CHARACTERIZATION OF CONTAINERS IN EMERGING APPLICATIONS: MICROSERVICES, FAAS AND GPUS

BY

HALDUN UMUR DARBAZ

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Nam Sung Kim

ABSTRACT

Containers have enabled new computing paradigms such as Functions-as-a-Service in data centers today. Containers are inherently more lightweight than virtual machines. This is caused by the fact that containers share the kernel with the host system, removing the need for a two-dimensional page walk. Containers also do not require a hypervisor. They rely on thin management layers in container frameworks and existing Linux functionality. Linux process and resource management features such as cgroups and namespaces are tightly integrated to containers. This allows for simple management and isolation of containerized applications.

Docker is currently the most prominent container framework. This thesis utilizes Docker containers to create data center use cases with databases, web servers, graph analytics, Functions-as-a-Service, and GPU-accelerated stencil, lower-upper decomposition, object tracking and neural network applications. Furthermore, this thesis analyzes Docker Engine performance by bringing up containers and breaks down bring-up overheads at function granularity. The virtual memory management aspects of Docker containers are also characterized with a focus on container infrastructure, page tables and page faults. This thesis reports on average 59.86% duplicated page table entries and 35.7% duplicated page faults across four containerized processes sharing a core. Additionally, this thesis identifies the source of 40% of container bring-up overhead and attributes it to memory allocation, garbage collection and process creation in Go and Linux. This thesis also identifies a 7% slowdown in containerized GPU applications with NVIDIA-Docker compared to native execution. Finally, this thesis provides guidance to architects for enabling container support in high-performance architectures, and identifies future work to be done in the area.

ACKNOWLEDGMENTS

I would like to express my gratitude to Professor Nam Sung Kim at the University of Illinois at Urbana-Champaign for his unending support beginning in my undergraduate years. My passion and drive for computer architecture grew under his guidance, without which this thesis would not have been written. His knowledge and impeccable mentorship were paramount during my graduate studies.

Additionally, I would like to sincerely thank Professor Josep Torrellas, my supervisors Dimitrios Skarlatos and Mohammad Alian at the University of Illinois at Urbana-Champaign. Collaborating with them has given me direction and a stronger understanding of the existing problems in computing systems. Their guidance and critique of my work have greatly contributed to my development as a researcher.

I was able to connect my academic work to state-of-the-art industry problems and engineering challenges through the support of Nathan Myers at Apple. The trust placed in me by him and his team has vastly expanded my understanding of computation and challenges at scale. Through novel graphics processing units work, I learned to be a better engineer and I greatly appreciate our time together.

Finally, I would like to thank all of my close friends and family. Those who have been with me every step of the way are the reason why I enjoy what I do today. Nourishing relationships and environments allowed me to focus on my studies and growth, even through the toughest times. Their persistent belief in my prospective success filled me with confidence. I am forever grateful for them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
VM	Virtual Machine
FaaS	Function-as-a-Service
ML	Machine Learning
DB	Database
TLB	Translation Lookaside Buffer
MMU	Memory Management Unit
PGD	Page Global Directory
PUD	Page Upper Directory
PMD	Page Middle Directory
PT	Page Table
PTE	Page Table Entry
THP	Transparent Huge Page
ASLR	Address Space Layout Randomization
CoW	Copy-on-Write
NVMe	Non-Volatile Memory Express
SSD	Solid State Drive
I/O	Input/Output

CHAPTER 1

INTRODUCTION

1.1 Cloud Computing

Cloud computing is one of the most dominant paradigms in the modern field of computing. The advent of scalable and high-performance architectures has enabled the evolution of online platforms and services. Data-driven and distributed applications continue their rapid evolution in data center environments today. A key technology utilized by data centers is virtualization. Virtualization is the backbone of application delivery, isolation and resource sharing for multiple tenants in a cloud environment. Traditionally, virtual machines (VM) have been utilized to provide cloud services where each tenant would be provided with a VM for their allocated compute time.

Recently, *containers* have fundamentally altered the way cloud computing functions. Containers package an application together with its dependencies, necessary libraries and system software. Containers share the host system's kernel instead of running a guest operating system on top of the host. The inherently lightweight design of containers allows data centers to better utilize their infrastructure and have more tenants per hardware, with examples of 10,000 containers per system [1]. The new application paradigms enabled by containers and the scale pose challenges in terms of virtual memory management, application scheduling and performance guarantees.

1.2 Service Providers and Applications

Major cloud providers have enabled containerized services. Amazon Web Services with ECS [2], IBM Cloud [3], Google Cloud [4] and Microsoft Azure [5] with the Kubernetes Engine [6] are some examples of container services.

One major use case of Docker-based virtualization is classical data servicing and processing applications such as databases and graph analytics. However, Docker has enabled new paradigms in data centers. At the forefront of these new paradigms is Serverless Architectures through Functions-as-a-Service (FaaS). FaaS eliminates long-running services in favor of short-lived instances of applications to execute shorter and simpler functionality such as data queries or basic processing. This allows for "pay-as-you-go" models and function-granularity scaling. FaaS is now available in major cloud providers such as AWS through Lambda [7], IBM with OpenWhisk [8] and Google with Cloud Functions [9].

Finally, accelerators and heterogeneous architectures have taken hold of cloud computing. NVIDIA provides support for containerized GPU applications by extending Docker into NVIDIA-Docker [10]. GPU-enabled Docker containers allow for the packaging of the CUDA runtime along with a Docker image in order to execute GPU compute workloads within containers.

1.3 Thesis Focus

Containers are inherently different from VMs in their use-cases, infrastructure and characteristics. An application that needs to scale worker processes to match incoming traffic can scale with containers. This is referred to as on-demand scaling. Containerized environments are scaled by bringing up or tearing down containers. As a result, the virtual memory management and container management performance are critical to low latency and scale. This thesis approaches containers by investigating Docker, the prevalent container framework in a use case oriented study. A thorough analysis of virtual memory management of Docker containers is conducted with applications commonly found in data centers. Furthermore, on-demand scaling performance is investigated and a breakdown of per-function overheads of creating and starting a container is provided.

The remainder of this thesis is organized as follows. First, the necessary background regarding Docker, containers, process and resource management, address translation, translation lookaside buffers, page tables, huge pages and active page management in Linux are provided in Chapter 2. Next, containers are compared to virtual machines and the major performance and

memory overhead differences are highlighted in Chapter 3. Afterwards, the experiment methodology is presented in Chapter 4. Performance and virtual memory characterization of Docker containers are then presented in Chapter 5. The related literature on containers, TLBs, address translation and page table sharing in architecture and hardware along with software approaches are reviewed in Chapter 6. Finally, the thesis reaches a conclusion about the strengths and weaknesses of containers in Chapter 7 and finishes with guidance for future architectural work in this area.

CHAPTER 2

BACKGROUND

This chapter introduces what containers are and their infrastructure, and discusses container management in Linux using Docker. It goes into detail regarding container images, processes, resource management and extensions to Docker. Afterwards, it also discusses the necessary CPU architecture and operating systems background needed to understand the analysis done later in the thesis, with a focus on TLBs, address translation and page tables in Intel x86-64.

2.1 Docker and Containers

2.1.1 Overview

Containers are a lighweight virtualization technology used for process isolation, resource management, application packaging and delivery. Containers are built upon several major Linux kernel features such as *namespaces*, *control groups* (cgroups) and *filesystems* (UnionFS for Docker). Two major container projects today are *Linux Containers* (LXC) [11] and *Docker* [12]. This thesis characterizes Docker containers specifically.

2.1.2 Container Images

Container images are the core of the packaged application. The container image consists of multiple read-only layers, along with a thin read-write layer added upon launching a container. The read-only layers contain application code and necessary libraries along with system support. These layers package together the application and all of its dependencies. The thin read-write layer is utilized during application runtime for any temporary data storage

the application needs. Image creation works by either creating a base image or extending from other images. A common way to construct an application image is by extending from a Linux image to include fundamental system support. Then, applications can extend the Linux image by adding their specific libraries and unique layers to create their own image.

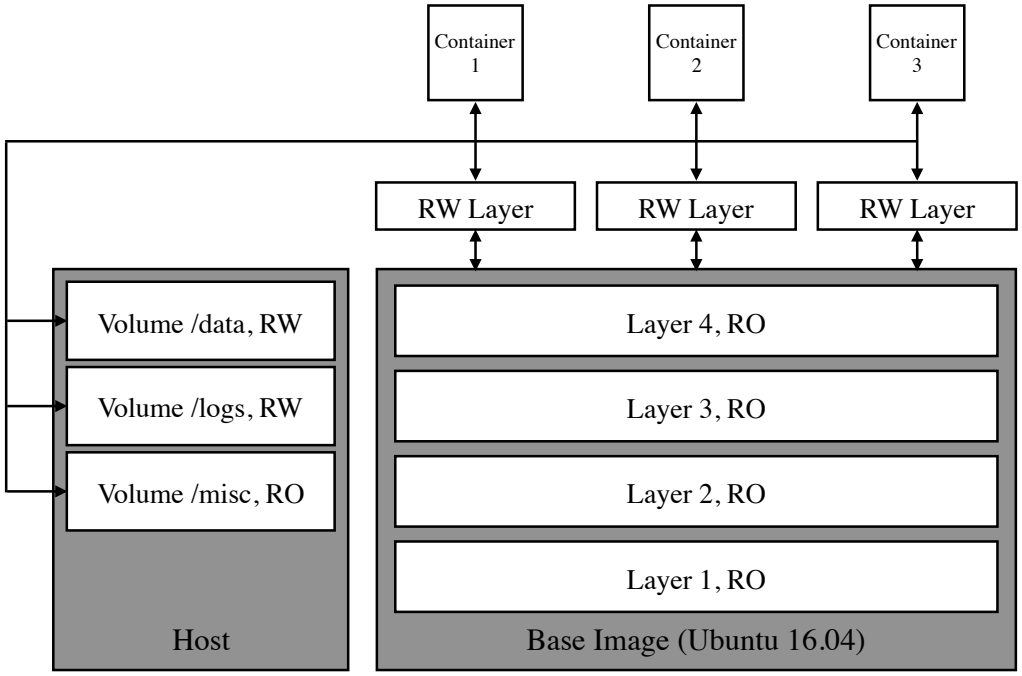

Figure 2.1: Container Image Structure

An example container image can be seen in Figure 2.1. All three containers share the same image and have their own private read/write layers. Additionally, the data, logs and misc directories of the host machine are mounted as volumes to the three containers. The mounted volumes are shared and visible across the containers.

2.1.3 Container Processes and Resource Management

A container instance is created from a container image. The Docker Engine [13] is the hypervisor replacement for Docker containers. It handles the start-up of container processes along with necessary process and resource management. A container can be created and started by sending the Docker Engine container creation and start commands. Once started, there will be

two processes visible to the host. Namely, the *containerd-shim* managing the container runtime, and the application process. The containerized application will run on top of the Docker Engine. Multiple container applications running on the same host machine will share the host kernel, as seen in Figure 2.2.

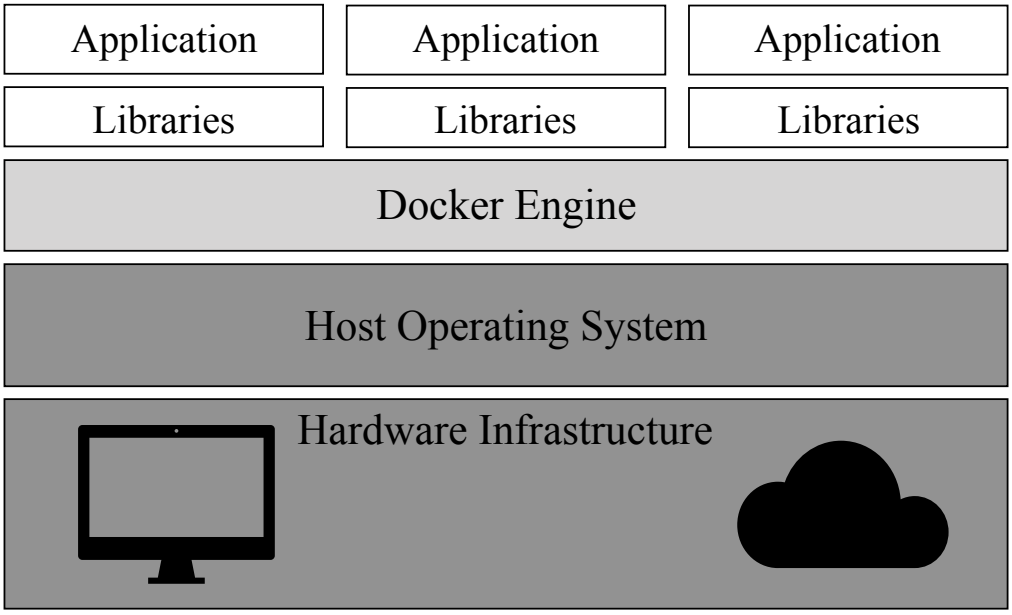

Figure 2.2: Docker Container Software Stack

Containers are stateless by nature. A container is stopped and torn down after its execution. Any container state does not persist beyond the execution of the application within the container. However, data persistence is an option through the use of *volumes*. Any directory on the host machine can be mounted as a volume to a container process during the start-up of a container. The mounted volume can be utilized to persist information from the container runtime and share data with other containers.

