applied sciences

MDPI

Article

Performance Analysis of Container Effect in Deep Learning Workloads and Implications

Soyeon Park and Hyokyung Bahn *

Department of Computer Engineering, Ewha University, Seoul 03760, Republic of Korea; sypark9646@ewhain.net
* Correspondence: bahn@ewha.ac.kr; Tel.: +82-2-3277-4247

Abstract: Container-based deep learning has emerged as a cutting-edge trend in modern AI applications. Containers have several merits compared to traditional virtual machine platforms in terms of resource utilization and mobility. Nevertheless, containers still pose challenges in executing deep learning workloads efficiently with respect to resource usage and performance. In particular, multi-tenant environments are vulnerable to the performance of container-based deep learning due to conflicts of resource usage. To quantify the container effect in deep learning, this article captures various event traces related to deep learning performance using containers and compares them with those captured on a host machine without containers. By analyzing the system calls invoked and various performance metrics, we quantify the effect of containers in terms of resource consumption and interference. We also explore the effects of executing multiple containers to highlight the issues that arise in multi-tenant environments. Our observations show that containerization can be a viable solution for deep learning workloads, but it is important to manage resources carefully to avoid excessive contention and interference, especially for storage write-back operations. We also suggest a preliminary solution to avoid the performance bottlenecks of page-faults and storage write-backs by introducing an intermediate non-volatile flushing layer, which improves I/O latency by 82% on average.

Keywords: performance; deep learning; container; virtual machine; event trace; system resource

Citation: Park, S.; Bahn, H. Performance Analysis of Container Effect in Deep Learning Workloads and Implications. *Appl. Sci.* **2023**, *13*, 11654. https://doi.org/10.3390/app132111654

Academic Editor: Douglas O'Shaughnessy

Received: 31 August 2023
Revised: 23 September 2023
Accepted: 7 October 2023
Published: 25 October 2023

1. Introduction

As deep learning is widely used in various IoT (Internet-of-Things) services [1,2], the demand for AI (artificial intelligence) technologies in the mobile edge environment is continuously increasing [3,4]. Specifically, there is an increasing trend to run deep learning workloads using virtualized environments such as virtual machines and containers. This is because virtualization has the effect of improving computational elasticity and efficiency, which are important considerations in deep learning workloads [3]. The execution cycle of a deep learning workload can be divided into three phases: training, servicing, and monitoring, as shown in Figure 1. During the training phase, users need to install the configuration for the framework, provision the required computing resources, and then proceed with training. When training is complete, the model is exported and the API server is installed. The model is then deployed for service. In the service phase, users should monitor the model's performance to determine whether the model is overfitting or underfitting the training data. When the model's performance degrades, the user must train the model again. Therefore, configuring and managing deep learning infrastructure is a complicated and time-consuming task. All of these steps can be performed in the data center or cloud, but some tasks such as data collection and post-training monitoring are increasingly shifted to the edge side. Reducing energy consumption and privacy risks is important in mobile edge as continuous training is performed by using data collected from mobile devices and sensors [5]. Container-based deep learning has been introduced to provide such features in a managed service [6,7].

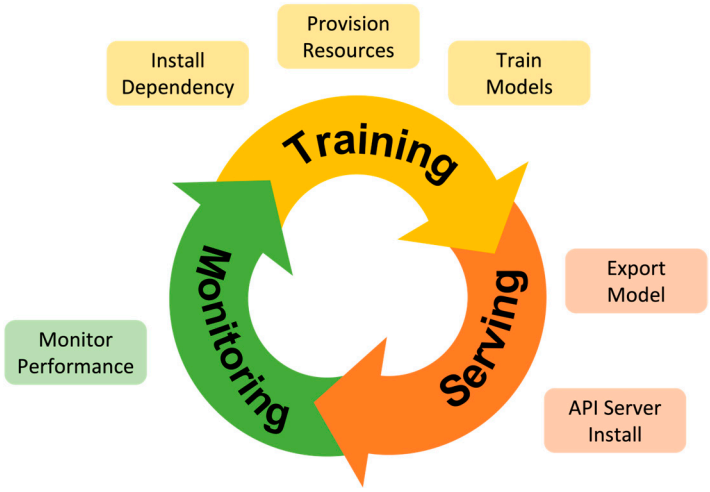


Figure 1. Execution cycle of a deep learning workload.

Containers provide several merits compared to traditional virtual machines for running deep learning workloads. That is, containers are lightweight and portable, so it is easy to deploy and manage containerized deep learning workloads. Containers also share the host OS, thereby minimizing the virtualization overhead. Thus, containers are efficient for compute-intensive deep learning workloads that require heavy resources. Another important merit of a container is that it provides easy management for configuration settings. In a virtual machine platform, each guest machine should have its own configuration files, thus keeping track of changes and ensuring the configurations are difficult. Moreover, in container environments, it is easy to recreate the same environments and dependencies used to train and deploy models. In contrast, in virtual machine environments, each guest machine has a set of dependencies, so recreating the same environment on another machine is not simple [8].

Despite these advantages, using containers to run deep learning workloads also presents some challenges. One of the biggest challenges is that containers are less isolated than virtual machines. This means that performance degradation is more likely to occur due to resource conflicts [9]. For example, if two containers try to access the same resource, a performance conflict may occur. Since containers share the host OS, they compete for system resources and need careful resource management techniques to prevent performance degradation. Fortunately, the overhead of containerization is not significant in traditional workloads. For example, running multiplayer gaming and video streaming workloads in a containerized edge environment incurs only a small amount of Docker overhead without affecting data processing in each container [10]. Also, as containers are frequently used for application deployment, network resources become a major bottleneck due to the large amount of data transfer between nodes [11]. However, this is not the case for deep learning workloads, which are resource-intensive and require more computation than communication [12]. Thus, containerization may introduce other resource bottlenecks such as CPU, memory, and storage. As deep learning is increasingly used in image processing in various service fields such as manufacturing and medicine, the importance of efficient resource management in containerized deep learning continues to grow [13,14].

Moreover, deep learning workloads have different resource usage patterns from traditional workloads, making efficient resource management more difficult [15,16]. Traditional workloads often have consistent resource usage patterns. For example, web servers or database servers repeatedly process the same tasks, such as http requests or query processing, so the number of requests varies, but the characteristics of the workload are uniform, making it relatively easy to cope with resource requirements [17]. In contrast, deep learning workloads are known to have very different resource usage patterns during the training and inference phases [18]. In the training phase, the model is learned on the data set, and

the parameters are updated. During this process, resource usage may show sudden spiky patterns. On the other hand, the inference phase uses models to predict new data and resource usage is not high and is relatively uniform.

