applied sciences

MDPI

Article

A Lightweight File System Design for Unikernel

Kyungwoon Cho [1] and Hyokyung Bahn [2,*]

[1] Embedded Software Research Center, Ewha University, Seoul 03760, Republic of Korea; cezanne@ewha.ac.kr
[2] Department of Computer Engineering, Ewha University, Seoul 03760, Republic of Korea
* Correspondence: bahn@ewha.ac.kr; Tel.: +82-2-3277-4247

Abstract: Unikernels are specialized operating system (OS) kernels optimized for a single application or service, offering advantages such as rapid boot times, high performance, minimal memory usage, and enhanced security compared to general-purpose OS kernels. Unikernel applications must remain compatible with the runtime environment of general-purpose kernels, either through binary or source compatibility. As a result, many Unikernel projects have prioritized system call compatibility over performance enhancements. In this paper, we explore the design principles of Unikernel file systems and introduce a new file system tailored for Unikernels named ULFS (Ultra Lightweight File System). ULFS provides system call services akin to those of general-purpose OS kernels but achieves superior performance and security with significantly fewer system resources. Specifically, ULFS is developed as a lightweight file system embracing Unikernel design principles. It streamlines system calls, removes unnecessary locks, and omits permission checks for multiple users, utilizing a non-hypervisor architecture. This approach significantly reduces the memory footprint of the file system and enhances performance. Through measurement studies, we assess the performance and memory requirements of various file systems from major Unikernel projects. Our findings demonstrate that ULFS surpasses several existing Unikernel file systems, including Rumpvfs, Ramfs-u, Ramfs-q, 9pfs, and Hcfs.

Keywords: Unikernel; file system; lightweight; non-hypervisor; ULFS; lock-free file system; ULV

check for updates

Citation: Cho, K.; Bahn, H. A Lightweight File System Design for Unikernel. *Appl. Sci.* **2024**, *14*, 3342. https://doi.org/10.3390/app14083342

Academic Editor: Arkadiusz Gola

Received: 29 February 2024
Revised: 12 April 2024
Accepted: 12 April 2024
Published: 16 April 2024

Copyright: © 2024 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

Unikernels represent a paradigm shift in operating system (OS) kernel design, tailored for optimizing a single application or service [1]. Compared to general-purpose kernels, these specialized kernels have the advantages of fast booting, high performance, minimal memory footprint, and fortified security [2]. Unikernels have garnered widespread adoption across a variety of domains, including network function virtualization [3], data processing [4], IoT (Internet of Things) [5,6], and edge computing [7,8]. Although some applications may need to be newly built for running on the Unikernel, applications developed for a general-purpose OS can be executed directly on the Unikernel if certain conditions are met. Specifically, if the application binary interface (ABI) of the same executable file format is used, or the application programming interface (API) is matched to ensure execution through source-level build, the application can be executed as is in the Unikernel.

From an application perspective, it is important that Unikernel supports the same system call services as a general-purpose OS kernel. In fact, Unikernels do not need to support all system call services in general-purpose OS kernels, but system calls for some core functions such as networking and file systems must be provided. In the case of file systems, most Unikernel projects have primarily focused on implementing APIs just to ensure the portability of applications. Consequently, they have either adopted a general file system or relied on the host's file system. Even in cases where a file system was developed from scratch or slightly modified for Unikernels, easily implementable approaches such as in-memory file systems are preferred. Performance optimization and security have not been major concerns.

When considering the design philosophy of Unikernels, performance optimization and security issues are important for a file system with minimal system resource usage. Specifically, setting appropriate configurations for file system I/Os is important to achieve optimized performance for a given application with minimal resource usage.

In this paper, we design and implement a new file system for Unikernel called ULFS (Ultra Lightweight File System), which focuses on improving performance based on design simplification. To evaluate this claim, we compare the performance of ULFS with existing file systems developed in various Unikernel projects through a variety of benchmarks and analyze issues in file system design. Our measurement studies show that ULFS outperforms several existing Unikernel file systems, including Rumpvfs, Ramfs-u, Ramfs-q, 9pfs, and Hcfs.

The remainder of this paper is organized as follows. Section 2 briefly reviews existing research pertinent to the goals of this paper. Section 3 describes the design issues of Unikernel file systems. Sections 4 and 5 describe the details of the ULFS proposed in this paper and evaluate its performance through measurement studies. Finally, Section 6 presents the conclusion of this paper.