Resource management is handled through cgroups. Control groups allow for limiting CPU, network, I/O and memory resources allocated to a given process. Docker integrates cgroups into the Docker Engine to manage and restrict container process resources.

Isolation is maintained through the use of namespaces. In Linux, each namespace sees its own allocated set of resources. Docker provides a namespace to each container to isolate containerized processes. It also allows for

packing different containers into the same namespace.

2.1.4 NVIDIA-Docker

NVIDIA-Docker brings CUDA support to containerized environments by packaging the CUDA runtime with GPU applications and their dependencies in the container image. NVIDIA-Docker containers are driver-agnostic and they automatically mount the necessary driver and GPU device components at bring-up [14].

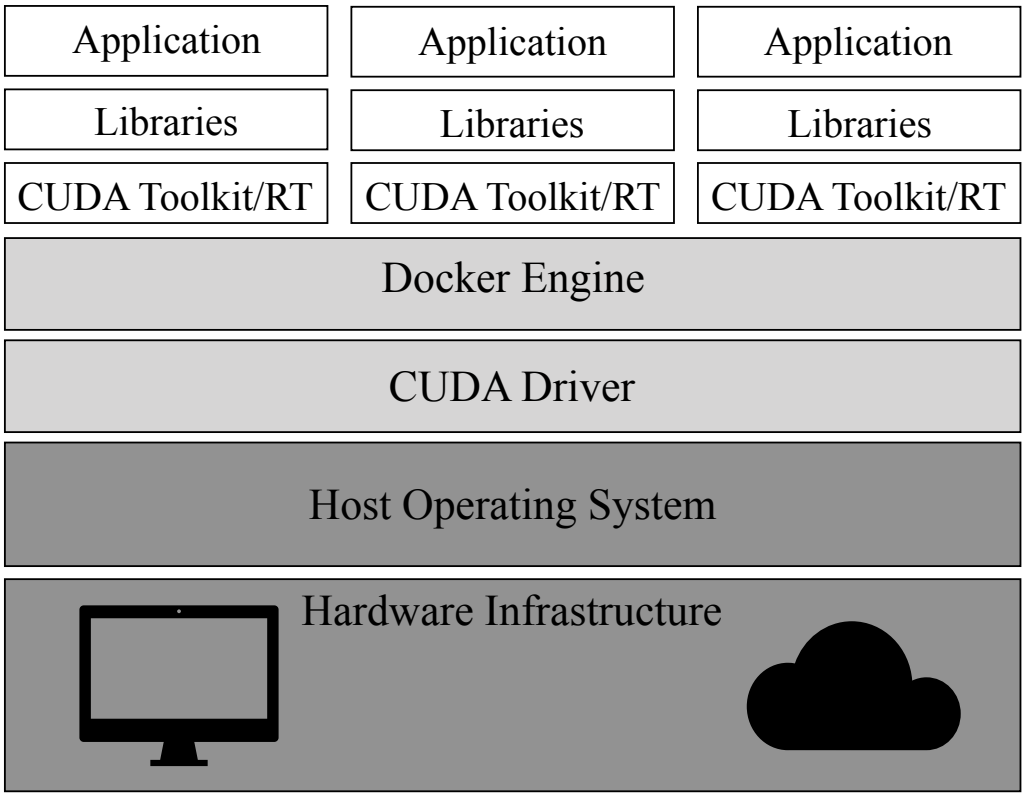

Figure 2.3: NVIDIA-Docker Container Software Stack

The NVIDIA-Docker container stack is shown in Figure 2.3. NVIDIA-Docker is fundamentally a wrapper around Docker. GPU accelerated containers have the CUDA toolkit and runtime built in to the image. Each NVIDIA-Docker container shares the host GPU driver, similar to the way containers share the host kernel.

2.2 Address Translation, TLBs and Page Tables

In architectures utilizing virtual memory, each memory access has to be translated from a virtual address to a physical address. Translation Lookaside Buffers (TLBs) cache translations that are used frequently. In a conventional TLB structure, each entry holds the virtual page number (VPN), the corresponding physical page/frame number (PPN/PFN), along with page flags, a unique process identifier (PCID) and a valid bit. A common TLB organization is shown in Figure 2.4. Intel TLB hierarchies today are comprised of split L1-TLBs for instructions and data along with a unified L2-TLB [15].

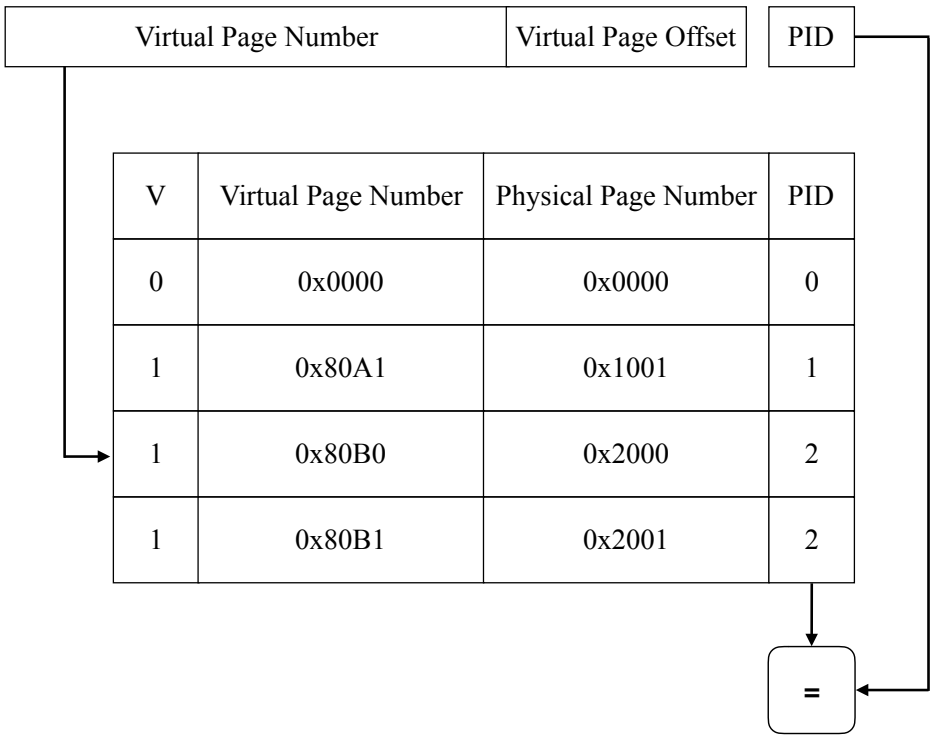

Figure 2.4: Process ID Tagged TLB Organization

When a memory access happens, the TLBs are indexed with the virtual address of the current access. If the L1 TLB misses, the L2 TLB is checked. If the address translation is found in the TLBs, the translation is utilized in the rest of the memory hierarchy. A *page table walk*, handled by the memory management unit (MMU), begins when the last-level TLB access misses. This process is shown in Figure 2.5 with 4-level page tables. Intel has future plans to extend this scheme to 5-level page tables [16]. In Intel

x86-64, the MMU first reads CR3 to get the physical address of the currently running process's last level of the page table, the Page Global Directory (PGD). Bits 47:39 of the virtual address are used to index into the PGD. This step provides the MMU with the physical location of the Page Upper Directory (PUD). Bits 38:30 of the virtual address are then used to access the PUD, yielding the physical location of the Page Middle Directory (PMD). The PMD is accessed with bits 29:21 of the virtual address, which results in the base address of the Page Table (PT). The final 21 bits of the virtual address are used for the PT index (20:12) and the page offset (11:0) for 4 KiB pages. The final access is done to find the Page Table Entry (PTE). If a PTE is found at the final location, the MMU successfully returns the physical translation to the processor and caches the translation.

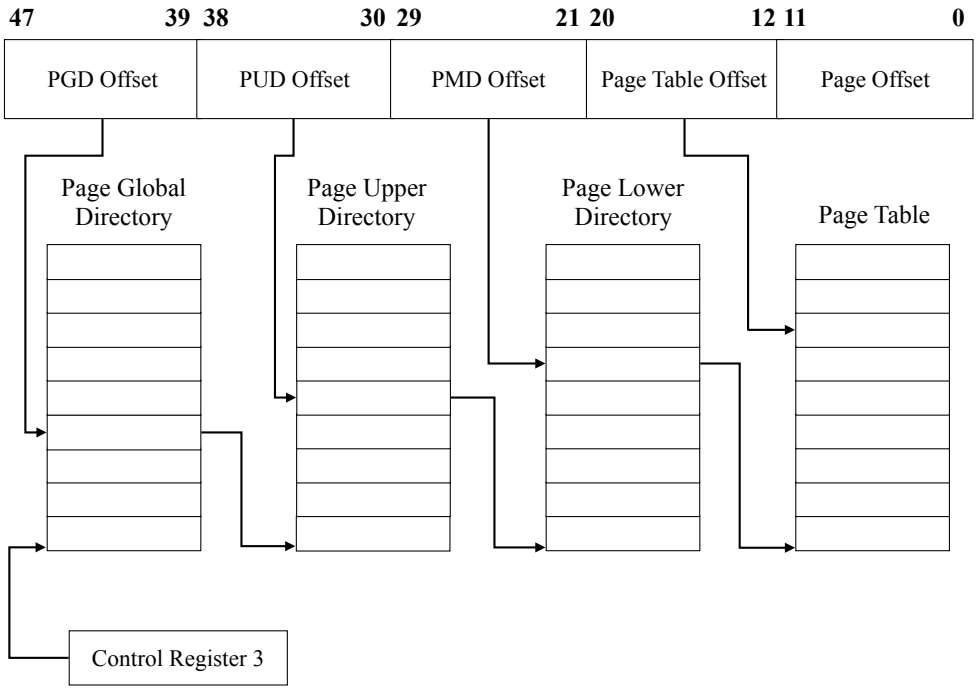

Figure 2.5: Multi-Level Page Table Hierarchy

When a PTE is not present at the accessed location, a *page fault* is triggered. There are two types of page faults: soft/minor page faults and hard/major page faults. The operating system is involved in resolving page faults through the page fault handler.

If the PTE that is not present at the walked location is present somewhere

else in memory (mapped to another process), then this is considered a soft page fault. Soft page faults occur when multiple processes share physical pages (such as libraries) despite using separate page tables. Hard page faults occur on the very first access to a page across all running processes. The page fault handler provides a mapping to the currently page faulting process and the missing PTE is populated. If the PTE is not present in memory at all, then this is a hard page fault. This begins an I/O request to the disk in order to bring the requested page into memory and creates a PTE for it at the location accessed by the faulting process.

2.3 Transparent Huge Pages

Scaling out the amount of managed memory becomes costly and challenging with small pages due to the number of page table entries present across page tables and within the TLB. Large numbers of PTEs across different processes exert high levels of TLB pressure and require more page walks. In order to deal with this problem, Linux utilizes *huge pages* [17]. Modern processor architectures and operating systems support multiple page sizes, namely 4 kB, 2 MB and 1 GB. Larger page sizes such as 2 MB and 1 GB are collectively referred to as huge pages. A huge page combines smaller 4 kB pages underneath a larger collective. Huge pages can be manually allocated by users, and they will not be swapped out to disk during memory pressure.

Due to difficulties in managing huge pages, *Transparent Huge Pages* (THP) [18] is used today. THP is abstracts away the management of huge pages and automates it in the kernel. Huge pages are created and destroyed by promoting and demoting contiguous chunks of pages. A kernel task periodically scans page tables to find a section of memory where a huge page can replace smaller pages. The decision to promote a set of pages to a huge page depends on an internal threshold. If a contiguous chunk of pages surpass the internal threshold, they may be promoted to a huge page. When the allocated pages within a huge page fall below the threshold, the huge page may be broken up.