It has been reported that deep learning workloads temporarily cause excessive memory usage during the training phase and that the bias in data accessed is weaker than traditional workloads because they rely less on hot data [19]. These characteristics make resource management more difficult compared to traditional workloads. Specifically, due to the spiky and bursty nature of memory usage in deep learning workloads, performance can rapidly deteriorate if insufficient memory space is allocated. However, equipping a large memory capacity to handle such situations will waste resources in the remaining time. So resource allocation that allows each container to run smoothly without over-allocation is a challenging issue. In addition, weak bias in data access makes it more difficult to design efficient caching mechanisms and limits performance gains through caching. Comprehensively monitoring data access patterns can be helpful in identifying and remediating resource usage issues. In particular, extracting and analyzing system traces from the host layer provides precise resource usage for each container, allowing for better resource allocation by accurately characterizing container workloads.

In this article, we investigate the resource requirements in the context of containerized deep learning workloads and aim to find performance implications for such environments. To do this, we extract and analyze the event traces of deep learning workloads in container environments and compare them to traces collected by running the same workloads on host systems. By doing so, we can see the impact of containerization in deep learning with respect to resource contention and performance overhead. We also investigate the impact of multi-tenancy by running deep learning workloads simultaneously in different containers to highlight challenges in resource management issues.

By analyzing system calls and event traces generated by container-based deep learning workloads, we provide implications for resource management in a containerized environment. Specifically, we observe that write-back operations to storage can be a major cause of performance bottlenecks. We also show that resource contention is significant in multi-tenant environments. To cope with this situation, we introduce a preliminary solution that adopts an intermediate non-volatile flushing layer to alleviate the performance bottleneck of storage write-backs. Instead of flushing to traditional storage, our solution effectively hides the write-back overhead by absorbing hot data in fast NVM media, improving I/O latency by 82% on average.

The remainder of this article is organized as follows. Section 2 briefly summarizes previous studies related to this article. In Section 3, we describe the experimental configurations of deep learning workloads to investigate the performance effect of container platforms. Section 4 analyzes the event traces captured and discusses their implications. In Section 5, we introduce our solution to cope with the performance bottleneck of containerized deep learning. Finally, Section 6 concludes this article.

2. Related Work

In this section, we briefly summarize previous research on deep learning workload characterization and container-based systems. We also review techniques proposed to improve the performance of such systems.

Recently, research has been conducted to characterize deep learning workloads in terms of reference patterns and resource usage. Park et al. comprehensively analyze memory reference patterns for neural network workloads and observe that they are significantly different from traditional workloads [19]. In particular, their analysis shows that the heap and data regions account for most memory references in deep learning workloads, but the memory reference bias is weaker than in traditional workloads, especially for write operations.

Berral et al. explored resource usage characteristics of containers used to train deep learning models [12]. They identify recurring patterns and similarities across containers

and suggest the potential to optimize resource allocation in a dedicated deep-learning cluster. Specifically, they utilize clustering techniques and conditional restricted Boltzmann machines to discover action steps during deep learning training. They also optimize resource allocation by dynamically adjusting container resources based on phase-specific statistical information. However, their approach only addresses resource allocation and overlooks communication considerations. Also, their methods are only effective for deep learning workloads with repetitive resource usage patterns and require sufficient historical data and clustering training.

Xu et al. investigated the effectiveness of Docker containers as a means to simplify the deployment and management of deep learning workloads [20]. In particular, they evaluate the impact of Docker containers on the performance of deep learning workloads. Their findings indicate that both CPU- and GPU-intensive tasks exhibit minimal overhead when running inside Docker containers. This means that Docker containers can be used for deep learning workloads without significant performance degradation. However, it is important to note that their study only evaluates the performance of Docker containers on specific hardware types, so different hardware configurations are likely to produce different performance results. As a result, further research is needed to evaluate Docker container performance across different hardware types and deep learning frameworks [20].

Bae et al. investigated the performance of Intel-Caffe, a distributed deep learning framework, on the Nurion supercomputers. Specifically, they focus on identifying the file I/O factors that affect the performance of Intel-Caffe in a container-based environment [21]. They observed that although the training phase of deep learning in a container-based environment has minimal overhead, page cache has a significant impact on the performance of deep learning frameworks.

Janecek et al. analyzed container workload characteristics by collecting system trace data from host systems [22]. Specifically, they classified containers based on their resource usage and behavior and identified idle containers in order to efficiently manage container clusters. However, their experiments use benchmarking tools to generate test data, which may not be representative of real AI workloads.

Zhang et al. explored resource consumption of containerized workloads on edge servers, with a particular focus on CPU resources, which are heavily consumed by container management and inter-container communication by daemon processes [23]. They present custom containers to improve CPU efficiency and evaluate the effectiveness of the algorithm based on specific metrics such as inter-container transfers, number of container starts, and application execution duration. Since their research focuses on CPU resources for network communication, additional considerations will be needed for deep learning workloads, where memory is another important bottleneck.

Avino et al. also performed a similar analysis to quantify the overhead of containerization in edge environments [10]. They showed that containerization incurs small CPU overhead without affecting the data processing of each container. However, their target workload is multimedia streaming and is not related to deep learning.

Recently, Rauschmayr et al. proposed a profiling tool that correlates system utilization metrics with framework operations in deep learning workloads [18]. Specifically, they deploy the profiling functionality as an add-on to Amazon SageMaker Debugger and identify resource usage patterns during the training and inference phases of deep learning.

Overall, previous research has focused on the potential to improve the performance of deep learning workloads using containers. However, further research is needed to evaluate the performance of these systems on different hardware configurations and AI workloads. Additionally, there are still challenges to be solved, such as resource allocation, communication, and memory usage optimization. Table 1 lists a brief summary of previous studies related to this article including their strengths and limitations.

Table 1. A brief summary of related work characterizing workloads with resource considerations.

