

Dynamic Random-Access Memory and Non-Volatile Memory Allocation Strategies for Container Tasks [†]

Che-Wei Chang ^{*} and Chen-Yu Ho

Department of Computer Science and Information Engineering, Chang Gung University, Taoyuan 333323, Taiwan

^{*} Correspondence: chewei@mail.cgu.edu.tw[†] Presented at 8th International Conference on Knowledge Innovation and Invention 2025 (ICKII 2025), Fukuoka, Japan, 22–24 August 2025.

Abstract

To support multimedia and deep learning applications running on containers within a server, both processor cores and main memory space are critical resources for performance tuning. With the growing memory demands of applications to maintain intermediate data, installing additional dynamic random-access memory (DRAM) modules increases not only hardware costs but also the static and dynamic energy consumption of a server. In this study, both DRAM and non-volatile memory (NVM) are leveraged to provide short access latency and large main memory capacity for a server running multiple containers with diverse applications. Contention for memory space and processor cores among containers is jointly modeled as part of the performance optimization problem for the hybrid memory system of the server. Our memory and computing resource scheduling algorithms are thus developed to judiciously balance the usage of cores and DRAM space among tasks, while NVM is utilized to increase the degree of parallelism to reduce the Makespan of task batches. Benchmark programs were used to generate the input task set, and experimental results show that our solution outperforms others by achieving at least an 18.34% reduction in Makespan when 100 distinct containerized tasks are executed on a system with 512 gigabytes (GB) of NVM, 32 GB of DRAM, and eight cores.

Keywords: non-volatile memory; heterogeneous memory; containers; MakespanAcademic Editors: Teen-Hang Meen,
Chun-Yen Chang and Cheng-Fu Yang

Published: 23 February 2026

Citation: Chang, C.-W.; Ho, C.-Y. Dynamic Random-Access Memory and Non-Volatile Memory Allocation Strategies for Container Tasks. *Eng. Proc.* **2025**, *120*, 68. <http://doi.org/10.3390/engproc2025120068>**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Virtual machines (VMs) offer a portable framework that allows users to install guest operating systems and run their applications with customized configurations on cloud servers. In VM-based environments, the presence of a hypervisor, such as VMware ESXi, kernel-based virtual machine, or Microsoft Hyper-V, is essential for offering comprehensive isolation, but it also consumes a relatively large amount of resources for the full system simulation. To reduce the memory, storage, and computing overheads of entire guest operating systems and the translation overheads of VMs [1], container technology proposes application programming interface (API)-level simulation for running applications requiring different APIs on different operating systems in separate packages. Unlike VMs, containers execute applications directly on the host system without the need to install separate guest operating systems or emulate entire hardware platforms [2]. Major cloud service providers, e.g., Aliyun Container Service [3], Amazon EC2 Container Service [4], and Azure Container Service [5], have adopted container technologies as a mainstream solution for cloud application deployment. However, with the rising demand from deep

learning and multimedia workloads, memory contention has emerged as a significant challenge in cloud server management, especially as DRAM reaches its physical limits in density scalability.

Resource management is a critical issue for ensuring the quality of each container instance, as well as for improving performance and reducing energy consumption in container servers. Further driven by the emergence of container-native orchestration frameworks from open-source communities, such as Kubernetes [6], Mesos Marathon [7], and Docker Swarm [8,9], various innovative scheduling algorithms have been proposed. A real-time management scheme [10] was developed to monitor the workload of each task. Based on the measured workloads, an optimization algorithm was proposed to dynamically schedule tasks for improving the efficiency of resource utilization. When multiple optimization factors have to be considered in cloud-edge environments for AI-based Internet of Things applications, a solution [11] was proposed to determine task priorities by considering both task urgency and resource requirements.

To optimize resource utilization and service quality for container-based workloads, a priority-aware scheduling algorithm [12] was proposed, which analyzes workload characteristics and behavioral patterns to enhance scheduling efficiency under the dynamic and uncertain conditions of cloud environments. Spillner [13] investigated memory management challenges in Function as a Service (FaaS) platforms. In such platforms, developers are required to explicitly specify memory configurations, and static overprovisioning can lead to considerable resource wastage. To address this, the study measured memory usage and analyzed trace logs to enable dynamic memory allocation adjustments. When available memory is insufficient to sustain the system, the Docker engine may terminate all containers, resulting in service interruption and significant performance degradation [14–16].