2.4 Page LRU Lists

The Linux kernel maintains two least recently used (LRU) lists as part of its page frame reclamation scheme [19]. These are the *active* and *inactive* lists. The LRU lists are maintained to keep track of which pages are least recently accessed or least frequently accessed. The information kept in the LRU lists is used when the kernel decides to reap the Page Cache. Allocated pages start out in the inactive list. When a page is accessed for the first time, its referenced bit is set. The second access to a page moves it to the active list and clears the referenced bit. Third and future accesses to a page set the referenced bit again. This process is seen in Figure 2.6.

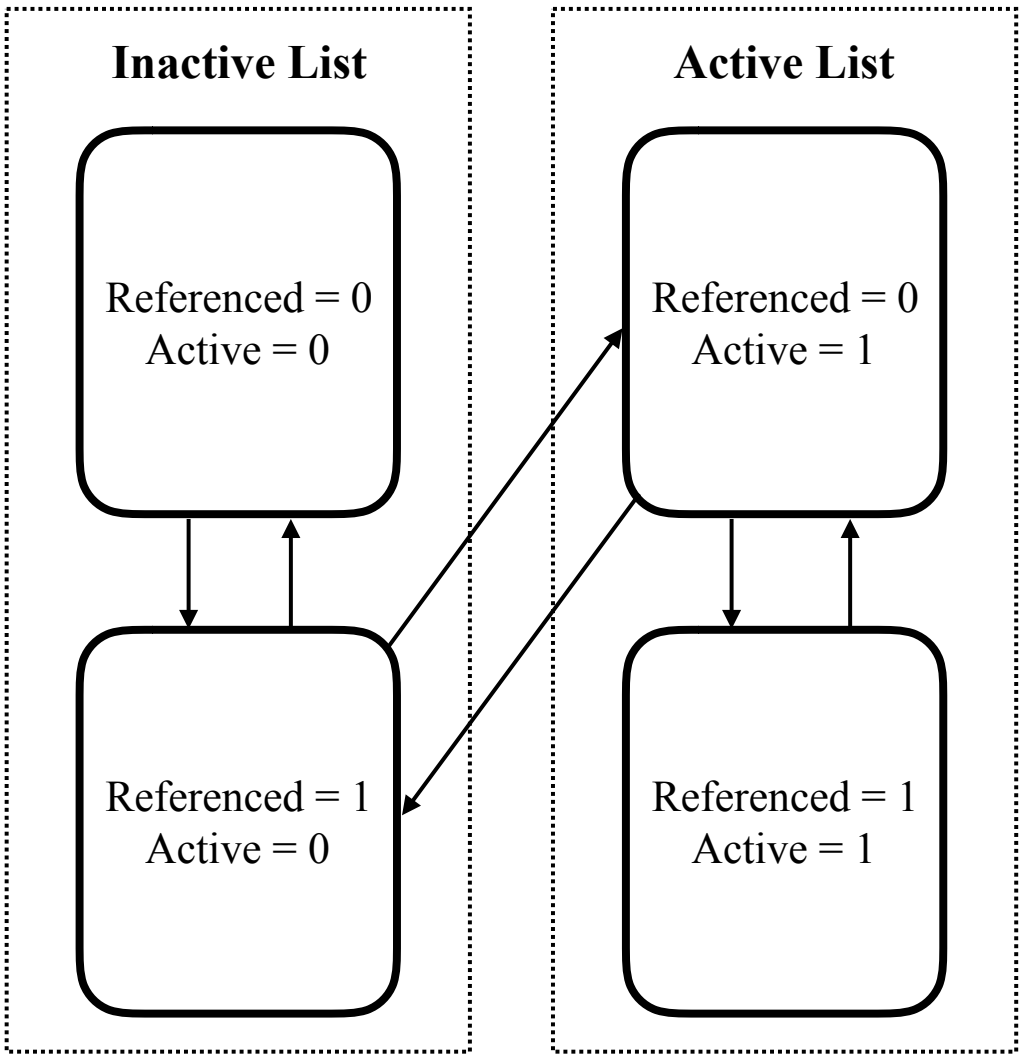

Figure 2.6: Page LRU List Flow

When the kernel runs low on Page Cache space, it pushes the tail of the active list to the inactive list. Pages in the inactive list are candidates for reclamation. A page on the active list is inferred as frequently and recently accessed, whereas those on the inactive list are accessed sporadically.

CHAPTER 3

CONSIDERATIONS

This chapter introduces the software stack of virtual machines (VM) along with two-dimensional page walks and address translation in fully virtualized environments. It highlights the drawbacks associated with virtual machines, and explains how Docker containers attempt to overcome the aforementioned challenges. Finally, a literature study is conducted to contrast the performance and memory overheads of virtual machines against those of Docker containers.

3.1 Virtual Machines

3.1.1 Software Stack

Virtual machines work with hardware virtualization. Hardware virtualization requires running a hypervisor such as VMWare ESXi [20], KVM [21] or Xen [22]. The hypervisor is a software layer that virtualizes the underlying hardware and enables resource management for multiple virtual machines. Each virtual machine has a guest operating system underneath the applications it virtualizes. The full software stack is seen in Figure 3.1.

3.1.2 Address Translation

Address translation works with nested pages in virtual machines. The first set of page tables are those managed by the guest operating system. The second set of page tables are the ones maintained by the host operating system. Since the memory address space for the guest operating system is also virtualized, each memory access has to be translated to a virtual address in the host system, and eventually a physical address in the host system.

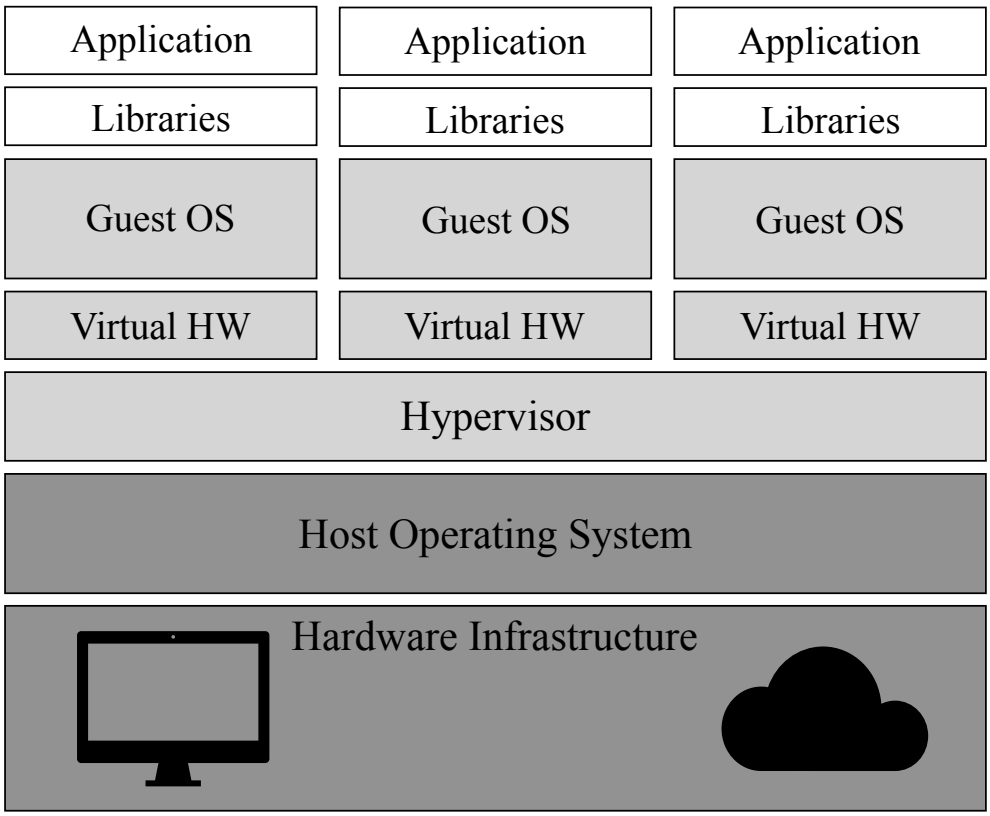

Figure 3.1: Virtual Machine Software Stack

In a bare metal environment with 4-level page tables, a page walk would first access the page global directory, then the page upper directory, followed by the page middle directory and finally the page table. A page walk in a virtual machine requires the access to each level of the guest page table to be translated. This is referred to as a two-dimensional page walk. Figure 3.2 visualizes a two-dimensional page walk. This results in 24 memory accesses, which is a major cause for performance slowdown in virtual machines [23]. Containers share the kernel with the host, thereby removing the need for a two dimensional page walk.

3.1.3 Performance Compared to Containers

The hardware virtualization and heavier software stack of virtual machines cause them to be inherently slower than containers. Previous work by Sharma

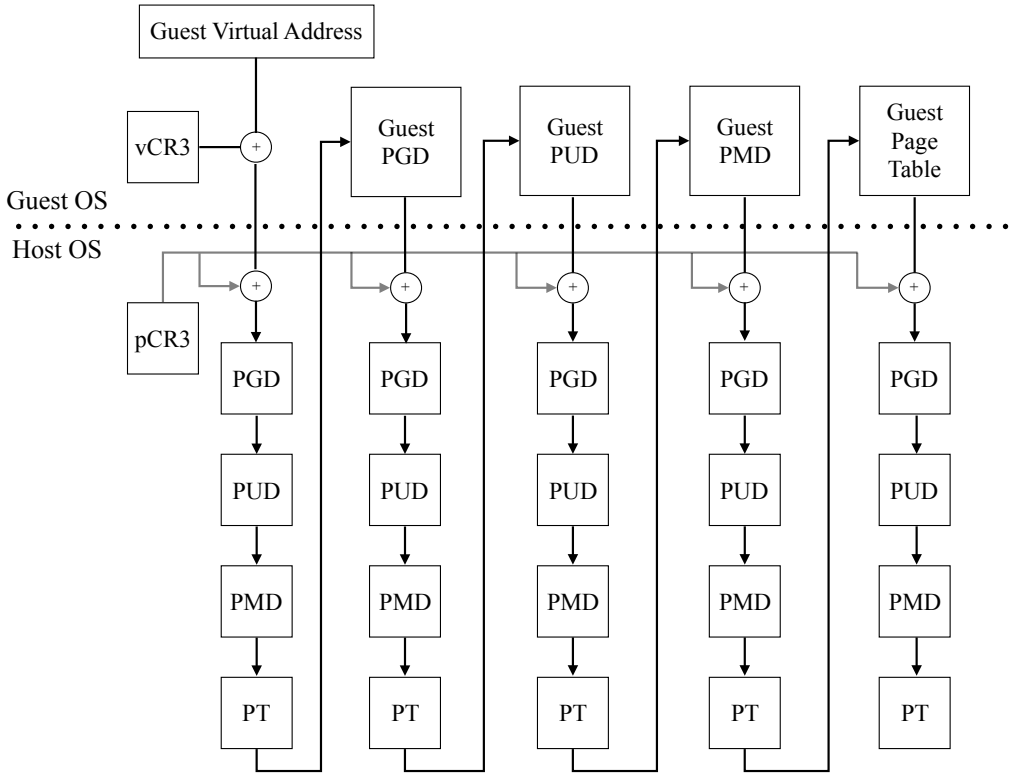

Figure 3.2: Address Translation in Virtual Machines

et al. [24] shows that containers can use up to 89.5% less memory across their set of applications. The authors also observe that soft-limited containers have up to 40% higher throughput than virtual machines.

Another study by Soltesz et al. [25] uses Linux-VServer [26] to represent a containerized environment. Their evaluation is conducted with operating system benchmarks (fork/exec/sh/ctx/mmap/page fault). VServer performance is within 1% of native performance, whereas Xen performed approximately 3x slower than the native baseline.

Furthermore, IBM Research has also evaluated Docker container performance against virtual machines [27]. The authors show that containers are within 4% range of native execution performance. Baseline kernel-based virtual machines (KVM) are up to 22% slower than native performance, whereas tuned virtual machines are at most 18% slower than native execution. One of their key findings is that network performance is considerably slower in containerized environments. Round trip latency is nearly twice the native

round trip time with both TCP and UDP protocols. Virtual machines have lower round trip latency and lower cycles/byte transmission time than containers. Another critical observation IBM Research makes is that the I/O performance of virtual machines is sluggish in random read/writes. Their measurements show two to three times higher read latency in kernel-based virtual machines. On the other hand, Docker containers have native level I/O performance with both sequential and random read/writes. Finally, the authors show that using the container filesystem causes a 1.5% decrease in MySQL transaction throughput whereas using volumes avoids the performance loss.

Overall, past research shows that container performance is near-native level. While the performance is dependent on the exact configuration of the container, containers consistently display higher performance than virtual machines. The only exception shown thus far is the network performance of Docker. This thesis does not consider virtual machines any further, and focuses on a case-study oriented characterization of containers directly.