	Resources Considered	Target Workloads	Strengths	Limitations
Park et al. [19]	Memory	AI workload	• Observe AI workload's memory usage patterns differentiated from traditional workloads • Determine the primary factor that leads to memory access in AI workloads	• More research is needed on different environments and models
Berral et al. [12]	CPU, memory	AI workload (containerized)	• Find resource usage patterns and similarities between containers • Improve resource usage in deep learning clusters	• Does not consider how tasks will communicate with each other
Xu et al. [20]	CPU, GPU	AI workload (containerized)	• Measure the effects of Docker containers in deep learning workloads	• Evaluation is limited to specific hardware types
Bae et al. [21]	I/O	AI workload (containerized)	• Observe the influence of page cache on the efficiency of deep learning frameworks	• Evaluation is limited to specific frameworks
Janecek et al. [22]	CPU, memory, I/O, network	Traditional workload (containerized)	• Categorize containers according to resource utilization and behavior • Identify and release idle resources	• Use benchmarking tools to analyze scenarios that may diverge from real-world conditions
Zhang et al. [23]	CPU, network	Traditional workload (containerized)	• Investigate resource utilization of containerized workloads running on edge servers • Enhance CPU efficiency by assessing the algorithm's efficacy	• The proposed algorithm does not consider the heterogeneity of edge nodes
Avino et al. [10]	CPU	Traditional workload (containerized)	• Present a quantitative assessment of the Docker container's CPU overhead	• Evaluation is limited to specific applications
Rauschmayr et al. [18]	CPU, I/O, GPU	AI workload (containerized)	• Propose a profiling tool that correlates system utilization metrics with framework operations	• Evaluation is limited to specific frameworks

3. Experimental Setup

This section describes an experimental configuration set up to quantify the performance of deep learning workloads running in Docker containers compared to running directly on the host system. The Docker framework we experiment with is composed of two images: a training image and a deployment image as shown in Figure 2. The training image does the work of loading the data, training the model, and storing the trained model outside the container. The deployment image runs the saved model and handles inference requests. To compare the performance of running deep learning workloads in Docker containers and directly on the host system, we construct two experimental sets. The first set runs training and deployment directly on the host machine, while the second set runs inside a Docker container. We run deep learning workloads on a single physical machine, and each workload is executed on a separate container. 4 containers are used in order to see the effect of multi-tenant environments. The hardware configuration of our experiment

consists of Intel i7-12700 CPU, Samsung DDR4 3200 16GB memory, Galax GeForce RTX 3060 GPU, Hynix Gold P31 SSD 2TB, and WD Blue 7200 HDD 2TB as listed in Table 2. We measure the performance of each set of experiments using a variety of metrics, including CPU usage, memory usage, and execution time.

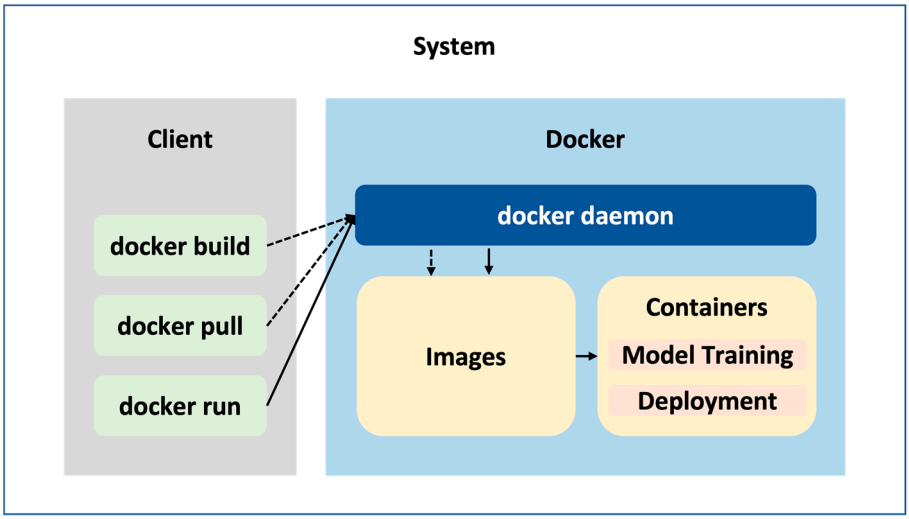


Figure 2. Interaction between system components.

Table 2. Hardware configuration of our experiments.

Item	Specification
Processor	Intel® Core™ i7-12700 Processor
Main Board	GIGABYTE Z690 GAMING X DDR4
Memory	Samsung DDR4 3200 (16 GB)
Graphics Card	GALAX GeForce RTX 3060
SSD	SK Hynix Gold p31 (2 TB)
HDD	WD BLUE 7200/256M (2 TB)

We leverage Ftrace, a kernel tracing framework, to evaluate the performance differences between the execution of the same task in a Docker container and the host system. Ftrace supports debugging and performance analysis by tracing a variety of events, including function calls, system calls, and interrupts. To ensure a fair and unbiased comparison, we control for various configurations that could potentially affect performance, such as the programming language and software library versions used. Details of software configurations we consider are listed in Table 3. All experiments are performed on the same system and the results reported are averages from five independent runs.

Table 3. Software configuration of our experiments.

Item	Version
Linux	Linux 5.19.0-42-generic
Docker	Docker 23.0.1
Python	Python 3.10
Pytorch	Pytorch 1.12.1
CUDA	CUDA 11.6

We collect event traces using the Ftrace utility while executing deep learning workloads consisting of two well-known datasets: the Wikipedia dataset [24] and the ImageNet dataset [25]. For the Wikipedia dataset, we perform preprocessing by using the Kakao morpheme analyzer (https://github.com/kakao/khaiii, accessed on 23 September 2023)

to separate samples into morphemes and generate word sets. The lengths of all samples are matched such that short samples are padded with trailing zeros. For the ImageNet dataset, the size of the training images was adjusted to 256 × 256 for image classification, and the shorter width and height were fixed to 256. As training models, we use PyTorch, the most widely used machine learning framework. Specifically, two large-scale models for text processing and two small-scale models for image analysis are used. Details of these models and their corresponding datasets are listed in Table 4. For model training, we first load the datasets, preprocess them, and repeat them for 10 epochs. The extracted event traces are categorized into four stages of deep learning: data load, model load, training, and inference.

Table 4. Model configuration of our experiments.

Dataset	Scale	Model
Text	Small	Mobile-BERT [26]
	Large	LSTM [27]
Image	Small	SqueezeNet [28]
	Large	AlexNet [29]

4. Analysis of Deep Learning Event Traces

In this section, we perform a comprehensive analysis of extracted event and system call traces while running containerized deep learning workloads. We first examine the event traces of CPU and memory systems and investigate the distribution of system calls to quantify potential bottlenecks and performance overhead. We also analyze the relationships between system calls and their triggering events.