To address the demands of huge memory space, NVM, such as Intel Optane DC persistent memory [17], has been considered as a candidate to replace or work with DRAM for increasing the size of main memory with lower costs and static energy consumption. However, the long write latency of NVM is a critical issue for system performance. In heterogeneous memory systems combining NVM and DRAM, a runtime data management solution [18] has been proposed to bridge the performance gap by considering the bandwidth and latency sensitivity of different data objects. To utilize DRAM as a buffer and cache for data stored in NVM [19], the proposed optimization algorithm achieves up to 4 times higher read throughput and 2 times higher write throughput. Another approach is tiered memory-management systems [20]. By adapting to application-specific memory usage patterns, asynchronous operations were used to reduce CPU overhead. Klinkenberg et al. [21] investigated the challenges of managing data across heterogeneous memory architectures that incorporate high-bandwidth memory and NVM, and proposed a methodology enabling efficient data placement by allowing developers to specify data access patterns.

Cloud platforms enforce stringent resource reservation to ensure that high-performance memory is not utilized by non-priority tasks and to maintain the isolation of sensitive data. Due to the different read/write performance of heterogeneous memory, the total execution time of each container task can vary. In this work, we jointly consider the computing and memory resource requirements of each container task and account for the different execution times of each task when it runs on DRAM or NVM. With the objective to reduce the Makespan of container task batches, our scheduling algorithms have to monitor the computing and memory resource usage to balance the need for enough computing units and ample memory space with good performance.

2. Container Task Scheduling

We created the container-based heterogeneous resources scheduling (CHRS) algorithm to jointly manage DRAM, NVM, and computing units for supporting the execution of multiple containers. CHRS minimizes the Makespan of the container tasks' execution. The algorithm schedules tasks onto cores with the memory constraint and efficiently uses the remaining regions during the scheduling process. It balances the usage of DRAM and NVM with the scheduling results of tasks onto computing units.

2.1. System Model and Problem Definition

As multimedia and deep learning applications are prevalent on web platforms, the importance of managing and allocating computing and memory resources is much more pronounced. Node.js [22], with a single-thread model, is a JavaScript runtime on Chrome specifically designed for fast, scalable network applications running on containers. Additionally, the recent advance of Node.js applications in FaaS [23] further increases the popularity of Node.js. However, all of the existing instances of container technology do not provide a heterogeneous memory-management mechanism, even though operating systems allow each container task to run on DRAM or NVM with the support of non-uniform memory access (NUMA) nodes.

In this study, each container task t_i has the execution T_i^P on NVM and the execution time T_i^D on DRAM. When a task starts its execution, it locks the required computing and memory resources until it is completed to ensure stable performance and container-level isolation, and if it runs on DRAM, it consumes M_i memory space of DRAM. In a system, the DRAM size is denoted as D , the size of NVM is assumed to be large enough to accommodate all concurrent tasks, and the number of computing units (cores) is C . The algorithm partitions container tasks onto DRAM and NVM, schedules tasks without exceeding the size of DRAM or the number of computing units, and minimizes the Makespan of the tasks' execution for performance optimization.

2.2. Level-Approach Task Scheduling with Memory and Computing Resource Constraints

Algorithm 1 schedules container tasks onto computing units and allocates DRAM to support the task execution. First, Step 1 (resp. 2) sorts the input task set in a non-increasing order of their execution time on DRAM (resp. required DRAM space), and the algorithm selects tasks for scheduling by the two different indexing approaches in different situations. In our scheduling concept and following examples in Figure 1, the y -axis is for denoting the time, and multiple adjacent (in terms of time) levels are used for the task scheduling, in which a level has a floor (time) and a ceiling (time), and tasks scheduled into the level should not exceed the floor or ceiling. Therefore, Step 3 initializes the floor at time 0, and Steps 5 to 7 create a new level and set the ceiling and floor for the new level if it is needed.

If $\frac{D_r}{D} \leq \frac{C_r}{C}$, the remaining computing units in the current level are relatively plentiful. Therefore, the algorithm selects the first task t_i which is feasible to be scheduled at the current level from $\mathbf{N}_H$ in Steps 8 and 9. If $\frac{D_r}{D} > \frac{C_r}{C}$, the remaining DRAM space is relatively abundant. Thus, the algorithm selects the first task t_i which is feasible to be scheduled in the current level from $\mathbf{N}_W$ to consume more DRAM space at the current level in Steps 10 and 11. If the task t_i is found, it is scheduled into the current level, and the algorithm goes to Step 8 for finding the next-to-be-scheduled task. Otherwise, the algorithm goes to Step 5 to create a new level until all tasks are scheduled. During the scheduling process of Algorithm 1, there could be some remaining regions (denoted as Q), as illustrated in Figure 1a. Thus, Step 16 calls Algorithm 2 to further use the remaining DRAM and computing units in Region Q .