2. Related Works

File systems deployed within Unikernels typically range from straightforward, memory-based systems to more complex arrangements that provide access to host files via virtualization, mediated by a hypervisor. A file system tailored for Unikernels is expected to deliver superior I/O performance while maintaining a minimal memory footprint, to align with its foundational objectives. At the same time, it is crucial that the file system be exempt from security vulnerabilities.

There have been efforts to adapt file systems for Unikernel environments, tailored to meet the unique requirements of each distinct project. The Nabla project, for instance, incorporates the Rumpkernel, a library operating system, to execute Unikernel applications via its dedicated hypervisor, the Solo5 tender [9]. Rumpkernel basically supports an in-memory file system called Rumpvfs as the root file system [10,11]. Based on Rumpvfs, they develop another file system called Rump_etfs (Extra Terrestrial File System), which provides linkage of files on the host to the Unikernel without the assistance of hypervisors. Structurally, Rump_etfs is similar to the ULFS proposed in this paper, as it does not use a hypervisor. However, as it is based on an in-memory file system, Rumpvfs, it does not follow well the lightweight design principle of Unikernels.

Unikraft is one of the most active Unikernel projects currently being developed [12]. It supports an imported 9pfs to bind files on the host to Unikernel as well as the customized in-memory file system Ramfs [13]. Unikraft also provides an application build tool called Kraft for easy creation and execution of Unikernel applications. Moreover, it supports various software platforms for Unikernel applications, including hypervisors such as Qemu and Xen, as well as public cloud platforms such as AWS and GCP. The Linuxu (Linux user-space) platform supports Unikernels to run directly on the host OS without a hypervisor. Although its performance is not competitive when compared with other platforms, Linuxu is widely used in the application development phase as various developer tools (e.g., debugger) inside the host can be utilized. Ramfs is available on both the Linuxu and Qemu platforms, but 9pfs only works on the Qemu platforms. In summary, Unikraft's file system is weaker than ULFS in terms of performance and resource usage because it either borrows the file system for virtual machine environments (9pfs) or customizes the existing in-memory file system Ramfs for Unikernel.

Hermitux is another Unikernel that can be executed directly on existing general-purpose kernels without being built again, so much research has been performed on it [14]. Hermitux provides its own in-memory file system called Minifs, but its use is limited in actual application environments as it only provides some bare-bones functions. The main file system of Hermitux is Hcfs (Hermit Core File System), which bypasses its own hypervisor Uhyve, and binds its files to the file system on the host at the API level. However,

this has the same effect as a system call to the host, which accompanies the overhead of hypercalls whenever a file system-related system call is made. Also, as Hcfs needs the support of host file systems, it has limitations in terms of security and performance. ULFS accesses host storage as a block device rather than making use of file system-level hypercalls to the host. Table 1 shows a comparison between ULFS and other Unikernel file systems under various concerns.

Table 1. Comparison of ULFS with existing Unikernel file systems.

File System	Target Project	Design Style	Need for Hypervisor	Implementation Approach	Limitations
Rumpvfs	Nabla (2018)	In-memory file system	No	Imported from Rump kernel without custom development	· Low performance due to adopting general purpose file system · No persistent storage
Ramfs	Unikraft (2021)	In-memory file system	Yes	Developed with Unikernel in mind but encompass too many features	· Low performance due to hypercalls and complex design · High memory requirement · No persistent storage
9pfs	Unikraft (2021)	Host-based file system	Yes	Imported from Qemu without custom development	· Low performance due to frequent hypercalls · Low security due to direct exposure to host
Hcfs	hermitux (2019)	Host-based file system	Yes	Redirect guest file system API requests to host	· Overhead of API request redirection · No isolation guaranteed as all file operations performed at host
ULFS	ULV (2022)	Hybrid file system	No	Developed from scratch with streamlined design	· Unfit for large scalable workload

In contrast to the aforementioned file systems, there exist instances where emphasis is placed on developing file system features specifically optimized for applications such as ETL (Extract, Transform, and Load), thereby prioritizing optimization over adherence to Posix compliance. Fingler et al. proposed a Unikernel technology for ETL applications instead of a containerized method, which is widely used for serverless functions [4]. They modified the Python interpreter to support HTTP GET/POST requests tailored for ETL. By ensuring compatibility between the higher-level API of the Unikernel and the host, they removed unnecessary networking and storage functionalities.