4.1. Basic Event Analysis

We use the Linux perf tool to profile system calls and events generated by container processes, Docker daemons, and host processes to quantify the performance effects of each system configuration. The container process executes deep learning workloads on a container based on a virtualized form. The Docker daemon is a process that handles Docker API requests and manages containers. Specifically, it builds, runs, and manages images, containers, and other Docker objects. The host process runs deep learning workloads directly on the host OS without virtualization. Table 5 shows basic event and system call traces extracted from the container process, Docker daemon, and host process. For system calls, the most frequently invoked events and their number of invocations are listed. As we see from this table, most system calls are related to memory allocation and I/Os. In terms of CPU performance, running workloads in Docker containers increases CPU cycles by approximately 50% compared to running workloads directly on the host machine. As we can see, the pure container overhead caused by the daemon process is not significant, but various inefficiencies resulting from the addition of a layer of containers increase the overhead.

When analyzing block-related system calls in the container environment, we observe a significant number of block_rq_insert events. In particular, by analyzing the call graphs associated with these events in Table 6, we see that there is a significant number of write-back operations due to dirty pages within the container. This sync process flushes all dirty pages to storage, causing write-back operations to wait for the flush.

We also measure page-faults and swaps occurring on the container-based and host systems. As can be seen in Table 5, the number of page-faults in the container-based system is more than twice that of the host system. There are two types of page-faults: major faults, which involve swap I/O, and minor faults, which can be handled by kernel mapping without storage access [30]. The increased number of page-faults in container-based systems suggests that containers may be more susceptible to I/O performance problems than traditional OSs. This is because containers share the file system and host

OS, which can lead to contention and fragmentation. This results in poor performance and increased I/O load in container-based systems. To alleviate the performance degradation of page swapping, it is important to carefully manage the memory management algorithms allocated to containers. For example, it should be ensured that containers have sufficient memory and do not execute workloads that are known to generate a lot of page-faults. Also, it is necessary to generate different container runtimes to provide better isolation from the host system.

Table 5. Event and system call traces extracted from container and host processes.

	Event	Container Process	Container Daemon	Host Process
CPU	cpu_core/cycles/	461,729,070,471,469	124,892,660,806	306,402,857,705,993
	cpu_atom/cycles/	93,706,213,673,857	93,207,046,124	61,663,593,615,121
	cpu_core/instructions/	412,465,430,821,263	198,780,273,963	178,197,584,510,724
	cpu_atom/instructions/	106,333,305,694,655	112,349,586,140	9,714,425,305,632
	cpu_core/cache-misses/	1,375,684,053,246	1,760,922,473	641,571,962,797
	cpu_atom/cache-misses/	539,034,658,626	1,663,241,340	61,020,677,212
Memory	page-faults	1,685,309,171	9081	825,523,873
	minor-faults	342,203,571	9035	150,807,077
	major-faults	1,343,104,572	14	674,716,307
	cpu_core/mem-loads/	0	0	0
	cpu_atom/mem-loads/	1,688,256	0	3313
	cpu_core/mem-stores/	41,600,872,930,344	19,056,452,909	12,134,821,271,569
	cpu_atom/mem-stores/	11,791,995,380,030	10,442,649,500	511,089,910,826
System call	block:block_touch_buffer	281,259	1244	244,841
	block:block_dirty_buffer	267,489	269	159,348
	block:block_rq_complete	977,810	143	1,100,795
	block:block_rq_insert	48,881	4	28,958
	block:block_rq_issue	971,675	528	1,096,962
	block:block_bio_backmerge	39,741	436	51,398
	block:block_bio_frontmerge	428	0	114
	block:block_bio_queue	1,007,074	993	1,145,499

Table 6. Call graph of block request insert.

perf record -g -e block:block_rq_insert -p [pid] && perf report
. . .
17.77% kworker/u40:5 -f [kernel.kallsyms] [k] blk_mq_insert_requests
ret_from_work
kthread
worker_thread
process_one_work
wb_workfn
wb_do_writeback
+ wb_writeback

4.2. Dirty Pages and Write-Backs

In this subsection, we track the frequency of write-back system calls to compare the incidence of dirty pages across containers, host processes, and multi-tenant environments. In a multi-tenant environment, we run four containers simultaneously. We set the flush interval for our Linux system according to the environment variables in Table 7.

Table 7. Environment variables for flush operations.

/proc/sys/vm	
dirty_background_ratio	10
dirty_ratio	20
dirty_expire_centisecs	3000
dirty_writeback_centisecs	500

Figure 3 shows the frequency intervals of write-back system calls invoked in a single container, multiple containers, and host systems. As shown in the figure, containers call write-back functions more frequently than host processes on average. Also, when we compare the standard deviation of write-back call intervals in Figure 3, containers exhibit a much larger variance than the host system. This implies that the estimation of write-back system calls is more difficult in container-based deep learning, causing potential performance penalties. When comparing single and multi-tenant container environments, however, there is no significant difference in the write-back system call interval. This implies that multiple containers are certain to flush more dirty pages than a single container per each write-back system call. That is, as containers need to perform global data synchronization consistently, multi-tenancy inherently increases I/O traffic to storage even if it does not increase write-back frequency [31]. Specifically, storage write traffic will be increased with multiple containers, as each container needs to synchronize its data with the data of other containers.