Algorithm 1: Level-Approach Task Scheduling onto Computing Units with DRAM Allocation

Input: A set of tasks $\mathbf{N}$, and a system with DRAM size D and C computing units

- 1 Sort $\mathbf{N}$ in a non-increasing order of T_i^D to get $\mathbf{N}_H$;
- 2 Sort $\mathbf{N}$ in a non-increasing order of M_i to get $\mathbf{N}_W$;
- 3 Set the floor L_F and the ceiling L_C as 0;
- 4 Let D_r and C_r be the remaining DRAM capacity and computing units;
- 5 **while** *there exist tasks to be scheduled* **do**
- 6 Select the first task t_i from $\mathbf{N}_H$; assign t_i to a new level; $\mathbf{N}_H \leftarrow \mathbf{N}_H \setminus \{t_i\}$; $\mathbf{N}_W \leftarrow \mathbf{N}_W \setminus \{t_i\}$;
- 7 Set the floor and ceiling of the new level by $L_F \leftarrow L_C$ and $L_C \leftarrow L_C + T_i^D$;
- 8 **if** $\frac{D_r}{D} \leq \frac{C_r}{C}$ *and* $\mathbf{N}_H \neq \emptyset$ **then**
- 9 Select the first task t_i from $\mathbf{N}_H$, which is feasible to be scheduled in the current level with the remaining DRAM and computing units;
- 10 **else**
- 11 Select the first task t_i from $\mathbf{N}_W$, which is feasible to be scheduled in the current level with the remaining DRAM and computing units;
- 12 **if** *such a task t_i exists* **then**
- 13 Pack t_i to the current level; $\mathbf{N}_H \leftarrow \mathbf{N}_H \setminus \{t_i\}$; $\mathbf{N}_W \leftarrow \mathbf{N}_W \setminus \{t_i\}$;
- 14 Define the remaining region from the top of t_i to the ceiling of the current level as Q ;
- 15 Call Algorithm 2 with Q ;
- 16 **goto** Step 8
- 17 **return** the result of task scheduling and resource usages;

Algorithm 2: Monotonic Stacking Scheme for Task Scheduling in a Region

Input: The input area and the currently remaining tasks and resources

- 1 **while** $\mathbf{N}_W \neq \emptyset$ **do**
- 2 Select the first task t_i from $\mathbf{N}_W$, which is feasible to be scheduled in the current region of the stacking scheme;
- 3 **if** *such a task t_i exists* **then**
- 4 Stack t_i on the top of the previous task;
- 5 $\mathbf{N}_H \leftarrow \mathbf{N}_H \setminus \{t_i\}$; $\mathbf{N}_W \leftarrow \mathbf{N}_W \setminus \{t_i\}$;
- 6 Recursively define the remaining region from the bottom of t_i to the ceiling of the current level and from the right most of t_i to the right most of the previous task as R ;
- 7 **if** $\frac{D_r}{D} \leq \frac{C_r}{C}$ **then** call Algorithm 2 with R ;
- 8 **else**
- 9 **break**;
- 10 **return** the result of task scheduling and resource usages;

