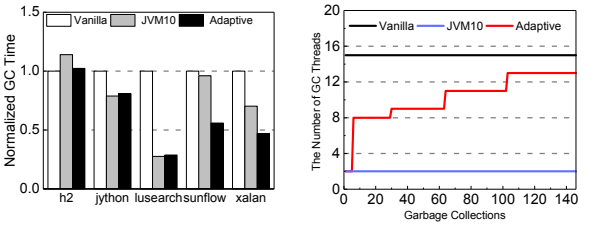
This presents a difficult trade-off between performance isolation and resource elasticity. On the one hand, CPU affinity in JVM 9 restricts the benchmarks, including their application threads, to use a few cores, which led to inferior overall performance. On the




(a) H2 (b) Jython (c) Lusearch (d) Sunflow (e) Xalan

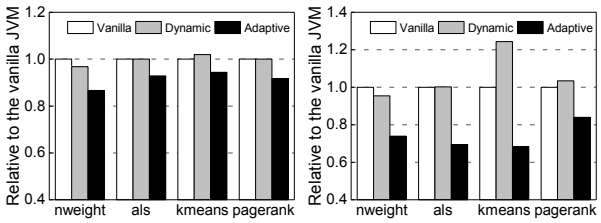
(f) H2 (g) Jython (h) Lusearch (i) Sunflow (j) Xalan

**Figure 7: DaCapo performance due to static CPU limit and effective CPU with a varying number of containers**




(a) GC time. (b) Number of GC threads in *sunflow*

**Figure 8: DaCapo performance due to static CPU shares and effective CPU with varying CPU availability**



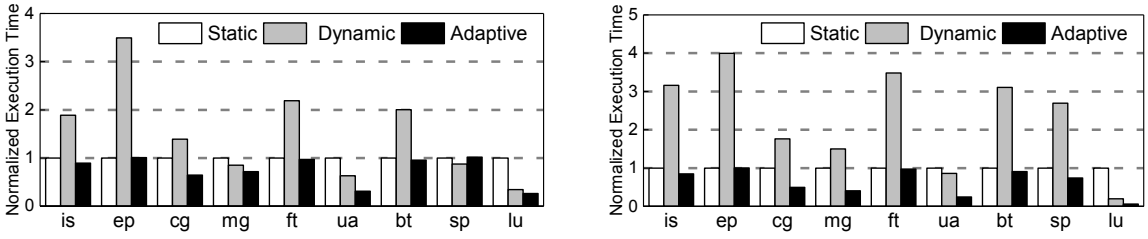

(a) Execution Time (b) GC Time

**Figure 9: Performance improvement on big data applications with large datasets**

other hand, CPU affinity provides isolation between GC threads and application threads. In comparison, the adaptive approach based on effective CPUs faced performance interference from co-running benchmarks. In addition, elastic GC threads do not necessarily lead to improved performance if GC is not scalable to a large number of threads.

JVM 10 further improves container awareness by considering CPU shares of a container. If CPU limit is not present, JVM 10 uses an algorithm similar to that in Algorithm 1 (line 4) to derive a core count based on CPU share, and uses this static core count throughout a JVM's life time. However, JVM 10's algorithm based on static CPU shares does not allow a container to dynamically adjust its CPU count according to the actual CPU usage in the host. We create a realistic scenario in which a mixture of containers with different CPU usages are collocated. Specifically, we collocated ten containers, each with an equal CPU share, on the same host. One container ran a Dacapo benchmark and the remaining nine containers ran different sysbench benchmarks. The host CPU was fully utilized when all ten containers were running benchmarks but CPU availability varied as different sysbench benchmarks completed at different times. Based on static CPU shares, JVM 10 limited the number of GC threads to 2 even when other containers became idle. The vanilla JVM (i.e., JVM 8 and earlier) configured 15 GC threads throughout the test, according to the number of online CPUs in the host. In contrast, our adaptive JVM varied the number of GC threads based on effective CPUs. Figure 8 (a) shows that the adaptive JVM outperforms JVM 10 in GC performance for most cases, by as much as 42%. Compared with the vanilla JVM, container awareness in JVM 10 led to significant reduction in GC time. For some benchmarks with short completion time (e.g., *jython* and *lusearch*), the performance of *adaptive* was slightly worse than that of JVM 10 because there was not enough time for *adaptive* to adjust concurrency in GC. Figure 8 (b) plots the number of GC threads throughout the execution of *sunflow*, which benefits from the *adaptive* JVM. As shown in the figure, *adaptive* adjusted the number of GC threads as some collocated sysbench benchmarks freed their CPU allocations.