3. Design Issues of Unikernel File Systems

3.1. Classification of Unikernel File Systems

Unikernel file systems can be classified into three types, as shown in Figure 1, depending on the storage interconnection method. The first is a type of in-memory file system in which all data in the file system is managed within the Unikernel without host intervention. File systems in this category have the advantage of providing performance comparable to general-purpose operating systems. Also, there is a merit to security as all file system data are hidden on the host and managed by the Unikernel. However, as in-memory file systems do not have persistency features (i.e., their contents will not be retained when the power is turned off), applications need to back up file system contents through additional I/O or network operations explicitly to preserve data. Therefore, file systems in this category are mainly used for maintaining temporary files, which are necessary only when the Unikernel is activated. This type of file system also has the drawback of large memory consumption for the file system, which is not desirable for Unikernels that aim at being lightweight.

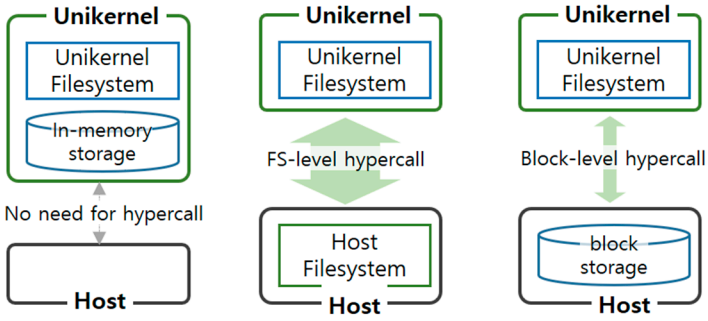

Figure 1. Architectural classification of Unikernel file systems.

The second type is a host-based file system that maps files in the Unikernel file system to files on the host or forwards API requests from the Unikernel file system to the host. In this type, file system storage is entirely dependent on the host side. Unlike in-memory file system types, applications that use these file system types can utilize the file system for persistent data I/O. Also, the memory space consumed by this file system type is less than that of the in-memory file system type. However, hypercall operations for this file system type become complicated as the Unikernel's file system needs to communicate with the host file system for each file operation. There is also a security problem such that files used by guests are visible to the host. Moreover, although the memory consumption of the file system is less than that of the in-memory file system, in general, it is difficult to control the memory usage of the host file system, possibly leading to increased memory usage.

The third type of file system makes use of a hybrid approach where the host provides a dedicated storage device to the Unikernel, and the Unikernel performs file services by making use of the storage on the host. Unlike the host-based file system type, a block device-level interface is used for this type, so hypercall operations between the host and Unikernel can be simplified. Also, it has the advantage of minimizing the vulnerability of guest file data being exposed to the host.

3.2. Hypervisor vs. Non-Hypervisor

In traditional Unikernel environments, isolation between tenants is achieved by running a hypervisor on the host and placing virtual machines for each application on top of it. However, even if a Unikernel application is executed as a process on the host without a hypervisor, it can provide security similar to a virtual machine. In this non-hypervisor environment, process isolation without a virtual machine is used, so guests can access the host's resources using system calls without hypercalls.

Meanwhile, the number of system calls is much greater than that of hypercalls, providing unnecessarily many functions to guests. This results in widening the attack surface in terms of security. We can resolve this issue by limiting the range of system calls and system arguments allowed to the guest, thereby activating only essential functions for Unikernel guests [15]. To do so, ULFS restricts the execution of system calls to only `pwrite64` and `pread64` through the Linux seccomp (secure computing) facility [15]. Moreover, although these system calls are invoked, ULFS does not permit execution if the file descriptor of the backend storage does not correspond to a block device. Since system calls that modify memory segments (such as `mprotect`) are not allowed, malicious guest code is confined to reading from the code segment or writing to the data segment, effectively safeguarding the host from potential harm. As `pwrite` and `pread` maintain the same interface even when executed as hypercalls, the attack surface is identical to that in virtualized environments. While there is potential for security vulnerabilities in the implementation of system calls, non-hypervisors are not likely to increase the potential security risks associated with implementing system calls compared to hypervisor environments, especially when considering the stability of the host operating system.

In hypervisor-based system architectures, file system operations impose a significant performance burden. This is because performing file system operations through the hyper-

visor involves hypercalls, which result in a context switch to the host kernel and a large performance overhead. Also, as host and guest have independent memory address spaces in hypervisor-based systems, file system I/O accompanies an explicit copy procedure.