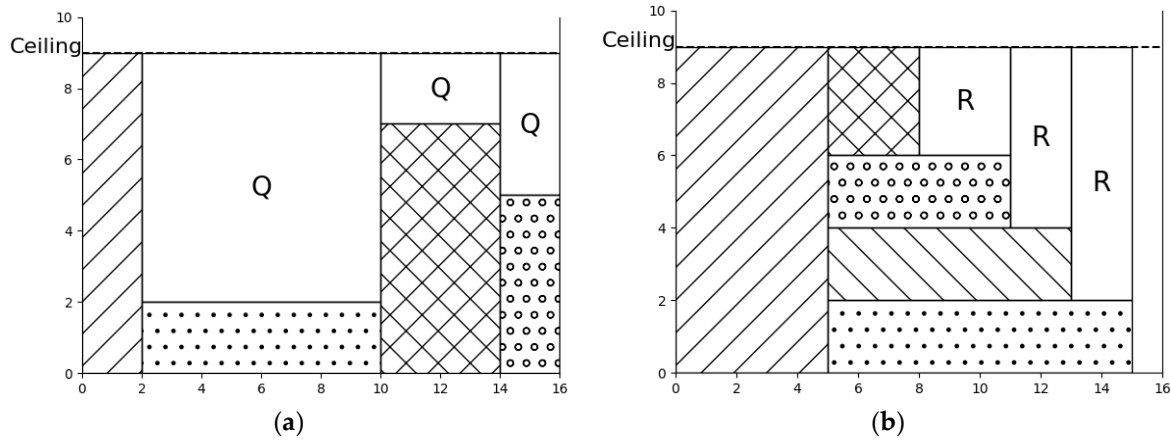


Figure 1. (a) Illustration of Region Q; (b) illustration of Region R.

2.3. Recursive Strategy for Using Remaining Regions of Computing and Memory Resources

Algorithm 2 develops a recursive strategy to use the remaining DRAM and computing units of Regions Q and R, which are illustrated in Figure 1. In Algorithm 1, a Region Q is created because a scheduled task occupies some DRAM space of the current level and is completed before the ceiling (time) of this level, as shown in Figure 1a. Therefore, the idea of Algorithm 2 is to find some other tasks to use the DRAM and computing resources of the scheduled task after the scheduled task is completed until the ceiling of this level. Thus, from Step 2 to Step 5, Algorithm 2 tries to find feasible tasks and stack them into the region. However, the task stacking process of Algorithm 2 recursively creates some other remaining regions denoted as Region R, as shown in Figure 1b. If we schedule a new task into Region R, the new task will consume one more computing unit in the current level, which is different from the situation of Region Q. Thus, in Algorithm 2, if we want to recursively call Algorithm 2 for using the resources of a Region R (in Step 8), we have to pass the test in Step 7 that ensures that the remaining computing units are relatively plentiful.

2.4. Overall Scheme for Managing DRAM, NVM, and Computing Units

Algorithm 3 calls Algorithm 1 for the task scheduling with DRAM and uses a best-fit fashion for the task scheduling on NVM. Steps 2 and 3 initially assign all tasks to run on DRAM and derive the Makespan. All tasks are then sorted in a non-decreasing order of DRAM affinity to obtain $\mathbf{D}$, where the DRAM affinity γ_i of a task t_i is defined as follows: $\gamma_i = \frac{T_i^P - T_i^D}{M_i}$. A task with a larger DRAM affinity is preferred to run on DRAM because of its smaller memory consumption and/or high performance improvement when it is moved from NVM to DRAM. The remaining part of Algorithm 3 iteratively attempts to move each task to NVM and updates the Makespan whenever the scheduling result changes.

Algorithm 3: Container-based Heterogeneous Resources Scheduling (CHRS)

Input: A set of tasks $\mathbf{N}$, and a system with DRAM size D and C computing units

- 1 Initialize the update flag $f \leftarrow \text{true}$;
- 2 Assign all tasks to DRAM for the initial assignment;
- 3 Run Algorithm 1 to get the Makespan m_D of tasks on DRAM and initialize the Makespan of tasks on NVM $m_P \leftarrow 0$;
- 4 Derive the DRAM affinity γ_i for each task $t_i \in \mathbf{N}$;
- 5 Sort tasks in $\mathbf{N}$ in an on-decreasing order of the DRAM affinity to get $\mathbf{D}$;
- 6 **while** $f = \text{true}$ **do**
 - 7 $f \leftarrow \text{false}$;
 - 8 **for each task** $t_i \in \mathbf{D}$ **do**
 - 9 Try to move t_i to NVM;
 - 10 Run Algorithm 1 to get the Makespan m'_D of the tasks on DRAM;
 - 11 Use a best-fit fashion to sequentially allocate the remaining time slots of computing units to run tasks on NVM with the Makespan m'_P ;
 - 12 **if** $\max\{m'_D, m'_P\} > \max\{m_D, m_P\}$ **then** withdraw the movement of t_i ;
 - 13 **else**
 - 14 $m_D \leftarrow m'_D$; $m_P \leftarrow m'_P$; $f \leftarrow \text{true}$;
- 15 **return** the scheduling results with DRAM and NVM and the Makespan $\max\{m_D, m_P\}$;

3. Performance Evaluation*3.1. Environment Setup and System Configuration*

In the experiments, we referred to the system summarized in Table 1 for hardware resource constraints and for generating the execution time and memory consumption on DRAM and NVM.