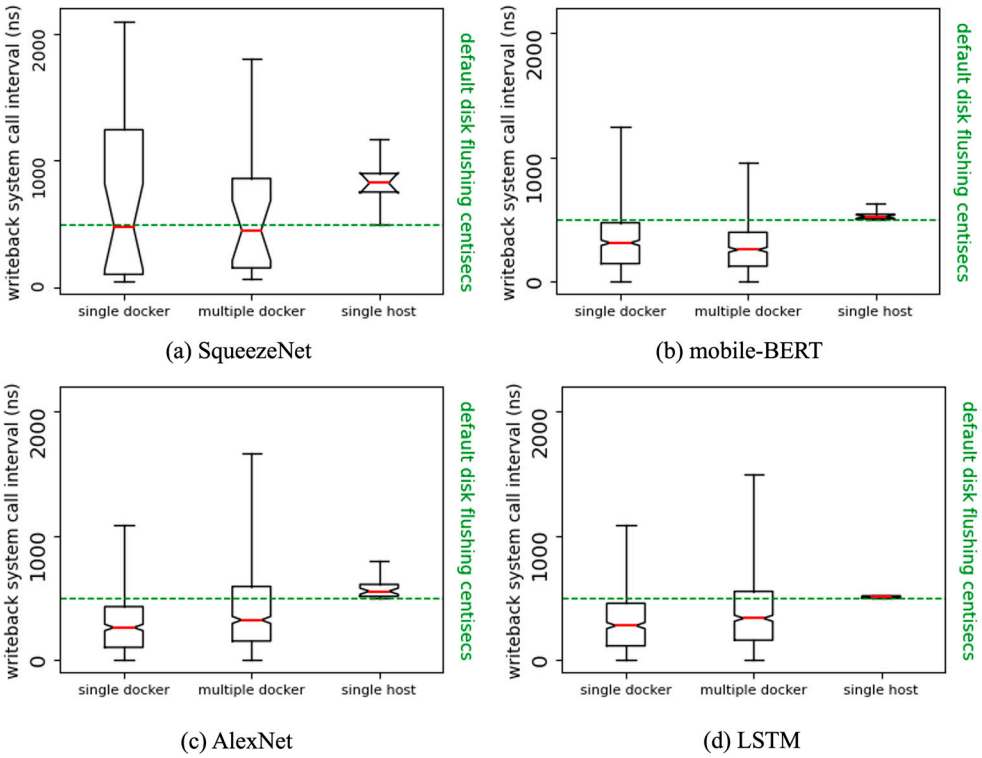


Figure 3. Comparison of sync system call intervals.

In our experiments, the write-back system call interval of Docker containers is shorter than the default storage flush interval listed in Table 7. This implies that dirty pages are generated more frequently in the container than with the default storage flush setting, resulting in more frequent sync system calls. Note that dirty pages have some modifications to the data after they are loaded into memory, so they need to be flushed to storage.

4.3. Page-Fault Analysis

Since page-faults and write-backs are quantified as significant causes of performance penalties in containerized deep learning, we further analyze memory and storage-related events using Linux Ftrace. Specifically, we investigate page-fault situations and their main reasons comprehensively. Table 8 shows the distribution of page-faults, and the files and functions that cause the page-faults. As we see from the table, a majority of page-faults are caused by a small number of specific files. In particular, the main sources of page-faults are found to be shared libraries, garbage collection, and regular expressions for path rules. Page-faults caused by shared libraries occur when an application loads a library file that is not resident in the main executable file. If the library is already in memory, this is a minor fault and only requires a re-mapping by the OS. If not, the library file needs to be loaded from storage and this is called a major fault. Garbage collection can also cause page-faults because garbage collection may result in the deallocation of memory that is still in use. Also, regular expressions for path rules can incur page-faults when complex expressions should be evaluated, which can lead to large memory space allocations.

Table 8. System call frequency distributions.

Rank	Task	File	Location	Cause	Percent
1	docker	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2	_dl_relocate_object	Shared library injection	6.2
2	docker-buildx	/usr/libexec/docker/cli-plugins/docker-buildx	runtime.memclrNoHeapPointers	Garbage Collection	5.1
3	docker	/usr/bin/docker	runtime.(*spanSet).push	Container start new command	4.5
4	docker-buildx	/usr/libexec/docker/cli-plugins/docker-buildx	runtime.(*spanSet).push	Container start new command	3.8
5	docker	/usr/bin/docker	runtime.memclrNoHeapPointers	Garbage Collection	3.0
6	docker-buildx	/usr/libexec/docker/cli-plugins/docker-buildx	runtime.memmove	Memory Copy	2.8
7	docker	/usr/bin/docker	runtime.memmove	Memory Copy	2.5
8	docker	/usr/bin/docker	runtime.getempty	Container first start command	1.9
9	ML process	/usr/libexec/docker/cli-plugins/docker-buildx	runtime.(*spanSet).push	Container start new command	1.6
10	docker-compose	/usr/libexec/docker/cli-plugins/docker-compose	0x6b239	Heap	1.5

To buffer the performance overhead caused by frequent storage flushes in container-based deep learning, NVM (non-volatile memory), also known as persistent memory, can be adopted. For example, eMRAM, a type of NVM for mobile systems, is being developed by Samsung Electronics [32]. If we flush hot dirty pages generated in Table 8 to the NVM, access to slow storage media can be eliminated. This can significantly accelerate the performance of container-based deep learning workloads, which will be discussed further in the next section.

5. Flushing Dirty Pages to Secondary Storage

In the previous section, we traced the statistics of dirty pages that were flushed to storage by kernel write-back calls while executing deep learning workloads. In this section, we further analyze the details of storage flush events. Specifically, we analyze the write-back activities as time progresses and the ratio of write-back triggering reasons (e.g., periodic, background, and sync). The most common cause of write-backs is “periodic,” which is

activated periodically on a schedule. These periodic write-backs ensure that dirty pages are flushed to storage within a certain time window regardless of the situation of the system. Another common cause of write-backs is "background," which is triggered when the ratio of dirty pages in memory exceeds a certain threshold. Note that periodic and background write-backs are not explicitly requested by a user space process but are performed by the kernel. In contrast, "sync" can be requested explicitly by a user process to flush all dirty data to storage, regardless of the reason for the write-back operation. Frequent invocation of "sync" may slow down the entire system so it should only be used if we need to flush all dirty data to storage.

Tables 9 and 10 show the frequency and ratio of write-backs based on call reasons in single and multiple Docker environments, respectively. As shown in Table 9, "periodic" accounts for the majority of write-backs (68.1–81.5%) in single Docker environments, but there is also a certain portion of "background" write-backs of 18.4–31.6%. Note that "background" write-backs are difficult to estimate, so they may degrade the performance of the entire system. Note also that the ratio of "background" write-backs grows even more in multi-tenant environments (by up to 42.1%) as shown in Table 10. This is because multi-tenants use more memory space, so "background" write-backs are triggered more frequently to reduce dirty pages in memory. Based on this result, we can conclude that multi-tenants incur more unpredictable write-backs, causing efficient management of flush I/Os difficult. Also, the differences observed between the two environments suggest that each requires a customized approach to manage the write-back process.

Table 9. Frequency and ratio of write-backs (single Docker).