3.2 GPU Translation Lookaside Buffers

CUDA programs follow a single-program multiple-data (SPMD) model where programs are developed similarly to scalar applications that execute with a program order. The SPMD model uses single-instruction multiple-data at the hardware level, where execution resources are wide with warp-level scheduling. In NVIDIA architectures, 32 threads are scheduled and executed concurrently in each pipeline [28]. Because of the execution model, the memory hierarchy organization is also effected to exploit the coalescing of memory accesses from a warp.

Virtual memory is used in today's highly programmable GPUs [29]. This allows better separation and permissions of address spaces of different programs executing on a GPU and effectively enable resource sharing. The Volta memory hierarchy is virtually indexed at the L1 caches and physically indexed at the L2 cache. The TLBs use 2 MB page table entries where the L1 TLB has 32 MB of coverage and the L2 TLB has 8192 MB of coverage [30]. The increasing importance of scientific and high-performance computing for data analytics, machine learning and scientific applications has ensured the

deployment of GPUs in cloud providers' infrastructure. As a result, GPU TLB performance has gained prominence.

Previous work on GPU memory hierarchies has shown the performance degradation of using a naive address translation hierarchy modeled after CPU TLBs and page walkers. According to [31], using a naive address translation hierarchy where TLBs are accessed before or in parallel to L1 caches causes performance degradation of 20-50%. The overhead is attributed to the challenges posed by the execution model, where a stalling thread will cause the entire warp to stop, to the coalescing requirements, and to timing problems due to massively parallel hardware. The authors of [31] propose exploiting intra-warp cache locality and introducing smarter warp scheduling by adding TLB-awareness to reduce the performance overheads of virtual memory to CPU levels.

This thesis does not further consider GPU memory hierarchies. The GPU accelerated services experimentation covers the direct overhead of containerization against native execution at the kernel runtime and API call level, as described in the later Chapters 4 and 5.

CHAPTER 4

METHODOLOGY

This chapter discusses the experiment methodology used to characterize Docker containers. It first details the overall setup and the evaluation systems. Then, it explains the focus of the analysis. Finally, the evaluated applications are discussed.

4.1 Experiment Setup

4.1.1 Evaluation Systems

This thesis performs its experimental evaluation and performance measurements using real systems. Three application categories are evaluated in this thesis, namely: *Microservices*, *Functions-as-a-Service* and *GPU-Accelerated Services*. Microservices and FaaS experiments are conducted on the physical machine described in Table 4.1. GPU-accelerated applications are evaluated on the system described in Table 4.2.

4.1.2 Characterization

Containers have been extensively studied to characterize CPU performance, networking and I/O compared to native execution and virtual machines. This thesis focuses on the characterization of three aspects of Docker containers that have not yet been explored.

Container Bring-up Performance: A per-function performance breakdown of bringing up containers is obtained and discussed using perf [32]. The breakdown exposes the cost of different functions that are used by the Docker Engine and by the Linux kernel to identify bottlenecks in starting up a container. The container bring-up time is also measured by stressing the Docker

Table 4.1: Microservices and FaaS Evaluation System

Processor Parameters	
CPU Model	Broadwell Xeon E5-2650v4 2-Socket
Core Count and Frequency	24 Cores, 2.2 GHz
L1 Caches	768 kB, 8-ways, 64-byte lines
L2 Cache	3 MB, 8-ways, 64-byte lines, Unified
L3 Cache	30 MB, 20-ways, 64-byte lines, Unified
L1 dTLB	64 entries (4kB), 4 entries (1GB), 4-ways
L1 iTLB	128 entries (4kB), 8 entries (2MB), 8-ways
L2 TLB	1536 entries, 6-ways, Unified
Main-Memory Parameters	
Capacity, Channels	64 GB, 2
Ranks/Channel	2
Banks/Rank	8
Frequency, Data Rate	2133 MHz, DDR4
Host and Docker Parameters	
Host OS	Ubuntu Xenial 16.04
Kernel	4.4.0
Docker Engine	17.06 CE

engine with varying request rates from a client. The client is a Go application that hooks to the Docker API to send container star requests and measures the bring-up time.

Virtual Memory Management Characterization: Because containers are a virtualization technology, the virtual memory management performance of containers is critical to their success. The address translation characteristics of containerized applications are intricately analyzed with a focus on page table entries and page faults to identify potential room for improvement. Large numbers of containers exist on data center machines. That is due to the fact that containers are inherently more lightweight and scalable than virtual machines. To effectively recreate a data center oriented container use case, four containerized processes are placed on a core to accomplish the memory characterization.

NVIDIA-Docker GPU Performance Compared to Native Execution: The performance overheads of containerization in traditional CPU workloads has been well explored. As such, this thesis investigates the performance of containerized GPU applications with NVIDIA-Docker and compares it to native execution. The composition of execution time is constructed and the

Table 4.2: GPU Accelerated Services Evaluation System

Processor Parameters	
CPU Model	Haswell Core-i7 4790
Core Count and Frequency	4 Cores, 3.6 GHz
L1 Caches	256 kB, 8-ways
L2 Cache	1 MB, 8-ways, Unified
L3 Cache	8 MB, 16-ways, Unified
L1 dTLB	64 entries (4kB), 4 entries (1GB), 4-ways
L1 iTLB	64 entries (4kB), 8 entries (2MB), 8-ways
L2 TLB	1024 entries, 8-ways, Unified
Device Parameters	
GPU Model	Pascal GTX 1050
Streaming Multiprocessors	10
CUDA Cores	64 Per SM
Frequency	1.45 GHz
L2 Cache	1 MB
Memory, Interface and BW	2 GB, GDDR5, 112 GB/s
Main-Memory Parameters	
Capacity, Channels	16 GB, 1
Ranks/Channel	2
Banks/Rank	8
Frequency, Data Rate	1.6 GHz, DDR3
Host and Docker Parameters	
Host OS	Ubuntu Xenial 16.04
Kernel	4.8.0
Docker Engine	18.06 CE
CUDA and Compute	8.0, 6.1
NVIDIA Driver	384.130

differences between containerized and native execution are highlighted with a single application per core.

4.2 Applications

4.2.1 Microservices

Microservices are a form of software architecture that integrates multiple small, modular components into a larger application [33]. Each modular

component accomplishes a set of functions and is an autonomous entity. As part of a larger application, it can form any component such as a front-end, data-servicing back-end or a computation engine. The decentralized structure of microservices eases deployment in data center environments, and as a result they have proven to be a strong target application class to be used with Docker containers. This thesis profiles Docker with microservices using the following application sets obtained from the Docker Hub [34].

Data-Serving: Databases and web servers form the core of data-servicing applications. To evaluate data-serving workloads, MongoDB [35], ArangoDB [36] and httpd [37] are used. MongoDB is a document-model NoSQL distributed database that utilizes memory-mapped storage. It is useful for real-time analytics, content management and mobile data serving. ArangoDB is a multi-model NoSQL database supporting key-value pairs, documents and graphs in memory-mapped files. It is widely used in graph analytics and has applications in cryptographic key management. Httpd is Apache's open-source HTTP server with multiprocessing support and efficient scaling, forming the backbone of websites. The data-serving applications are all tested with 500 MB datasets.

Compute: Compute engines are used in microservices to process data in memory and provide analytics or results to end-users. To evaluate compute workloads, Graphchi-PageRank [38] and FIO [39] are used. Graphchi is a graph-processing framework and PageRank represents example compute applications constructed with it. FIO is a flexible memory-mapped IO benchmark. Both compute applications are tested with 500 MB datasets. Graphchi's dataset was obtained from SNAP [40].

Client: The Yahoo Cloud Benchmark Suite [41] is used to drive requests in the evaluation section. The MongoDB and ArangoDB3 clients are used as drivers to the evaluated databases, while the REST web client is used to drive the evaluated web server, httpd. The workload configuration for YCSB is shown in Table 4.3.

4.2.2 Functions-as-a-Service

Functions-as-a-service enable a new software paradigm where execution is instantiated and completed at function-granularity. It replaces the persistent

Table 4.3: YCSB Configuration

YCSB Workload Parameters	
Hot Spot Data Fraction	0.2
Hot Spot Operation Fraction	0.8
Field Count Per Entry	1
Field Length	100 Bytes
Access Distribution	Uniform

and long running servers with instances of functions that scale with demand. This enables a pay-as-you-go model in data centers and software services. To represent FaaS, this thesis utilizes two C/C++ functions. These are named *URIParser*, which parses a provided input into URI tokens and *Hashing*, which generates indices given a set of keys based on the sdbm algorithm [42]. Both of these FaaS applications are evaluated in two forms: dense and sparse. The dense versions of the applications touch all of the input data given to the function. The sparse versions touch 20% of each 4 kB page before moving onto the next one. The two different forms ensure different memory access patterns are evaluated. The functions are tested with 500 MB datasets.

4.2.3 GPU-Accelerated Services

Accelerated computing provides highly efficient performance in data centers for emerging application classes such as those rooted in machine-learning or high-performance and scientific computing. Domain-specific processors such as the Google TPU [43] and NVIDIA Volta [44]. To represent such workloads, this thesis evaluates four GPU applications. Three of these applications represent the high-performance scientific computing domain with lower-upper decomposition (LUD), Particle Filter from the Rodinia benchmark suite [45, 46] and Stencil from the Parboil benchmark suite [47]. To represent machine learning workloads, this thesis utilizes a convolutional neural network (CNN), LeNet.

CHAPTER 5

ANALYSIS

This chapter discusses the results obtained from the experimentation described in Chapter 4. More specifically, it details how well the Docker Engine can scale and the bring-up overheads of different applications. Then, a characterization of the page tables and active pages of each application is provided. The page fault characteristics are identified for microservices and functions-as-a-service. Finally, GPU accelerated containers are discussed and their performance is compared to native execution.

5.1 Docker Engine and Container Bring-up

5.1.1 Docker Engine Performance

Figure 5.1 shows the mean and tail (95th percentile) container bring-up times on average where a client sends *start* requests to a set of pre-created containers. Start brings a container online and begins executing the application encapsulated in the container. The starting time is critical for applications to be brought up and torn down quickly for on-demand scaling in data centers. The client tests the Docker Engine performance by varying the request rate from 1 to 8 containers per second (CPS). It can be inferred that the Docker Engine's parallelism does not strongly scale. At a one container per second rate, container start requests are completed in 0.68 seconds on average, with a tail response latency of 0.78 seconds. The mean and tail response latencies for four containers per second are 1.55 seconds and 2.24 seconds, respectively. At eight containers per second, the average container start time is 2.65 seconds, with a tail response latency of 4.59 seconds. While the latency increase is sub-linear, container bring-up time does increase significantly due to the serialization of requests at the Docker Engine side.

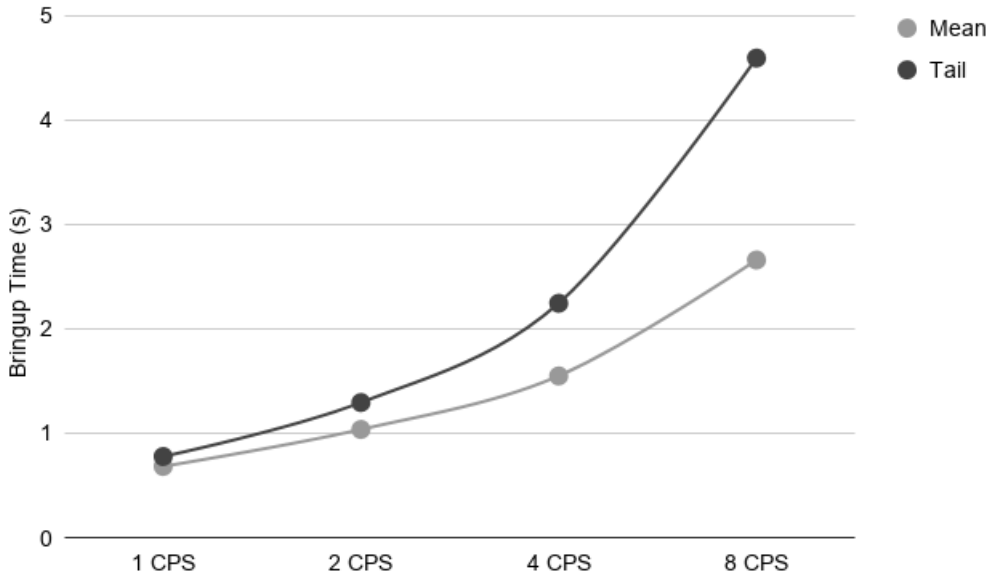

Figure 5.1: Docker Engine Scaling Performance

At 8 CPS, it is observed that applications generally have similar bring-up times with the exception of MongoDB as seen in Figure 5.2. The bring-up time is affected by the container size, application start-up time and process launching in the kernel as discussed later in this section.