Table 1. System used for the experiment.

Item	Specification
Operating system	Red Hat Enterprise Linux 9.0
Kernel version	5.14.0-70.30.1.el9_0.x86_64
Container	Podman 4.1.1
Processor	Intel Xeon Gold 6240 CPU @ 2.60GHz
Memory	32 GB DDR4 DRAM 512 GB Intel Optane DC persistent memory

In the Dell server running Red Hat Enterprise Linux (RHEL) 9.0 with Intel DC persistent memory (as the NVM instance in our experiments), we limited the DRAM usage and NVM usage to no more than 32 GB and 512 GB, respectively. A group of benchmark suites, including Phoronix Test Suite, Iozone, sysbench, and STREAM, was tested. For measuring the performance gap of a task running on NVM and DRAM, we defined the performance ratio of a task as the execution time on NVM divided by the execution time on DRAM. In all of our tests, the minimum performance ratio is 1.01, which was measured from sysbench, and the maximum performance ratio is 26.87, which was measured from STREAM. In the tests, sysbench executed a CPU-bound program with minimal memory accesses, whereas STREAM ran a program characterized by intensive memory accesses. When we had to generate synthetic tasks, 1.01 (resp. 26.87) was then used as the lower bound (resp. upper bound) of the performance ratio of each task, and the upper bound and lower bound of the execution time on DRAM were set to 16.00 s and 0.01, respectively. For each experiment, 100 tests were conducted, and the averages were reported as the results.

3.2. Evaluated Solutions

Since there is no existing algorithm that can manage the heterogeneous memory, including NVM and DRAM, in the container environment for task scheduling, to make the comparison for the performance evaluation, classical algorithms for the 2D strip packing problem were ameliorated to account for the constraints of the studied problem. Furthermore, two random approaches were included to illustrate the baseline performance with and without the level concept during the scheduling process. For all solutions, the task scheduling on NVM is managed by Algorithm 3, where we assume that the NVM is always large enough to accommodate all concurrent tasks. All of the evaluated solutions are listed as follows.

- CHRS: This is the proposed solution of this paper, where Algorithm 3 balances the loading on heterogeneous memory and uses Algorithms 1 and 2 for the task scheduling on DRAM for the overall Makespan reduction.
- First-fit decreasing height (FFDH): This solution sorts all tasks in a non-increasing order of their heights (the execution time with DRAM) and schedules tasks one by one. We modified the original FFDH to account for both the DRAM size and the number of cores. It schedules a task to a level in a first-fit fashion. If there is no feasible level to accommodate a new task, a new level is created for the task.
- Best-fit decreasing height (BFDH): The behavior of BFDH is similar to FFDH. The difference is that BFDH selects the level with the minimum remaining DRAM space from all feasible levels for accommodating a new task.
- Fixed-level random: This solution randomly selects a task and tries all levels one by one to schedule the task into a level with enough remaining DRAM space and cores to accommodate the task, and where the height of the task is not greater than the height of the level. If there is no feasible level, a new level is created to accommodate the task, and the height of the level is equal to the height of the task.
- Flexible-level random: This solution is similar to fixed-level random. The difference is that this solution can assign a task to an existing level even though the task is higher than the level. In this case, the ceiling of the level is extended to the height of the task.

3.3. Results

In the first experiment, different numbers of tasks were tested to evaluate the performance of all solutions with different total workloads (Figure 2a). The number of cores and the DRAM size were fixed at eight cores and 32 gigabytes, respectively, and the number of tasks was increased from 10 to 100 by a step of 10.

When the number of tasks is no less than 20, CHRS outperforms all of the other solutions. When the number of tasks is 10, the performances of FFDH and BFDH are slightly better than that of CHRS. With only 10 tasks, the memory contention issue is not quite significant, and thus, FFDH and BFDH, by reducing the height of each level of the 2D strip packing, reduce the Makespan (in this case, only two levels were used in the task scheduling). For the tests with 20 tasks (resp. 100 tasks), CHRS outperforms FFDH and BFDH by 9.95% (resp. 18.34%) of the Makespan reduction.