**Big data applications** While *DaCapo* and *SPECjvm2008* are widely adopted Java benchmarks, they are limited by their input sizes and require only small heap sizes. Therefore, not all such small-scale workloads benefit from the adaptive resource views as GC may not be scalable to a large thread count. Realistic Java-based workloads, such as big data processing frameworks, require much




(a) Five containers with equal shares.

(b) One container with a quota equivalent to 4 cores.

**Figure 10: Performance comparison of OpenMP applications of NPB due to static, dynamic and adaptive threads**

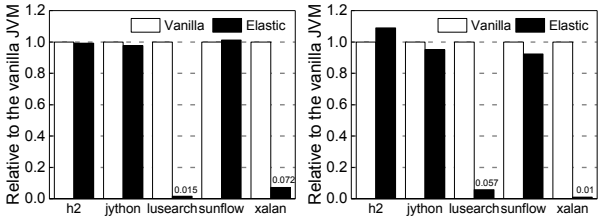
larger heap sizes. Figure 9 shows the results with four applications in the *HiBench* benchmark. Because HiBench is not compatible with JDK 9 or 10, the baseline is *vanilla* JDK 8 with no container awareness. We incorporated container awareness into JDK 8 and enabled dynamic threads (denoted as *dynamic*). The results on the overall execution time and GC time suggest that the adaptive approach consistently outperformed the baseline and the one based on static cgroups settings.

**OpenMP applications** OpenMP provides another perspective on dynamic parallelism in runtime systems. Unlike JVM, which creates all threads at launch time, OpenMP creates threads when a parallel region is executed. The *static* strategy launches the same number of threads, matching the number of online CPUs [1], for all parallel regions. The *dynamic* strategy uses formula $n\_onln - loadavg$ to determine thread counts for each parallel region. Figure 10 shows the performance comparison in two scenarios. We first co-ran 5 containers with equal shares, each running an identical NPB program. In the second test, we ran one NPB program in one container and assigned the container a CPU quota equivalent to 4 cores. This limits the container to use up to 4 CPUs. Surprisingly, the *dynamic* approach had the worst performance in both scenarios. In the five-containers experiment, the high system-wide load limits the number of threads in containers, even they each are guaranteed a fair share. In the one-container test, *dynamic* launched a large number of threads into a 4-CPU container. Both misconfigurations led to substantial performance degradation. Our *adaptive* approach achieved significantly better performance compared with *static* or *dynamic*.

## 5.3 Results on Elastic Heap

The goal of devising elastic heap management is to utilize additional memory when it is available without being constrained by manually set memory limits and to gracefully scale up and down heap size in a shared environment without invoking memory thrashing or swapping. We use *DaCapo* benchmarks and a micro-benchmark to evaluate our elastic heap management in HotSpot. We first created a container with a 1GB hard memory limit. If an application uses more than 1GB of memory, it triggers swapping and will likely cause a performance collapse. We started *DaCapo* benchmarks with an initial heap size of 500MB without a maximum heap size. This

[1]Users may specify the number of threads by setting environment variable `OMP_NUM_THREADS`.