In contrast, when hypervisor is not used, file I/O operations can be performed by mapping process address space directly to the host memory through `mmap` [16], which potentially leads to performance improvement. There is no additional cost for copying because file I/O is performed directly to the memory space mapped to the process that runs the Unikernel.

3.3. Approaches for Lightweitghting File System

Optimized file system design in Unikernel environments will be possible through characterization of file access patterns, I/O size, access frequency, etc. [17], but to the best of our knowledge, there are no specific characteristics known for file access characteristics in Unikernels.

As Unikernel is designed to execute a single application per kernel, the memory footprint of a single Unikernel is not expected to be large. Thus, the goal of our design is to maximize performance while minimizing resource usage. Specifically, performance can be improved by simplifying or eliminating unnecessary file system functions, such as permission management for multiple users or quota restrictions for each user.

This leads to a reduction in memory footprint, improving booting and execution performance. Moreover, if multiple hardware threads are not necessary for a Unikernel application, file systems can also assume a single hardware thread, and non-reentrant file systems with a single hardware thread enable a simplified and non-preemptive implementation [18]. Performance can be further accelerated by eliminating locking, which is a major cause of performance degradation in most file system implementations.

4. Implementations of ULFS

In this section, we describe the details of the proposed non-hypervisor-based Unikernel file system, which we call ULFS (Ultra Lightweight File System) [19]. Note that we design and implement ULFS based on our open-source project ULV (Ultra Lightweight Virtualization) [20]. ULV is designed as a non-hypervisor Unikernel, so it works directly on a host operating system without a hypervisor. Since Unikernels are intended to run a single application, we aim to implement ULFS under the assumption that the file I/O traffic in a Unikernel application is not heavy and file access patterns are simple (e.g., sequential). Based on this philosophy, ULFS is developed with a lightweight structure by removing features unrelated to a single process that performs primitive file I/O, such as access permission and state management. In particular, we attempted to improve performance largely by eliminating complicated lock structures throughout the file system that are not needed for single-thread applications.

ULFS uses a host's file as a guest's file system storage to perform file I/O. Since a file on the host can be mapped to the address space of a host process through `mmap`, file I/O can be performed by using memory reference interfaces. As the memory-mapped area uses the host's memory space in the form of a page cache, it can enhance file I/O operation latency. This is achieved by processing subsequent I/O requests directly from the cache, thereby eliminating the need for storage access. Also, ULFS improves file system performance since it is designed as a lock-free file system with a single-threaded and non-re-entrant style.

ULFS manages the entire file system space in blocks of 4 KB, and blocks can be identified with an integer block id (bid). The superblock, which is the leading block, has a bid value of 0 and contains basic information about the file system. Currently, it only maintains information about the maximum number of blocks. ULFS makes use of map blocks for managing block allocation of the file system in the form of a bitmap. As shown in Figure 2, the first map block is the block following the superblock and has bid number 1. From then on, map blocks are placed at regular intervals, as the maximum number of blocks managed by a single map block is fixed in our design. Thus, the locations of map

blocks can be quickly searched by tracking the number of unallocated blocks for each map block when searching for available blocks.

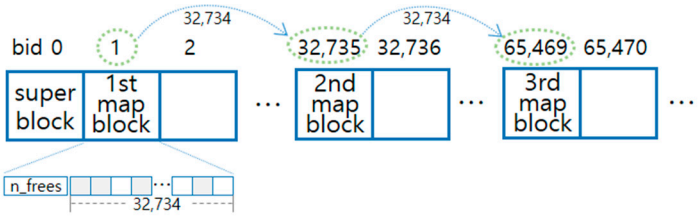

Figure 2. Block allocation based on map blocks in ULFS.

Files in ULFS are managed using traditional inode structures [21], and each inode contains the information of a file. The inode block is a block that manages a group of inodes in a tabular form, as shown in Figure 3. The bid number 2 is always assigned to the first inode block, and the maximum number of inodes managed by a single inode block is fixed to 170, as determined by the following equation: $\frac{inode\ block\ size - meta\ data\ size\ in\ inode\ block}{inode\ size} = \frac{4096-16}{24}$. If more inodes are necessary, a new inode block is allocated. As shown in Figure 3, inode blocks are managed as a linked list, and bid values for the preceding and following inode blocks are maintained.