In the second experiment (Figure 2b), the number of tasks was fixed at 100, and the number of cores was varied from 1 to 15. The result shows the sensitivity of our solution to the number of cores. Again, when the number of cores is less than four, FFDH and BFDH are slightly better than CHRS, but when the number of cores is no less than four, CHRS can consistently outperform all of the other solutions. When the number of cores is more than four, the performance gap between CHRS and the other solutions is no less than 7.38%.

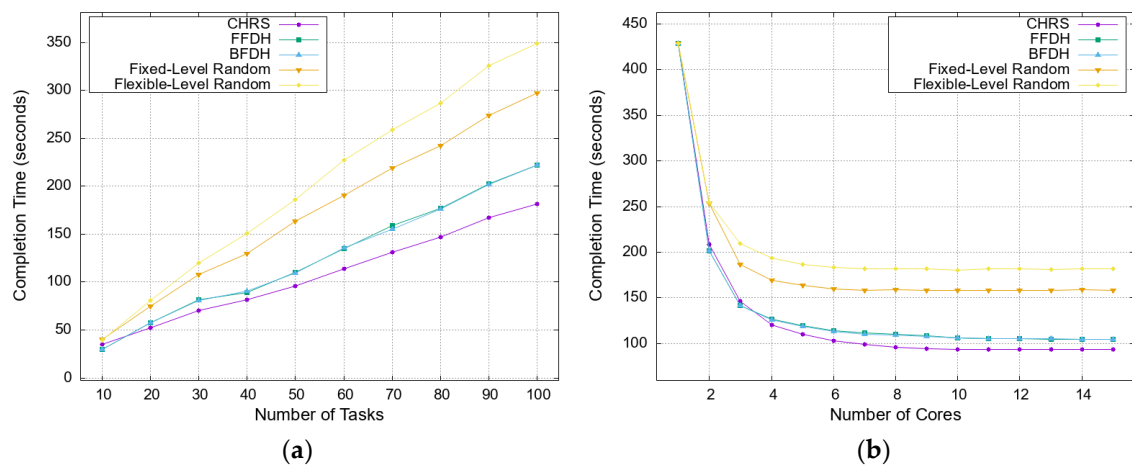


Figure 2. (a) Evaluation of all solutions with different numbers of tasks; (b) evaluation of all solutions with different numbers of cores.

4. Conclusions

To meet the demand for large memory space with limited hardware cost and energy consumption on container servers, NVM is used to work with DRAM as the main memory. This method is promising in serving applications with huge memory footprints. However, there is no existing container implementation supporting heterogeneous memory management with DRAM and NVM. This paper includes the level concept for task scheduling, and the three algorithms are developed to schedule tasks into levels with memory and computing resource constraints, efficiently use the regions' remaining resources, and balance the workloads on DRAM and NVM. By measuring the performance and memory consumption of benchmark programs on our server with Intel Optane DC persistent memory, synthetic tasks were generated to evaluate the capability of our solution with different computing workloads, hardware resources, and application memory requirements. Experimental results show that the developed method indeed outperforms the other solutions when both memory and computing resource contentions are significant in a system. The popular Node.js single-thread model runs on container environments with DRAM and NVM. The multiple-thread models of other container-based application environments are also promising.

Author Contributions: Conceptualization, C.-W.C. and C.-Y.H.; methodology, C.-W.C. and C.-Y.H.; software, C.-Y.H.; validation, C.-W.C. and C.-Y.H.; formal analysis, C.-W.C. and C.-Y.H.; investigation, C.-W.C. and C.-Y.H.; resources, C.-W.C. and C.-Y.H.; data curation, C.-Y.H.; writing—original draft preparation, C.-Y.H.; writing—review and editing, C.-W.C. and C.-Y.H.; visualization, C.-W.C. and C.-Y.H.; supervision, C.-W.C.; project administration, C.-W.C.; funding acquisition, C.-W.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by Chang Gung University and Chang Gung Hospital under grant Nos. NERPD2M0493, NERPD2Q0251, and BMRPD84, and by the National Science and Technology Council under grant Nos. 111-2221-E-182-039-MY3, 111-2923-E-002-014-MY3 and 114-2221-E-182-010-MY3.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