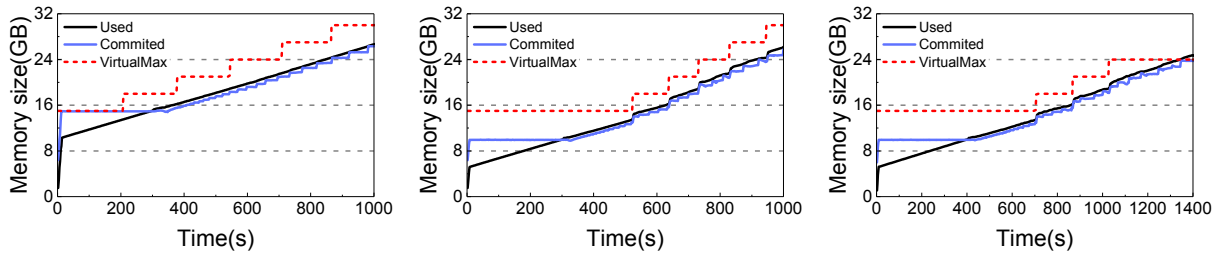

(a) Execution Time

(b) GC Time

**Figure 11: Avoiding memory overcommitment in DaCapo**

allows the JVM to automatically set the maximum heap size to one quarter of the physical memory size, i.e., 32GB, a much higher value than the hard limit. Figure 11 shows benchmark completion time and GC time under the vanilla JVM (JDK 8) and the optimized JVM with elastic heap. Note that JDK 9 and 10 detect the 1GB hard limit and automatically set the maximum heap size to one quarter of the hard limit, i.e., 250MB. Most *DaCapo* applications crash with out-of-memory (OOM) errors. As shown in the figures, elastic heap did not offer any benefit to applications that did not use more than 1GB memory, such as *h2*, *jython*, and *sunflow*. The vanilla JVM incurred performance collapse due to swapping for applications that exceeded the hard limit. In contrast, our elastic heap was aware of the hard limit and never allowed the heap to grow beyond the limit, though at a cost of more frequent GCs. The performance due to elastic heap is an order of magnitude better than that in the vanilla JVM.

Second, we wrote a Java micro-benchmark with controlled memory demand. The benchmark iterates for 40,000 times and at each iteration allocates 1MB objects and deallocates 512KB objects in the JVM heap. This creates an ever-increasing heap space with half capacity storing "dead" objects. The benchmark results in a working set size of 20GB while touching at most 40GB memory space. We created a single container with a 30GB hard limit and 15 GB soft limit. The JVM used was from JDK 10 with awareness on memory limits. Figure 12 (a)-(b) show the trend of `used`, `committed`, and `VirtualMax` (i.e., dynamic `reserved` space) memory when executing the micro-benchmark. Note that `VirtualMax`, i.e., the red dotted line, is set according to effective memory but not used by the vanilla JVM. In Figure 12 (a), the vanilla JVM initialized the




(a) Single container with vanilla JVM (b) Single container with elastic JVM (c) Five containers with elastic JVM

**Figure 12: The statistics of used, committed, and reserved memory in the vanilla JVM and elastic JVM**

heap to be one quarter of the hard limit, i.e., around 10GB, and the adaptive sizing algorithm quickly expanded the heap. Accordingly, `VirtualMax` was increased. In Figure 12 (b), the elastic JVM started with a smaller initial heap size (around 10GB, one quarter of the initial `VirtualMax`) and ramped up slowly until the used space approached to 90% of effective memory (`VirtualMax`). Eventually, both the vanilla JVM and the elastic JVM converged to the hard limit. Figure 12 (c) shows the results when five such containers are collocated, each running the micro-benchmark. Since each container touches 40GB, all of them will eventually reach the 30GB hard memory limit, causing memory thrashing as the aggregate demand exceeding the physical memory size. Therefore, the vanilla JVM failed to complete any of the micro-benchmarks. In comparison, the elastic JVM was able to complete all benchmarks and determined an appropriate heap size (i.e., 24GB) for each container.