5.1.2 Functional Breakdown of Container Bring-up

Containers are brought up with the fork() system call which is invoked by the Docker Engine. The Docker Engine first intercepts a container operation and decodes what the operation type is. For a container start operation, the Docker Engine uses the Containerd back-end [48]. The Containerd back-end jumps to the Go-Runc library [49]. The Go-Runc library executes a Start() call that goes to the Go os/exec package [50]. The executed Start() call is a fork() system call wrapper that launches the container process. As a result, container bring-up is lengthy and it involves kernel-level tasks alongside the application start-up.

Figure 5.3 shows the bring-up overhead split between kernel space tasks and user space tasks (in application space, libraries and runtimes). On average, 56.94% of the time bringing up a container is spent in the kernel space, and 43.06% is spent in the user space, libraries and runtimes.

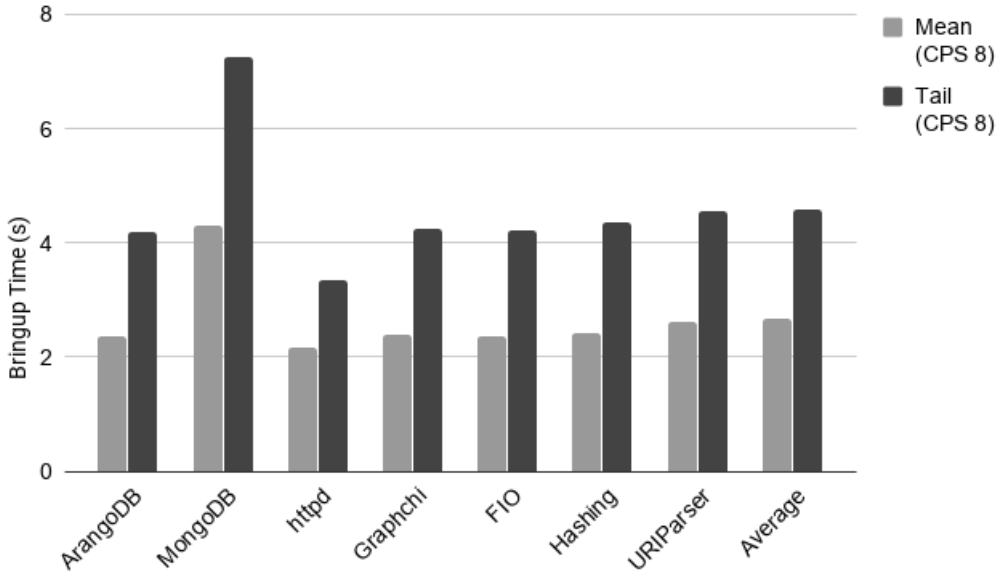

Figure 5.2: Container Bring-up Time per Application

Of the container bring-up overhead, 41.65% can be attributed to two collectives as seen in Figure 5.4. These are the *memory management* and *garbage collection* tasks done by the kernel and Go runtime. Of the bring-up time, 21.8% is spent in the Go garbage collector, and 19.85% is spent in memory management and process start-up tasks in the kernel.

Memory Management: The major function signatures identified under this collective are *copy_page_range()*, which copies a set of page table entries to a new set of page tables; *handle_mm_fault()*, which is the page fault handler; and *unmap_page_range*, which removes a set of page table entries from a given page table.

Garbage Collection: Two major signatures are visible in the Go garbage-collection and memory allocation tasks. Namely, *mallocgc()* and *gcDrain()*. The former is the memory allocation handler for the Go runtime. The latter traverses a list of objects tracked by the garbage collection scheme used in Go [51], and colors objects to determine if an object is reachable or unreachable by the application.

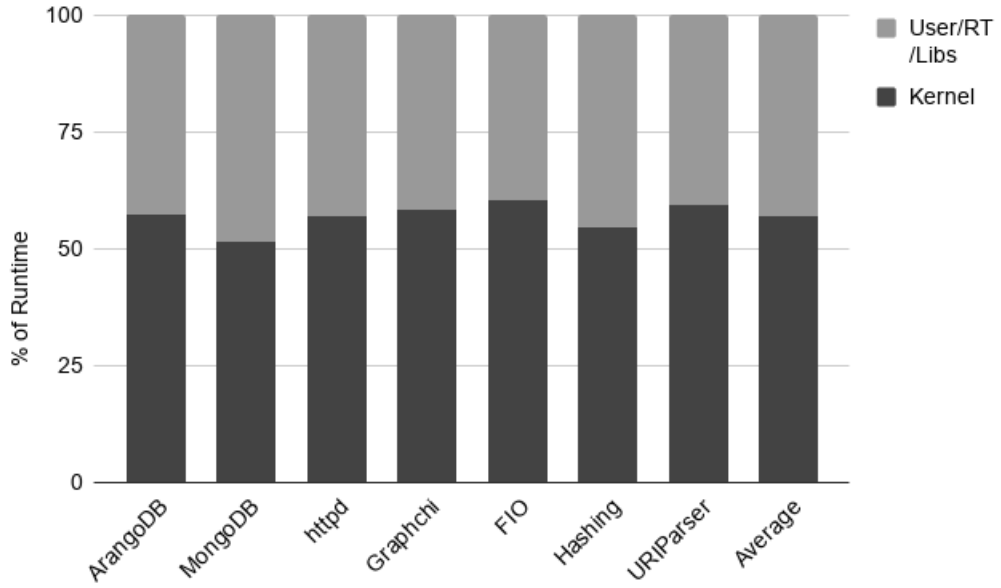

Figure 5.3: Container Bring-up Kernel vs User Space

5.2 Virtual Memory Characterization

5.2.1 Page Table Entry Duplication

Containers of the same image type end up having highly similar page table mappings as a result of the fork() based creation and the shared container infrastructure caused by the image-based creation. In a data center environment where different tenants would be provided with different ASLR seeds, container processes owned by one tenant could end up with the same virtual memory layout. When multiple processes are servicing the same data in a containerized environment in the previously described scenario, each container of the same type ends up having identical virtual-to-physical page mappings for shared data. This phenomenon is studied by placing four containers of the same type on a core.

At the application initial state, which is right after the bring-up, applications have varying levels of *groupable* and *exclusive* page mappings. A page mapping is defined to be groupable if the virtual page number and the physical frame number are the same across multiple processes. If the mapping is unique, then it is deemed to be an exclusive page mapping. Figure 5.5 shows that on average, 53% of the total page table mappings are groupable and 47%

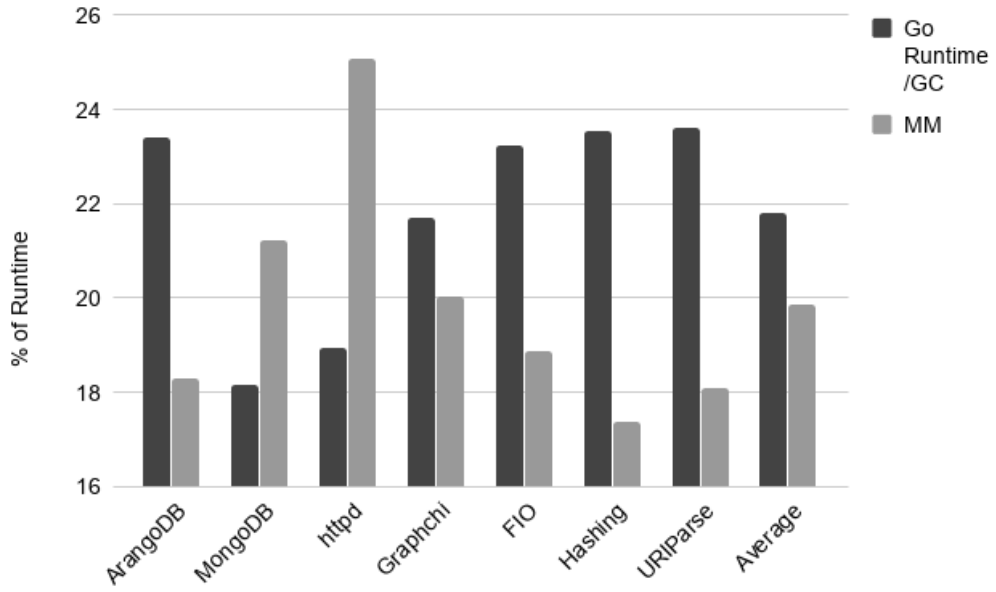

Figure 5.4: Container Bring-up Overheads Functional Breakdown

are exclusive. The highest groupability observed is 76.3% with httpd and the lowest is 2.63% with FIO. The more internal memory a process allocates, the more exclusive its page mappings are. Application code, libraries, shared memory regions and memory mapped files contribute to the groupable page mappings.

FIO and Graphchi have the largest exclusive page mapping ratios at the application initial states. This is because neither of them have populated their mapped input datasets at that time. For ArangoDB and MongoDB, the initial set of groupability comes from partially loaded datasets and the shared application code and container infrastructure. Httpd launches a large number of worker and manager processes. The large number of processes contributes to a high level of shared application code and libraries in addition to the shared infrastructure across the containers. This ultimately leads to majority groupable page table entries at the initial state for httpd. For the Hashing and URIParser functions, the groupability comes from the container infrastructure, application code and memory-mapped, partially populated input datasets.

The steady states of applications and page tables are measured after two minutes of execution time. Hashing and URIParser applications do not have a steady state as they are short-lived functions. As a result, they are omitted

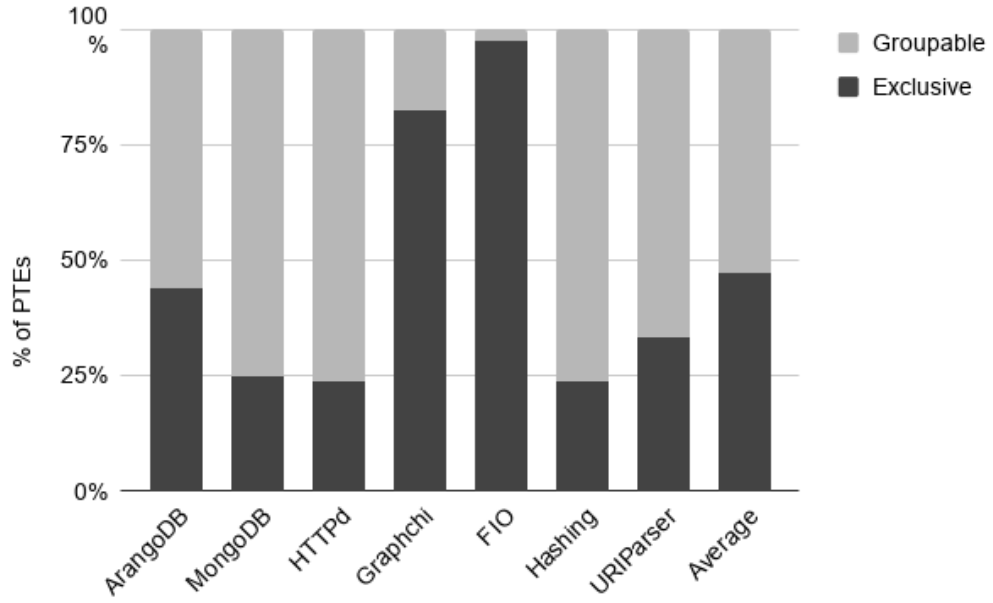

Figure 5.5: Page Table Entries at Application Initial State

from the steady state analysis. Figure 5.6 shows the groupability of page table entries at the application steady state. On average, 63.7% groupability is observed across microservices. The lowest share of exclusive mappings is in MongoDB with 2%. MongoDB and ArangoDB randomly service data with minimal internal buffering and processing. That causes their groupability to increase as applications run for longer periods of time and more of their memory mapped datasets populate page table entries. Httpd serially services the files it hosts, also with minimal processing and internal buffering. FIO does not initially populate the page table entries for its mapped dataset and instead does so over time with randomized accesses. That is the reason for the increase in groupability to 55.3% in FIO. Graphchi has the least groupable set of mappings, with 74.7% of its mappings marked as exclusive. Graphchi's behavior is attributable to the fact that it only memory maps the vertices of the input graphs while internally buffering the edges. That leads to large amounts of differing mappings across Graphchi processes.