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

References

1. Chhajer, S.; Thyagaturu, A.S.; Yatavelli, A.; Lalwaney, P.; Reisslein, M.; Raja, K.G. Hardware accelerations for container engine to assist container migration on client devices. In Proceedings of the IEEE International Symposium on Local and Metropolitan Area Networks, Orlando, FL, USA, 13–15 July 2020.
2. Xu, X.; Yu, H.; Pei, X. A novel resource scheduling approach in container based clouds. In Proceedings of the IEEE 17th International Conference on Computational Science and Engineering, Chengdu, China, 19–21 December 2014.
3. Aliyun. Alibaba Cloud Container Compute Service. 2024. Available online: https://www.alibabacloud.com/en?_p_lc=1 (accessed on 1 July 2024).
4. Amazon. Amazon EC2 Container Service. Available online: <https://aws.amazon.com/ecs/> (accessed on 1 July 2024).
5. Microsoft. Azure Container Service. 2024. Available online: <https://azure.microsoft.com/en-us/services/container-service/> (accessed on 1 July 2024).
6. Bernstein, D. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [CrossRef]
7. Saha, P.; Govindaraju, M.; Marru, S.; Pierce, M.E. Integrating Apache Airavata with Docker, Marathon, and Mesos. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 1952–1959. [CrossRef]
8. Cérin, C.; Menouer, T.; Saad, W.; Abdallah, W.B. A new Docker Swarm scheduling strategy. In Proceedings of the IEEE International Symposium on Cloud and Service Computing, Kanazawa, Japan, 22–25 November 2017.
9. Vieux, V.; Luzzardi, A.; Totla, N.; Chen, D.; Xian, J.; Beslic, A.; Kaewkasi, C.; Sun, A.; Prasad, A.; Firshman, B.; et al. Classic Swarm: A Docker Native Clustering System. 2018. Available online: <https://github.com/docker-archive/classicswarm> (accessed on 1 July 2024).
10. Muniswamy, S.; Vignesh, R. Dsts: A hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment. *J. Cloud Comput.* **2022**, *11*, 33. [CrossRef]
11. Tang, B.; Luo, J.; Obaidat, M.S.; Vijayakumar, P. Container-based task scheduling in cloud-edge collaborative environment using priority aware greedy strategy. *Clust. Comput.* **2023**, *26*, 3689–3705. [CrossRef]
12. Zhu, L.; Huang, K.; Fu, K.; Hu, Y.; Wang, Y. A priority-aware scheduling framework for heterogeneous workloads in container-based cloud. *Appl. Intell.* **2023**, *53*, 15222–15245. [CrossRef]
13. Spillner, J. Resource management for cloud functions with memory tracing, profiling and autotuning. In Proceedings of the 6th International Workshop on Serverless Computing, Delft, The Netherlands, 7–11 December 2020.
14. Mao, Y.; Oak, J.; Pompili, A.; Beer, D.; Han, T.; Hu, P. DRAPS: Dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster. In Proceedings of the IEEE 36th International Performance Computing and Communications Conference, San Diego, CA, USA, 10–12 December 2017.
15. Azab, A. Enabling docker containers for high-performance and many task computing. In Proceedings of the IEEE International Conference on Cloud Engineering, Vancouver, BC, Canada, 4–7 April 2017.
16. Torre, R.; Urbano, E.; Salah, H.; Nguyen, G.T.; Fitzek, F.H.P. Towards a better understanding of live migration performance with Docker containers. In Proceedings of the 25th European Wireless Conference, Aarhus, Denmark, 2–4 May 2019.
17. Intel. Intel Optane DC Persistent Memory. 2020. Available online: <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html> (accessed on 1 July 2024).
18. Wu, K.; Huang, Y.; Li, D. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2017.
19. Zhao, X.; Challa, P.; Zhong, C.; Jiang, S. Developing index structures in persistent memory using spot-on optimizations with dram. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, London, UK, 7–11 May 2024.
20. Raybuck, A.; Stamler, T.; Zhang, W.; Erez, M.; Peter, S. Hemem: Scalable tiered memory management for big data applications and real NVM. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual, 26–29 October 2021.
21. Klinkenberg, J.; Kozhokanova, A.; Terboven, C.; Foyer, C.; Goglin, B.; Jeannot, E. H2m: Exploiting heterogeneous shared memory architectures. *Future Gener. Comput. Syst.* **2023**, *148*, 39–55. [CrossRef]
22. Chhetri, N. A Comparative Analysis of Node.js (Server-Side Javascript). Master's Thesis, St. Cloud State University, St. Cloud, MI, USA, 2016.
23. de Carvalho, L.R.; de Ara'ujo, A.P.F. FaaS-oriented node.js applications in an RPC approach using the node2faas framework. *IEEE Access* **2023**, *11*, 112027–112043. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.