	Background	Periodic	Sync
SqueezeNet	2400 (31.6%)	5182 (68.1%)	24 (0.3%)
Mobile-BERT	3040 (18.4%)	13,500 (81.5%)	20 (0.1%)
AlexNet	1674 (22.0%)	5866 (77.2%)	57 (0.8%)
LSTM	2157 (25.8%)	6155 (73.6%)	49 (0.6%)

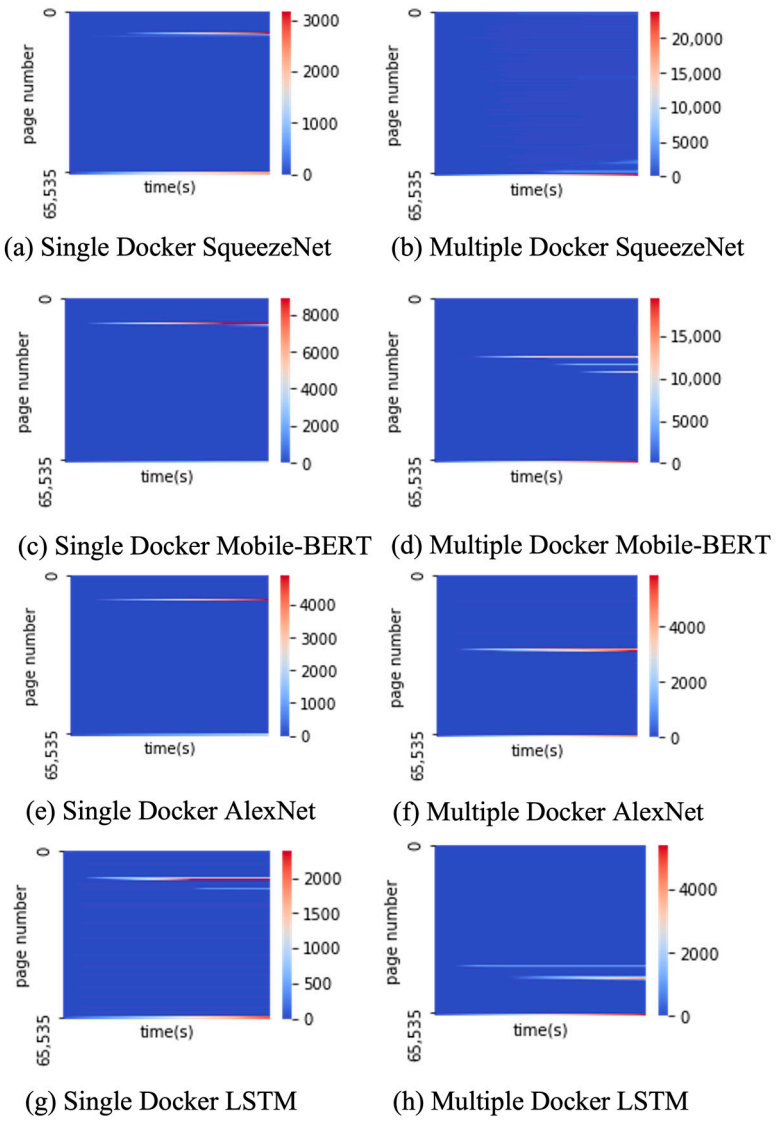
Table 10. Frequency and ratio of write-backs (multiple Dockers).

	Background	Periodic	Sync
SqueezeNet	23,753 (42.1%)	32,542 (57.6%)	180 (0.3%)
Mobile-BERT	19,466 (34.9%)	36,106 (64.8%)	186 (0.3%)
AlexNet	4884 (28.5%)	12,020 (70.2%)	208 (1.2%)
LSTM	5361 (35.0%)	9773 (63.8%)	186 (1.2%)

Reducing the number of write-backs can be achieved by appropriately setting the write-back parameters or using a file system that supports asynchronous writing. However, setting appropriate parameters in deep learning varies depending on the model and the number of tenants (i.e., single or multi-tenant).

To investigate the characteristics of write-back activities over time, we plot in Figure 4 the number of write-backs that occur for each memory page as time progresses. As shown in the figure, dirty pages that are being written back are evenly distributed across all pages in the early stages of deep learning, but after a certain time point, some limited pages are consistently flushed. This means that some limited data in deep learning is constantly modified, and it is necessary to efficiently manage these data flushes for stable performance of the deep learning training process. In the early stages of deep learning training, data is loaded, models are initialized, and weights are changed frequently. This means that there are a variety of dirty pages that need to be flushed back into storage [33]. However, as the model learns and the weights converge over time, the number of dirty pages decreases. That is, after a certain time point, the model repeatedly updates some limited data, which generates certain pages increasingly hotter as shown in Figure 4. Note that these hot

dirty pages can cause performance issues as they should essentially be flushed to storage frequently, wasting CPU and memory resources.



(a) Single Docker SqueezeNet (b) Multiple Docker SqueezeNet

(c) Single Docker Mobile-BERT (d) Multiple Docker Mobile-BERT

(e) Single Docker AlexNet (f) Multiple Docker AlexNet

(g) Single Docker LSTM (h) Multiple Docker LSTM

Figure 4. Write-back traces captured for each page number as time progresses.

To handle this situation, we suggest an intermediate non-volatile flushing layer residing between the main memory and the storage. Our architecture shown in Figure 5 has the mission of improving the write-back performance by making use of NVM as the front-end cache of secondary storage, leading to reduced I/O traffic and frequency of flush operations to slow disk storage. Specifically, our preliminary architecture utilizes NVMe as a secondary storage buffer to accelerate the write-back performances. As NVM is also a non-volatile medium like hard disks, we can eliminate storage flushing operations. To assess the effectiveness of this system architecture, we conduct simulation experiments for write-back activities in container-based deep learning workloads and compare the results of the original system and those with our NVM-added architecture. We use the parameters of a Toshiba DT01ACA1 hard disk drive (HDD) with a read/write access latency of 8 milliseconds for secondary storage. For NVM media, we use the parameters of a phase change memory (PCM) with a write latency of 350 nanoseconds. Note that PCM is a well-known NVM media that can be placed in front of slow storage to accelerate I/O performances [16].