The total set of page table entries for an application does not provide an accurate picture of an application's behavior during runtime. Figure 5.7 shows the active set of pages at the application initial state. An active page is a page that has been accessed at least multiple times. On average, 52.4% of page mappings are active at the application initial state. It is observed

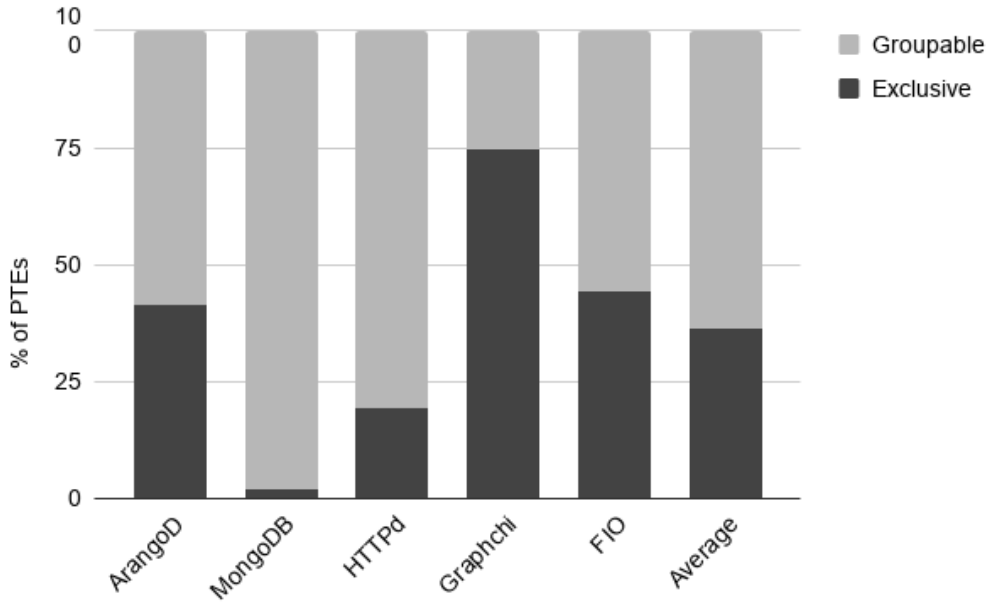

Figure 5.6: Page Table Entries at Application Steady State

that most of ArangoDB, MongoDB and FIO's pages are inactive at the initial state whereas httpd, Graphchi, Hashing and URIParser are mostly active. FIO's active pages are mostly comprised of application code and container infrastructure that was utilized during start-up. ArangoDB and MongoDB's active pages are mostly comprised of the partially populated pages from the memory-mapped datasets in addition to the code and infrastructure utilized during start-up. The large number of worker processes spawned by httpd indicates that the code and infrastructure utilized upon start-up is replicated across multiple processes, raising the active groupable page count. Graphchi's start-up includes pre-allocated buffers to read in graph edges. Hashing and URIParser are short lived and therefore a majority of their pages have been accessed and operated on at the initial state.

After two minutes of execution, active pages make up 71.9% of all pages on average as shown in Figure 5.8. A key observation here is that the groupable page mappings also increase with the number of active pages as expected. At the steady state, 59.86% of page table entries are deemed active and groupable. As the selected applications run, they operate on their memory mapped datasets and allocate internal buffers as needed. ArangoDB, MongoDB, httpd, Graphchi and FIO all display this behavior at different levels.

An important takeaway is the lack of transparent huge page support across

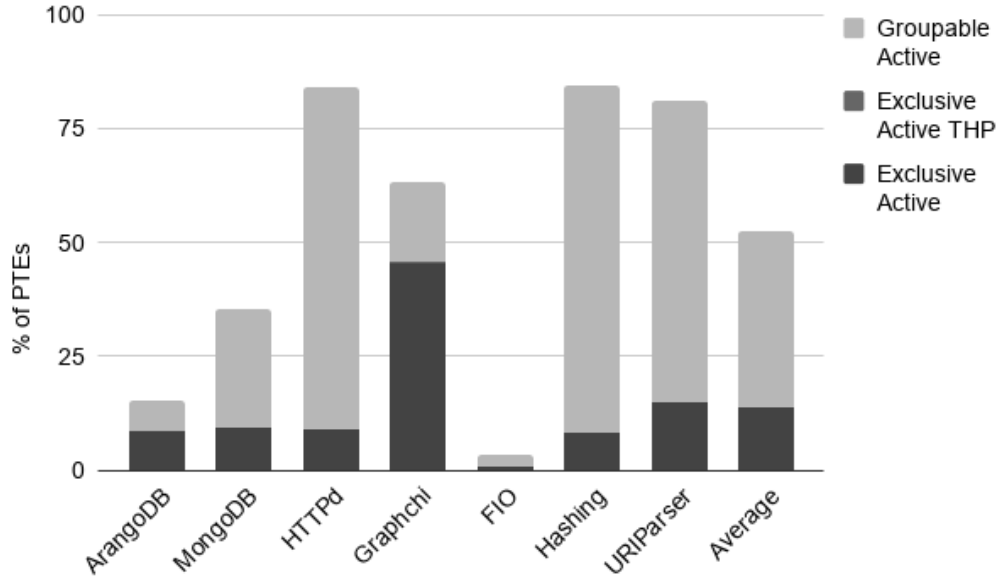

Figure 5.7: Active PTEs at Application Initial State

the board. Out of the active pages, the largest transparent huge page coverage is seen in Graphchi, where 0.07% of the page mappings are part of a huge page. Transparent huge pages only support anonymous page mappings, such as pages used for internal buffers. As a result, all of the huge page mappings cover exclusive pages, and none of the groupable pages are part of huge pages. The majority of page mappings analyzed are groupable. The incurred page fault and page walking overheads are not alleviated by huge pages, and need to be supported.

5.2.2 Page Fault Analysis

The shared structure of containers of the same application type paired with the page table similarities also leads to potentially increased TLB pressure and duplicated page faults. To characterize page faults, microservices are profiled for 10 seconds of runtime, whereas the FaaS applications are measured from bring-up to finish.

Figure 5.9 captures the percentage of total page faults that occurred on identical virtual-to-physical page mappings. On average, 35.74% of total page faults are duplicated. Page fault duplication is expected to increase with more similar memory access patterns across applications. ArangoDB

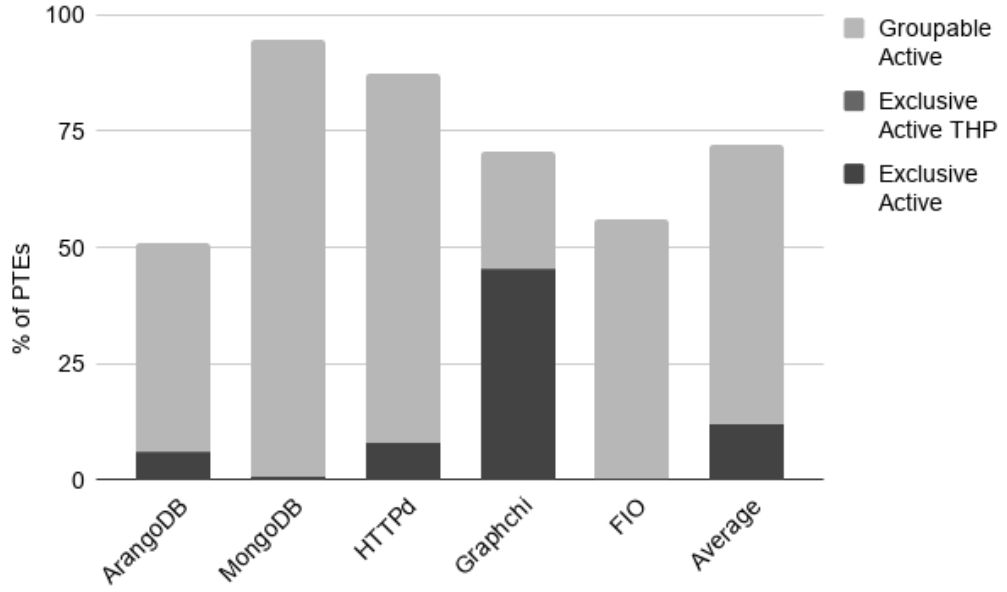

Figure 5.8: Active PTEs at Application Steady State

and MongoDB service their data as configured in Table 4.3. FIO does fully random reads across the whole memory mapped dataset. Graphchi's PageRank is probabilistic and separate Graphchi processes may display differing memory accesses. Under those conditions, the aforementioned four applications have page fault duplication ranging from 10.87% to 21.42%.

Httpd services files serially in a sequential memory access pattern. Hashing and URIParser applications also serially read and process their input datasets, sequentially in the dense versions and with constant strides in the sparse versions. The resulting similar memory access patterns also contribute to page fault duplication. These three applications in five forms display 51.38% to 55.68% page fault duplication.

Figure 5.10 shows the page fault rates across microservices and functions-as-a-service. On average, 15.07 page faults per one million instructions were observed.

ArangoDB, MongoDB, and Graphchi applications have lower page faults per million instructions. The database applications service 20% of their dataset 80% of the time. The reuse will prevent further page faults. Graphchi's traversal will mostly touch the in-memory buffered edges, leaving a lower page fault per million instruction rate with respect to its overall operation as a graph analytics benchmark. The page fault rate goes up to 47.32 page faults

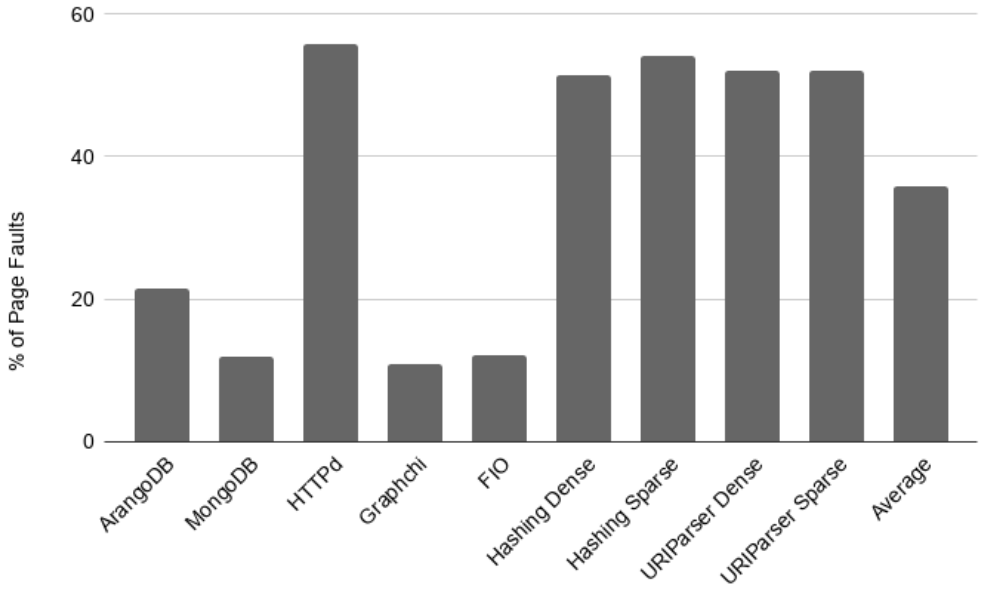

Figure 5.9: Page Fault Duplication

per one million instructions in httpd. This can be attributed to the minimal processing httpd does, leaving few non-memory instructions and servicing data sequentially. Most of the execution time is spent servicing new data, which leads to high page fault rates. A similar pattern is observed in FIO, albeit at a smaller magnitude.