## 5.4 Overhead

There are two sources of overhead associated with the adaptive resource views. First, the update timer periodically fires to re-calculate effective CPU and memory in each container's `sys_namespace`. On our testbed, the update to a `sys_namespace` takes 1 $\mu s$. No additional synchronization or locking is needed for the updates – 1) each update is done along with the existing scheduler bookkeeping routines when a scheduling period ends; 2) no locking is enforced between the updater and queries from the containers. Second, there is a cost each time a container queries its virtual `sysfs` for effective CPU and memory. From user space, the time to call `sysconf` and query effective CPU and memory takes 5 $\mu s$ and 100 $\mu s$, respectively. Reading effective memory is more expensive because it involves querying multiple files in `sysinfo`. Since the adjustment to the number of threads only occurs when there is a change to CPU allocation (every 24 *ms* to 100 *ms* in Linux) and memory adjustment is less frequently due to data loading and freeing, we do not anticipate the overheads to contribute much in the overall performance.

# 6 RELATED WORK

There are recent works focusing on evaluating the performance and isolation of containers compared with other virtualization techniques [10, 26, 27]. Sharma et al., [26] analyzed the performance in containers and VMs due to resource overcommitment and ineffective isolation. They found that performance degradation is more significant in containers than that in VMs due to the weak isolation. To this end, approaches have been proposed to enhance isolation or add container awareness into the OS or application stack. Arnautov et al., [3] proposed to run containers inside SGX enclaves. Khalid et al., [16] proposed a mechanism for accounting packet processing in the OS kernel to provide isolation between different types of containers. Zhang et al., [32] recognized a similar semantic gap and incorporated container awareness into an MPI library.

Felter et al.,[10] first discovered that the lack of knowledge on resource limits incurs performance degradation in containerized applications. The JDK community has recently released a series of container optimizations based on static resource limits and shares. However, it takes a tremendous effort for individual applications to incorporate container awareness. In this work, we proposed a per-container view on resource allocations and developed a virtual `sysfs` interface to seamlessly connect with user space applications without requiring any source code changes.

There is a large body of work dedicated to dynamic parallelism and elastic memory management. As observed in many applications [18, 24], oversubscription incurs substantial performance degradation due to various reasons [15]. Therefore, searching for the optimal *degree of parallelism* (DoP) and optimizing core allocations have been an active research topic [13]. Atachiants et al., [4] pointed out the difficulties in matching program parallelism to platform parallelism in a multiprogrammed and multi-tenant environment. There are studies [12, 15, 24] focusing on dynamically changing DoP according to system load, and some are in the area of Java garbage collection [5, 6, 11, 22, 29].

Memory overcommitment is a form of elastic memory management and has attracted much attention in literature. Memory oversubscription has been studied in virtualized environments [2, 8, 19, 25, 30] and there are various memory ballooning mechanisms [17] proposed to dynamically adjust VM memory allocations without incurring major performance overhead. Nakazawa et al., [20] analyzed the performance degradation of containers when memory is overcommitted and proposed techniques to mitigate it by dynamically identifying heavily loaded containers and tuning their swappiness to prevent them from being swapped out. Many studies have shown that swapping out memory pages belonging to a heap area of a *Java virtual machine* (JVM) degrades the performance significantly [31]. In this work, we leveraged adaptive resource views to allow JVM to make more informed decisions on heap sizing. Various heap sizing approaches have been proposed [31]. Our

proposed elastic heap management does not rely on specific sizing algorithms and is complementary to the existing approaches.