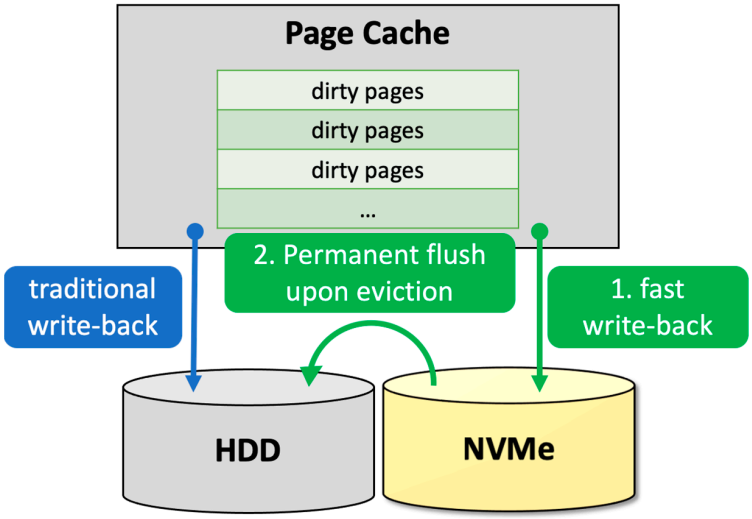


Figure 5. The proposed system architecture with the NVMe flushing layer.

Figure 6 shows the I/O latency of the proposed NVM-added architecture in comparison with the traditional system that does not use NVM as workloads and system situations are varied. As shown in the figure, our preliminary architecture improves the I/O latency significantly in all cases. Specifically, the improvements for SqeezeNet, Mobile-BERT, AlexNet, and LSTM, are 83%, 90%, 84%, and 77%, respectively, in single Docker and 78%, 77%, 86%, and 83%, respectively, in multi-tenant Docker environments. The improvement is the largest in the single Docker Mobile-BERT dataset. Note that Mobile-BERT is the smallest text dataset so most flush operations can be eliminated even with a small NVM capacity. We also observe that the improvement is large in multi-tenant environments with relatively heavy models such as Alexnet and LSTM because an increased number of write-backs in such models puts more strain on the HDD. In summary, our architecture has the effect of significantly improving I/O latency in containerized deep learning by absorbing storage flushing into an intermediate buffer between the container and slow HDD storage.

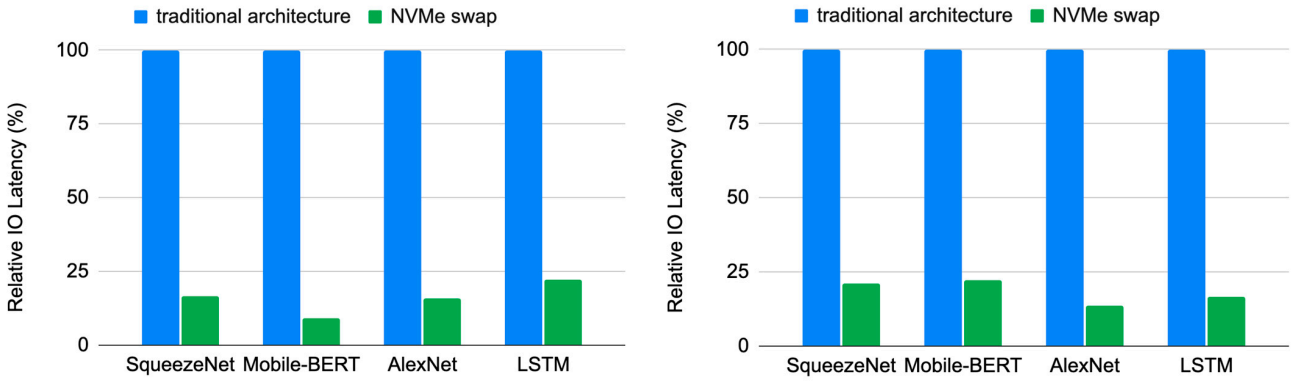


Figure 6. I/O latency improvement by adopting the proposed flushing layer. (**a**) Single Docker; (**b**) Multiple Dockers.

6. Conclusions

In this article, we quantified the resource requirements of containerized deep learning workloads through measurements and trace-based analysis. Specifically, we extracted and investigated the system event trace of container-based deep learning and compared it to traces collected by running the same workload on host systems to identify the overhead of

containerization and potential performance penalties. Based on our analysis, we observed that memory management, especially write-backs of dirty pages to storage, can be the main bottleneck in container-based deep learning. This is because containers share the host kernel and file systems, which can cause contention, and each container may have different synchronization intervals. By analyzing system calls and event traces generated by container-based deep learning, we provided implications for resource management in a containerized environment. We also introduced a preliminary solution that adopts an intermediate non-volatile flushing layer to alleviate the performance bottleneck of storage write-backs. Instead of flushing to traditional storage, our solution effectively hid the write-back overhead by absorbing hot data in fast NVM media, improving I/O latency by 82% on average.

In this article, we focused on analyzing the overhead of containerization in deep learning workloads executed in IoT environments that have limited resource capacities. Thus, we selected workloads that are relatively lightweight and suitable for deployment on edge devices rather than huge workloads where trace extraction imposes significant overhead on the system. In the future, we will extend our target architecture to high-performance systems that can support more complicated deep learning workloads, such as ResNet, NASNet, or GoogLeNet, to analyze resource usage patterns of containerized deep learning.

Author Contributions: S.P. implemented the architecture and algorithm, and performed the experiments. H.B. designed the work and provided expertise. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2021-0-02068, Artificial Intelligence Innovation Hub) and (No. RS-2022-00155966, Artificial Intelligence Convergence Innovation Human Resources Development (Ewha University)).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Veiga, T.; Asad, H.; Kraemer, F.; Bach, K. Towards containerized, reuse-oriented AI deployment platforms for cognitive IoT applications. *Future Gener. Comput. Syst.* **2023**, *142*, 4–13. [CrossRef]
2. Li, H.; Ota, K.; Dong, M. Learning IoT in edge: Deep learning for the internet of things with edge computing. *IEEE Netw.* **2018**, *32*, 96–101. [CrossRef]
3. Yu, R.; Li, P. Toward resource-efficient federated learning in mobile edge computing. *IEEE Netw.* **2021**, *35*, 148–155. [CrossRef]
4. Nam, S.A.; Cho, K.; Bahn, H. A new resource configuring scheme for variable workload in IoT systems. In Proceedings of the IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), Gold Coast, Australia, 18–20 December 2022; pp. 1–6. [CrossRef]
5. Kukreja, N.; Shilova, A.; Beaumont, O.; Huckelheim, J.; Ferrier, N.; Hovland, P.; Gorman, G. Training on the edge: The why and the how. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 899–903. [CrossRef]
6. Xiong, Y.; Sun, Y.; Xing, L.; Huang, Y. Extend cloud to edge with kubeedge. In Proceedings of the IEIEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018; pp. 373–377. [CrossRef]
7. Divya, V.; Sri, R.L. Docker-Based Intelligent Fall Detection Using Edge-Fog Cloud Infrastructure. *IEEE Internet Things J.* **2020**, *8*, 8133–8144. [CrossRef]
8. Desai, P.R. A survey of performance comparison between virtual machines and containers. *Int. J. Comput. Sci. Eng.* **2016**, *4*, 55–59.
9. Pahl, C.; Lee, B. Containers and clusters for edge cloud architectures—A technology review. In Proceedings of the 3rd IEEE International Conference Future Internet of Things and Cloud, Rome, Italy, 24–26 August 2015; pp. 379–386. [CrossRef]
10. Avino, G.; Malinverno, M.; Malandrino, F.; Casetti, C.; Chiasserini, C.F. Characterizing Docker Overhead in Mobile Edge Computing Scenarios. In Proceedings of the ACM HotConNet, Los Angeles, CA, USA, 25 August 2017; pp. 30–35. [CrossRef]