Table 5.1: Full Page Fault Breakdown

Page Fault Breakdown (%)						
Application	Read	CoW	Shared	Anon	Swap	WP
ArangoDB	48.164	0.668	0.026	47.973	0.000	3.168
MongoDB	52.420	0.730	0.029	43.477	0.000	3.345
HTTPd	61.598	0.857	0.034	33.484	0.000	4.028
Graphchi	3.127	0.365	7.401	85.166	0.000	3.940
FIO	14.652	0.223	81.925	1.722	0.000	1.478
Hashing Dense	23.399	2.265	0.034	38.473	0.003	35.826
Hashing Sparse	29.976	2.081	0.030	34.312	0.002	33.599
URIParser Dense	21.104	2.341	0.037	39.771	0.000	36.747
URIParser Sparse	23.930	2.228	0.030	36.916	0.000	36.896
Average	30.930	1.306	9.950	40.144	0.001	17.670

Table 5.1 shows the full page fault breakdown for each application. This thesis analyzes six types of page faults. The page fault handler will first

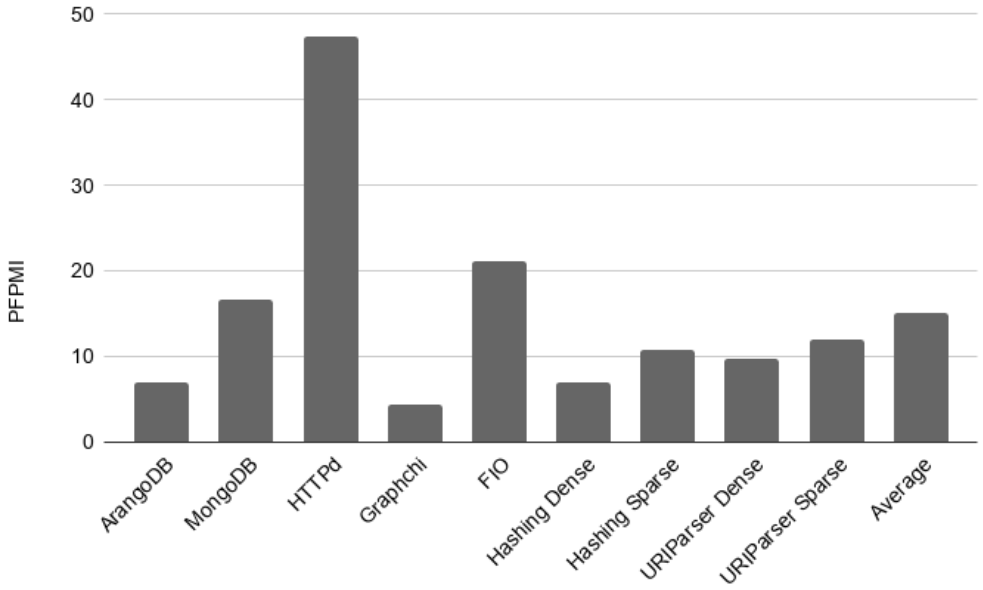

Figure 5.10: Page Faults Per Million Instructions

check to see if the searched PTE is present in the page tables. If the PTE is not present in the page table, the handler will check whether the page is mapped anonymously. If it is, that is an *anonymous fault*. If the page is not anonymously mapped and the access is a read, the page fault is a *read fault*. If the access is a write on a page that is not shared, it is a *copy-on-write fault*. Otherwise it is a *shared fault*. In the case where the PTE exists but is not present, then that is a *swap fault*. Finally, if the PTE is present on a write access fault and the PTE is not marked for writes, that is a *write protect fault*. The majority of the page faults observed are read and shared faults, at 30.93% and 40.14%.

The persistence of groupability across page table entries and continued duplicated page faults depends on the page mappings staying the same as the parent process. Figure 5.11 shows the percentage of copy-on-write faults across the total number of page faults in each application. A copy-on-write fault causes a process to break off from the parent mappings, and by extension the mappings of other containers of the same type. Upon a copy-on-write, a new private copy of the page is created for the faulting process, and thus the mapping becomes exclusive. On average, 1.30% of the overall page faults are copy-on-write faults. The low rate of copy-on-write faults indicates that the analyzed page table mappings stay intact throughout execution.

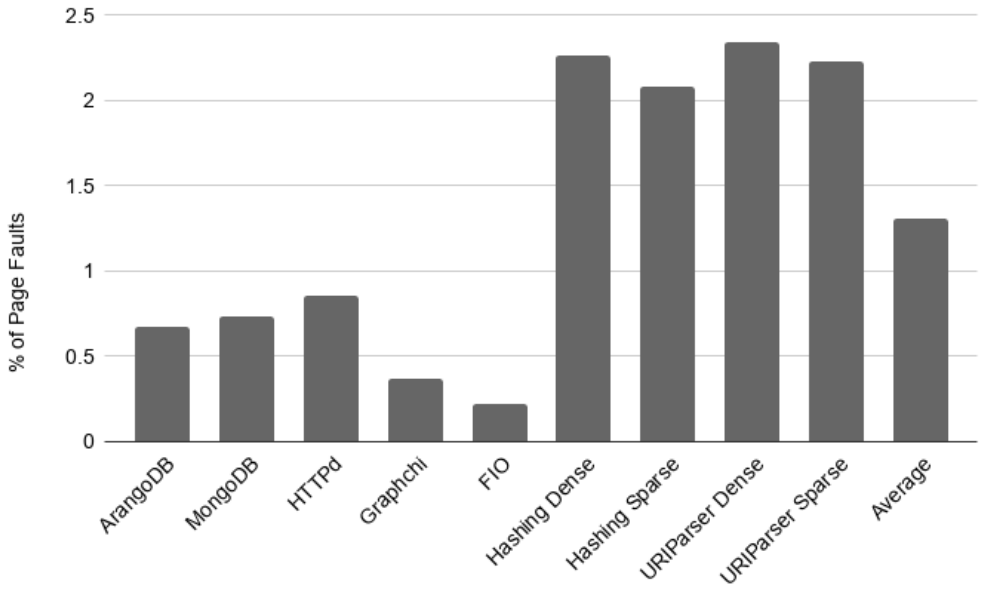

Figure 5.11: CoW Page Fault Percentage

5.3 Containerized GPU Application Performance

Figure 5.12 displays the CUDA kernel and API call split across the set of containerized GPU applications. On average, native execution spends 25.854% of the application runtime in GPU kernels and 74.146% of application runtime in CUDA API calls. Containerized execution spends 25.32% of its total runtime in GPU kernels, and 74.68% in CUDA API calls. The overall split is similar, with a 0.5% degradation in GPU utilization from a shift in kernel execution to API time. The difference is caused by comparable kernel execution time and an increase in the CUDA API time when the containerized environment is used.

The containerized application experiences an average 7% slowdown when application time is measured end-to-end. Figure 5.13 displays the containerized performance for each application normalized to native execution. Containerization overhead ranges from 3.4% in Stencil to 10.8% in Particle Filter. The source of the slowdown stems from the CUDA just-in-time (JIT) compilation, as shown in Figure 5.14. The just-in-time compiler in CUDA ensures that older device architectures are supported by ensuring the correctly generated PTX is executed during runtime [52]. NVIDIA-Docker incurs an 11.9% JIT compilation overhead on average, ranging from 9.4% in LeNet to 13.1%

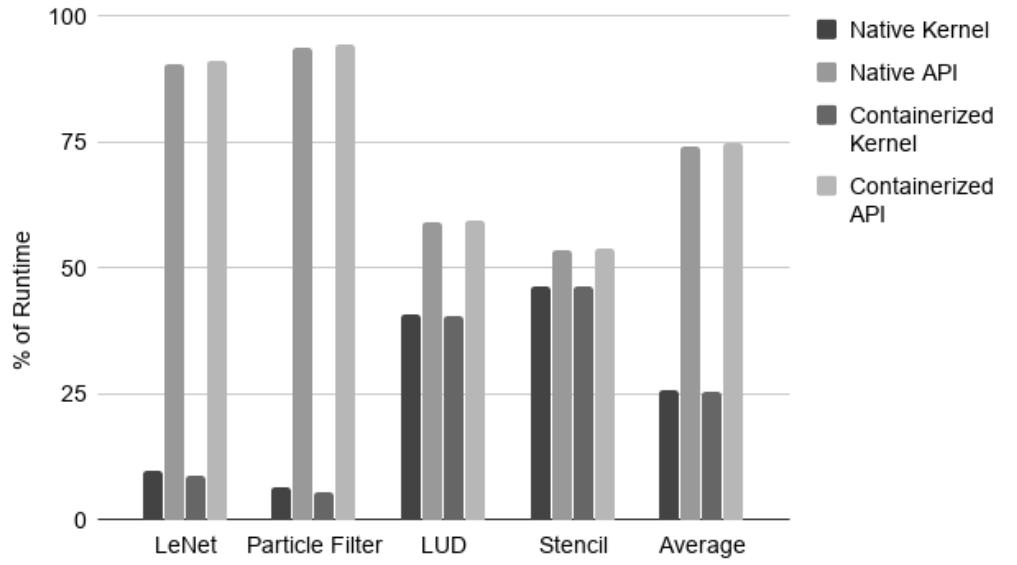

Figure 5.12: NVIDIA-Docker Kernel and API Distribution

in Stencil.

Two other critical API calls are compared. These are cudaMalloc and cudaMemcpy. GPU memory is allocated in a CUDA application by using cudaMalloc. Memory transfer between the GPU and the CPU is done using cudaMemcpy. Containerized execution incurs a 12.2% overhead on cudaMalloc, and 1.4% overhead in cudaMemcpy on average. The difference in cudaMemcpy ranges from 3.3% faster than native execution to 7.1% slower than native execution. This can be attributed to noise, and cudaMemcpy is concluded to run at native-level. The total time spent in cudaMalloc does not exceed 1.2% of total end-to-end runtime. As a result, the cudaMalloc slowdown is not a major overhead to the application.

The evaluated applications are compiled to generate appropriate parallel thread execution assembly (PTX) for the evaluated system described in Table 4.2 in order to minimize JIT overhead. NVIDIA-Docker containers are meant to be device and driver agnostic. In a use case where a particular compute capability is not specified during compilation to support a wider range of devices, the observed JIT overhead will greatly increase. As a result, it can be concluded that NVIDIA-Docker performs at near-native level where the container supports specific devices, but performance losses are observed where the supported architecture range increases.

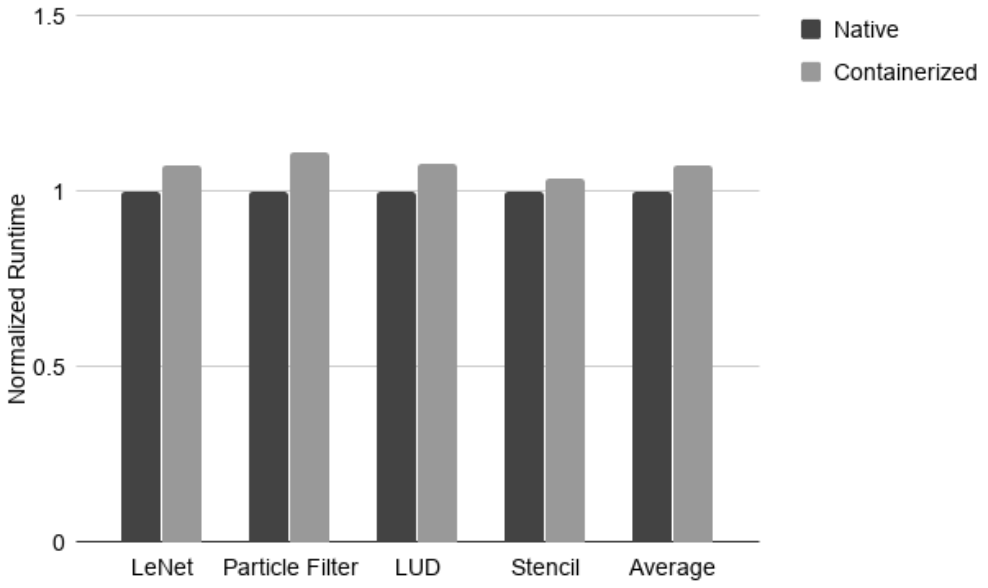

Figure 5.13: NVIDIA-Docker Application End-to-End Runtime

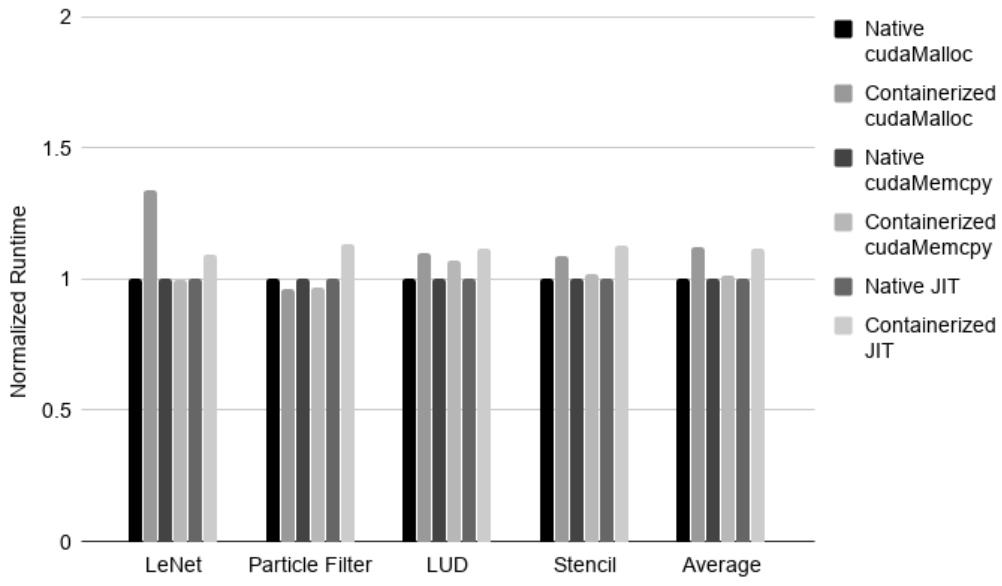

Figure 5.14: NVIDIA-Docker API Overheads

CHAPTER 6

RELATED WORK

This chapter provides a brief summary of the previous research relevant to this thesis. Past container characterization studies, enhanced TLB designs, new address translation flows and page translation sharing schemes are highlighted.