## 7 CONCLUSION

In this paper, we discovered that the weak isolation between containers sharing the same OS kernel has led to an important semantic gap in resource management in containerized systems. The illusion of being able to access all resources available on a physical host while being constrained by resource limits and shares, presents challenges to containerized applications in determining an appropriate degree of thread-level parallelism and a proper memory size. We have demonstrated that a significant portion of applications on DockerHub are susceptible to the semantic gap. To bridge this gap, we proposed a per-container view of resources. The central design is a `sys_namespace` for each container that reports real-time resource allocations of a container based on resource limits, shares, and the actual usage from other containers. We further develop a virtual `sysfs` interface to seamlessly connect `sys_namespace` with user space applications. Through two case studies, we showe that a reasonable amount of engineering effort is sufficient to enable dynamic parallelism and elastic heap for Java and OpenMP applications.

## 8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. This research is supported by National Key Research and Development Program under grant 2018YFB1003600, National Science Foundation of China under grant No.61732010. The first author also gratefully acknowledges financial support by the *China Scholarship Council* (CSC). The corresponding authors are Song Wu and Jia Rao.

## REFERENCES

[1] 2018. *LXCFS.* https://linuxcontainers.org/lxcfs/.
[2] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: a memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 349–366.
[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 689–703.
[4] Roman Atachiants, Gavin Doherty, and David Gregg. 2016. Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation. *IEEE Transactions on Software Engineering (TSE)* 42 (2016), 764–785.
[5] Stephen M. Blackburn and Kathryn S. Mckinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32.
[6] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. 2018. Dynamic vertical memory scalability for OpenJDK cloud applications. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 59–70.
[7] DaCapo. 2009. *DaCapo Benchmarks.* http://dacapobench.org/.
[8] Williams Dan, Hani Jamjoom, Yew Huey Liu, and Hakim Weatherspoon. 2011. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 205–216.
[9] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. 2014. Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 73–84.
[10] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS)*. 171–172.
[11] Lokesh Gidra, Julien Sopena, and Marc Shapiro. 2013. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 229–240.
[12] Md E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. Mckinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 161–175.
[13] Wim Heirman, Trevor E. Carlson, Kenzo Van Craeynest, Ibrahim Hur, Aamer Jaleel, and Lieven Eeckhout. 2014. Undersubscribed threading on clustered cache architectures. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 678–689.
[14] Hibench. 2019. *Hibench.* https://github.com/intel-hadoop/HiBench.
[15] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2013. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 155–168.
[16] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating Network-based CPU in Container Environments. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 313–328.
[17] Jinchun Kim, Viacheslav Fedorov, Paul V. Gratz, and A. L. Narasimha Reddy. 2015. Dynamic Memory Pressure Aware Ballooning. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 103–112.
[18] Jihye Kwon, Kang-Wook Kim, Sangyoun Paik, Jihwa Lee, and Chang-Gun Lee. 2015. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 232–244.
[19] Pin Lu and Kai Shen. 2007. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 29–43.
[20] Rina Nakazawa, Kazunori Ogata, Seetharami Seelam, and Tamiya Onodera. 2017. Taming Performance Degradation of Containers in the Case of Extreme Memory Overcommitment. In *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD)*. 196–204.
[21] NPB. 2019. *NAS Parallel Benchmarks.* https://www.nas.nasa.gov/.
[22] Junjie Qian, Witawas Srisaan, Sharad Seth, Hong Jiang, Du Li, and Pan Yi. 2016. Exploiting FIFO Scheduler to Improve Parallel Garbage Collection Performance. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 109–121.
[23] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 145–160.
[24] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 26–37.
[25] Tudor Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. 2013. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 337–350.
[26] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference (Middleware)*. 1–13.
[27] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. 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 (EuroSys)*. 275–287.
[28] SPECjvm. 2008. *SPECjvm2008 Benchmarks.* https://www.spec.org/jvm2008/.
[29] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 35:1–35:15.
[30] Jingjing Wang and Magdalena Balazinska. 2017. Elastic memory management for cloud data analytics. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 745–758.
[31] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J Eliot B Moss. 2006. CRAMM: Virtual Memory Support for Garbage-collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 103–116.
[32] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High Performance MPI Library for Container-Based HPC Cloud on InfiniBand Clusters. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. 268–277.