11. Boeira, C.; Neves, M.; Ferreto, T.; Haque, I. Characterizing network performance of single-node large-scale container deployments. In Proceedings of the 10th IEEE International Conference on Cloud Networking (CloudNet), Cookeville, TN, USA, 8–10 November 2021; pp. 97–103. [CrossRef]
12. Berral, J.; Wang, C.; Youssef, A. AI4DL: Mining Behaviors of Deep Learning Workloads for Resource Management. In Proceedings of the 12th USENIX HotCloud, Online, 13–14 July 2020.
13. Sharma, R.; Pachori, R.; Sircar, P. Automated emotion recognition based on higher order statistics and deep learning algorithm. *Biomed. Signal Process. Control* **2020**, *58*, 101867. [CrossRef]
14. Patalas-Maliszewska, J.; Halikowski, D. A Deep Learning-Based Model for the Automated Assessment of the Activity of a Single Worker. *Sensors* **2020**, *20*, 2571. [CrossRef] [PubMed]
15. Kwon, S.; Bahn, H. Classification and Characterization of Memory Reference Behavior in Machine Learning Workloads. In Proceedings of the IEEE/ACIS 23rd International Conference Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Taichung, Taiwan, 7–9 December 2022; pp. 103–108. [CrossRef]
16. Lee, J.; Bahn, H. Analyzing Memory Access Traces of Deep Learning Workloads for Efficient Memory Management. In Proceedings of the 12th International Conference Information Technology in Medicine and Education (ITME), Xiamen, China, 18–20 November 2022; pp. 389–393. [CrossRef]
17. Arlitt, M.F.; Williamson, C.L. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw.* **1997**, *5*, 631–645. [CrossRef]
18. Rauschmayr, N.; Kama, S.; Kim, M.; Choi, M.; Kenthapadi, K. Profiling Deep Learning Workloads at Scale using Amazon SageMaker. In Proceedings of the 28th ACM SIGKDD Conference Knowledge Discovery and Data Mining, Washington, DC, USA, 14–18 August 2022; pp. 3801–3809. [CrossRef]
19. Park, S.; Bahn, H. Memory Access Characteristics of Neural Network Workloads and Their Implications. In Proceedings of the IEEE Asia-Pacific Conference on Computer Science and Data Engineering, Gold Coast, Australia, 18–20 December 2022; pp. 1–6. [CrossRef]
20. Xu, P.; Shi, S.; Chu, X. Performance evaluation of deep learning tools in docker containers. In Proceedings of the 3rd International Conference Big Data Computing and Communications (BIGCOM), Chengdu, China, 10–11 August 2017; pp. 395–403. [CrossRef]
21. Bae, M.; Jeong, M.; Yeo, S.; Oh, S.; Kwon, O.-K. I/O performance evaluation of large-scale deep learning on an hpc system. In Proceedings of the International Conference High Performance Computing & Simulation (HPCS), Dublin, Ireland, 15–19 July 2019; pp. 436–439. [CrossRef]
22. Janecek, M.; Ezzati-Jivan, N.; Azhari, S.V. Container workload characterization through host system tracing. In Proceedings of the IEEE International Conference Cloud Engineering, San Francisco, CA, USA, 4–8 October 2021; pp. 9–19. [CrossRef]
23. Zhang, J.; Zhou, X.; Ge, T.; Wang, X.; Hwang, T. Joint task scheduling and containerizing for efficient edge computing. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 2086–2100. [CrossRef]
24. Mikolov, T.; Grave, E.; Bojanowski, P.; Puhrsch, C.; Joulin, A. Advances in pre-training distributed word representations. *arXiv* **2017**, arXiv:1712.09405.
25. Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the IEEE Conference Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255. [CrossRef]
26. Sun, Z.; Yu, H.; Song, X.; Liu, R.; Yang, Y.; Zhou, D. Mobilebert: A compact task-agnostic bert for resource-limited devices. *arXiv* **2020**, arXiv:2004.02984.
27. Sak, H.; Senior, A.W.; Beaufays, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In Proceedings of the Interspeech, Singapore, 14–18 September 2014; pp. 338–342.
28. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv* **2016**, arXiv:1602.07360.
29. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
30. Bahn, H.; Kim, J. Reducing the Overhead of Virtual Memory Swapping by Considering Application Characteristics and Memory Situations. In Proceedings of the 12th International Conference Information Technology in Medicine and Education, Xiamen, China, 18–20 November 2022; pp. 434–438. [CrossRef]
31. Gao, X.; Gu, Z.; Li, Z.; Jamjoom, H.; Wang, C. Houdini's escape: Breaking the resource rein of linux control groups. In Proceedings of the ACM SIGSAC Conference Computer and Communications Security, London, UK, 11–15 November 2019; pp. 1073–1086. [CrossRef]
32. Suh, K.; Lee, J.; Shin, H.; Lee, J.; Lee, K.; Hong, Y.; Han, S.; Kim, Y.; Kim, C.; Pyo, S.; et al. 12.5 Mb/mm^2 Embedded MRAM for High Density Non-volatile RAM Applications. In Proceedings of the IEEE Symposium on VLSI Technology, Kyoto, Japan, 13–19 June 2021.
33. Singh, B.; De, S.; Zhang, Y.; Goldstein, T.; Taylor, G. Layer-Specific Adaptive Learning Rates for Deep Networks. In Proceedings of the 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 9–11 December 2015; pp. 364–368. [CrossRef]