6.1 Container Characterization

Researchers investigate Docker storage system performance using NVMe SSDs, varying file systems, storage drivers and block sizes in [53]. Their observation shows that using Docker volumes brings container I/O performance close to raw block device performance consistently. This work focuses on storage operations in Docker.

The authors of [54] evaluate the bring-up time, memory use and CPU utilization of containers and virtual machines with big data applications at large cluster sizes. They use k-means clustering, logistic regression, Pagerank and SQL join to study container and virtual machine performance. Their main observation is that containerized environments have stronger scaling than virtual machines, attributed to more effective CPU utilization at large cluster sizes and lower memory usage in containers. This study further establishes the lightweight nature of containers.

Finally, another study tests container performance using MapReduce [55] performance [56]. The authors find that container performance is comparable to native execution in sorting applications with large datasets. They also note that no significant performance impact is observed by running two containers side by side, one with their evaluation applications and one that stresses CPU, memory or I/O. Their findings are obtained with LXC.

Existing container characterization studies all focus on different aspects of

container performance and isolation. All of these works are orthogonal to the analysis done in this thesis.

6.2 Enhanced TLBs and Address Translation

Traditional processor hardware has separate structures for different page sizes in their TLBs to avoid energy efficiency along with design and verification complexity issues. The drawback of this approach is underutilized TLB hardware in cases where the kernel extensively uses one type of page size. MIXTLB [57] proposes a new TLB design that unifies all page sizes under one hardware structure to increase TLB utilization while preserving an energy-efficient and simple design.

Guest operating systems in virtual machines keep their own set of page tables. Memory accesses in virtual machines need to be translated to an address in the host machine. This causes page walks to be two-dimensional. POMTLB [58] implements an L3 TLB that is kept in a portion of main memory. A miss in the L2 TLB for their scheme checks the in-memory TLB for the desired entry instead of initiating a page walk immediately. The authors convert two-dimensional page walks to single accesses and also utilize the cache hierarchy to keep more TLB entries in the processor.

Previous research has also evaluated the performance benefits of using flat nested page tables and an inverted shadow page table [23]. Their work focuses on eliminating the page walking overhead of virtual machines, increasing TLB hit rates and reducing the number of memory accesses in a page walk.

Enhancing TLB structures and accelerating the address translation flow are extensively studied in computer architecture. Containers do not suffer from two-dimensional page walks. However, this study observed the duplication in page table entries and the duplicated page fault rates. As a result, enhancing address translation and TLB organization can benefit containers.

6.3 Page Table Sharing

In the Android kernel, every new process forks from a wrapper process called the zygote. This, coupled with the fact that mobile applications rely on high

levels of shared libraries, presents an opportunity to speed up runtime by sharing translations, as shown in [59]. The authors identify code and library pages that are shared across multiple applications and mark them as global. Global page translations are shared in ARM TLBs, and their scheme results in reduced PTE copying overhead and fewer page faults. They also observe increased TLB performance as a result.

VMWare has implemented a page and translation sharing mechanism in the ESX server [60]. The author identifies identical pages mapped to multiple virtual machines and maps each guest operating system page to the same guest physical page. This allows greater memory commitment to virtual machines and reduces the memory footprint of each virtual machine.

Page and translation sharing mechanisms directly relate to the memory characteristics of Docker containers as identified by this thesis.

CHAPTER 7

CONCLUSION

7.1 Conclusion and Takeaways

Docker is the prominent container framework today in an environment dominated by cloud computing. Containers on the same physical host share the same kernel, rendering them inherently more lightweight. While the performance of containers is much closer to native execution than virtual machines, they have several characteristics that should be supported by future architectures.

First, the Docker Engine is reliant on the Go runtime and kernel tasks to bring up a container. Go memory allocation and garbage collection overheads coupled with process creation in the kernel constitute 40% of the container bring-up time.

Second, up to 97% groupable address translations and up to 54% duplicated page faults are observed. Microservices operate on very large datasets and it is critical to reduce the unnecessary translation overhead. The shared infrastructure of containers and process-level scaling of containerized applications present an opportunity to share translations to drastically reduce the page faults observed by containerized applications and also de-duplicate redundant page table entries.

Functions-as-a-service scale at function granularity. Functions can be brought up, executed quickly and torn down on-demand. As a result, optimizing the container bring-up process is key to having resilient and high-performing architectures for containers.

GPU-accelerated service performance is reliant on the compilation process. Specifically compiling a CUDA application for a small set of compute capabilities and GPU architectures or a particular device reduces the overhead of just-in-time compilation. Deploying an NVIDIA-Container image for a

wide set of architectures or without specifying any compute capabilities to the NVIDIA CUDA Compiler yields greater JIT overhead, further degrading application end-to-end performance. In an optimal scenario where device architecture is specified, this thesis observes an 11.9% end-to-end performance degradation imposed by NVIDIA-Docker. Kernel performance, however, is not affected by containerization.

7.2 Future Work

Studying applications thoroughly is key to developing domain-specific processors and accelerators today. Hardware/software co-design allows one to optimize across layers and enable high-performance computing with more optimal solutions than hardware-only or software-only approaches. With the advent of heterogeneous computing in data centers, NVIDIA-Docker and GPU computing gain prominence. A case-study based investigation of NVIDIA-Docker overheads should be investigated by stressing shared GPU resources by instantiating many processes using modern frameworks such as Caffe [61] and Pytorch [62] to effectively recreate a data center environment. These frameworks are deployed through NVIDIA NGC [63], as they power many high-performance applications today.

The transformation in application classes today must be followed by researchers in heterogeneous computing. The effects of containerization on high-performance, scientific and parallel computing should be instrumented intricately to enable the next generation of processor architectures.

REFERENCES

[1] S. Seelam, "Docker at Insane Scale on IBM Power Systems," 2015. [Online]. Available: https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems/

[2] Amazon Web Services, "Amazon Container Services," 2018. [Online]. Available: https://aws.amazon.com/containers/

[3] IBM, "IBM Cloud Kubernetes Service," 2018. [Online]. Available: https://www.ibm.com/cloud/container-service

[4] Google, "Containers at Google," 2018. [Online]. Available: https://cloud.google.com/containers/

[5] Microsoft, "Azure Kubernetes Service," 2018. [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/

[6] Google, "Kubernetes Engine," 2018. [Online]. Available: https://cloud.google.com/kubernetes-engine/

[7] Amazon Web Services, "AWS Lambda," 2018. [Online]. Available: https://aws.amazon.com/lambda/

[8] IBM, "IBM Cloud Functions," 2018. [Online]. Available: https://www.ibm.com/cloud/functions

[9] Google, "Google Cloud Functions," 2018. [Online]. Available: https://cloud.google.com/functions/

[10] NVIDIA, "GPU Enabled Docker Container," 2018. [Online]. Available: https://www.nvidia.com/object/docker-container.html

[11] LXC Contributors, "What's LXC?" 2018. [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[12] Docker Inc., "What Is a Container," 2018. [Online]. Available: https://www.docker.com/resources/what-container

[13] Docker Inc., "Docker Engine," 2018. [Online]. Available: https://docs.docker.com/engine/#why-docker

[14] R. Olson, J. Calmels, F. Abecassis, and P. Rogers, "NVIDIA Docker: GPU Server Application Deployment Made Easy," 2016. [Online]. Available: https://devblogs.nvidia.com/nvidia-docker-gpu-server-application-deployment-made-easy/

[15] R. Rahman, "Intel Xeon Phi Microarchitecture," 2013. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture

[16] "5-Level Paging and 5-Level EPT," White Paper, Intel, May 2017.

[17] Kernel.org Contributors, "Huge TLB Page," 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[18] Red Hat, "Huge Pages and Transparent Huge Pages," 2018. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge

[19] M. Gorman, "An investigation into the theoretical foundations and implementation of the linux virtual memory manager," M.S. thesis, University of Limerick, Ireland, 2003.

[20] VMWare, "VMware ESXi: The Purpose-Built Bare Metal Hypervisor," 2018. [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html

[21] KVM Contributors, "Kernel Virtual Machine," 2018. [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[22] Xen Project, "The Hypervisor (x86 ARM)," 2013. [Online]. Available: https://www.xenproject.org/developers/teams/hypervisor.html

[23] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337214 pp. 476–487.

[24] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988337 pp. 1:1–1:13.

[25] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "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*, ser. EuroSys '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273025 pp. 275–287.

[26] Linux-VServer Developers, "Linux-VServer Overview," 2013. [Online]. Available: http://www.linux-vserver.org/Overview

[27] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *IBM Research Report*, Jul 2014.

[28] "NVIDIA Tesla V100 GPU Architecture," White Paper, NVIDIA, August 2017.

[29] N. Sakharnykh, "Maximizing Unified Memory Performance in CUDA," 2017. [Online]. Available: https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

[30] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," Citadel, High Performance Computing RD Team, Tech. Rep., 2018.

[31] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942 pp. 743–758.

[32] Kernel.org Contributors, "perf: Linux Profiling with Performance Counters," 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[33] MongoDB, Inc., "What are Microservices?" 2018. [Online]. Available: https://www.mongodb.com/what-are-microservices

[34] Docker Inc., "Docker Hub," 2018. [Online]. Available: https://hub.docker.com/

[35] MongoDB, Inc., "What is MongoDB?" 2018. [Online]. Available: https://www.mongodb.com/what-is-mongodb

[36] ArangoDB, Inc., "ArangoDB - a Native Multi-Model Database," 2018. [Online]. Available: https://www.arangodb.com/why-arangodb/multi-model/

[37] Apache Software Foundation, "httpd - Apache Hypertext Transfer Protocol Server," 2018. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/httpd.html

[38] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884 pp. 31–46.

[39] J. Axboe, "fio - Flexible I/O tester," 2018. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

[40] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[41] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152 pp. 143–154.

[42] M. I. Seltzer and O. Yigit, "A new hashing package for UNIX," in *Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991*, 1991, pp. 173–184.

[43] Google Inc., "Cloud TPU," 2018. [Online]. Available: https://cloud.google.com/tpu/

[44] NVIDIA, "NVIDIA Volta," 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

[45] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797 pp. 44–54.

[46] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274 pp. 1–11.

[47] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, Tech. Rep., 2012.

[48] Containerd Contributors, "An Open and Reliable Container Runtime," 2018. [Online]. Available: https://github.com/containerd/containerd

[49] The Open Container Initiative Contributors, "CLI Tool For Spawning and Running Containers According to the OCI Specification," 2018. [Online]. Available: https://github.com/opencontainers/runc

[50] The Go Authors, "Go OS/Exec Package," 2009. [Online]. Available: https://golang.org/pkg/os/exec/

[51] The Go Authors, "Go GC: Prioritizing Low Latency and Simplicity," 2015. [Online]. Available: https://blog.golang.org/go15gc

[52] NVIDIA, "Just-in-Time Compilation," 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#just-in-time-compilation

[53] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Docker characterization on high performance SSDs," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 133–134.

[54] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *IEEE International Conference On Cloud Computing*, vol. 12, 07 2018, pp. 178–185.

[55] IBM, "Apache MapReduce," 2018. [Online]. Available: https://www.ibm.com/analytics/hadoop/mapreduce

[56] M. G. Xavier, M. V. Neves, and C. A. F. D. Rose, "A performance comparison of container-based virtualization systems for MapReduce clusters," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 299–306.

[57] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *ACM SIGOPS Operating Systems Review*, vol. 51, 04 2017, pp. 435–448.

[58] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080210 pp. 469–480.

[59] X. Dong, S. Dwarkadas, and A. L. Cox, "Shared address translation revisited," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901327 pp. 18:1–18:15.

[60] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[61] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[62] Pytorch Contributors, "From Research to Production," 2018. [Online]. Available: https://pytorch.org/

[63] NVIDIA, "NVIDIA GPU Cloud," 2018. [Online]. Available: https://www.nvidia.com/en-us/gpu-cloud/