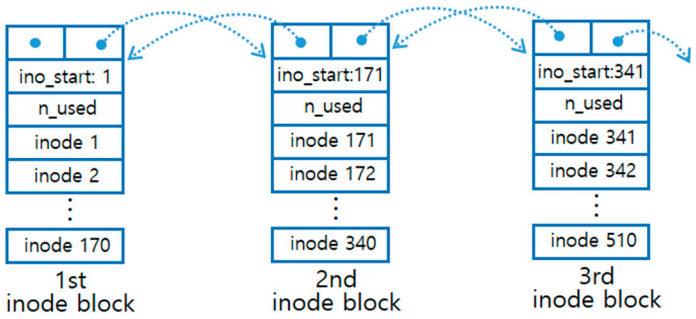

Figure 3. Inode block structures of ULFS.

File system APIs of the POSIX standard require an inode number to identify a file. In ULFS, inode numbers are sequentially assigned to inodes on a logical table consisting of inode blocks. Thus, inode information of a file can be easily identified by sequentially scanning inodes. However, as the inode block location (bid) and the inode offset within an inode block are internally maintained, quick access to inode information is possible. That is, the inode number can be calculated by maintaining the number of the first inode in the block (`ino_start` in Figure 3). Additionally, the inode block records the count of in-use inodes (`n_used` in Figure 3), in order to quickly check whether a new inode can be allocated.

Each inode maintains file type, size, and location of the block that makes up the actual data of the file as shown in Figure 4. For indicating the locations of file blocks, two bids are maintained in the inode; the first represents the head of data blocks, whereas the second points to a bid block that is an index block to maintain bids of subsequent data blocks. Actually, a bid block does not maintain all bid information of a file directly but is structured as indirect blocks to find the sweet spot between space and performance. As shown in Figure 4, the maximum number of blocks accommodated by a bid block is 1023. A data block corresponding to the logical offset of a file can be quickly searched based on this structure.

The file system hierarchy of ULFS is constructed by directories, which maintain file names and inode locations belonging to that directory as a fixed-length data structure. The data block of the root directory is fixed at bid number 2 and can be accessed quickly when referencing files by absolute path.

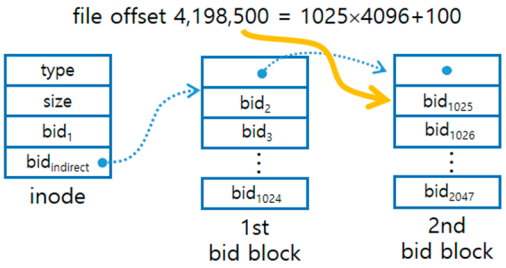

Figure 4. Example of finding a bid for the file offset (4,198,500) through bid blocks.

The proposed ULFS is similar to an early UNIX file system design [22], so it may be inefficient in complex file access patterns, but such a simple structure has merits in minimizing the operational demands on the file systems, especially for a single application workload. Note that we developed a lightweight file system from scratch, so it is differentiated from existing Unikernels that make use of the general-purpose kernel's file system as is, or simply slim down existing file systems.

Before concluding this section, we briefly discuss the trade-offs and limitations of the ULFS design from two perspectives. Firstly, the ULFS presented in this paper represents an optimized file system design tailored for scenarios where a single thread operates within each Unikernel, as illustrated in Figure 5a. Therefore, in instances where multiple threads are active within a Unikernel, as depicted in Figure 5b, ULFS may suffer from race conditions. This limitation arises because ULFS does not implement multi-thread locking, potentially leading to inconsistencies across different threads. Secondly, ULFS is engineered to minimize resource consumption while enhancing I/O performance. This is achieved by streamlining data structures to simplify the size of inodes or directory entries through fixed-length configurations. However, this approach might not be as effective against complex workloads. In particular, handling files larger than 4 KB requires the utilization of an indirect indexing block, which could increase latency in data access. Additionally, ULFS implements the POSIX APIs in a very simplified manner, and this simplicity could necessitate a wide range of application modifications or even make integration impossible for applications sensitive to specific file system semantics. For example, among the many fields returned by an fstat call, ULFS only sets st_ino, st_mode, and st_size, opting to default other fields that are typically unnecessary for most applications to ensure faster performance. However, applications sensitive to file ownership information might require considerable porting effort to integrate with ULFS. Thus, integrating with ULFS may pose challenges that require careful consideration for application compatibility.

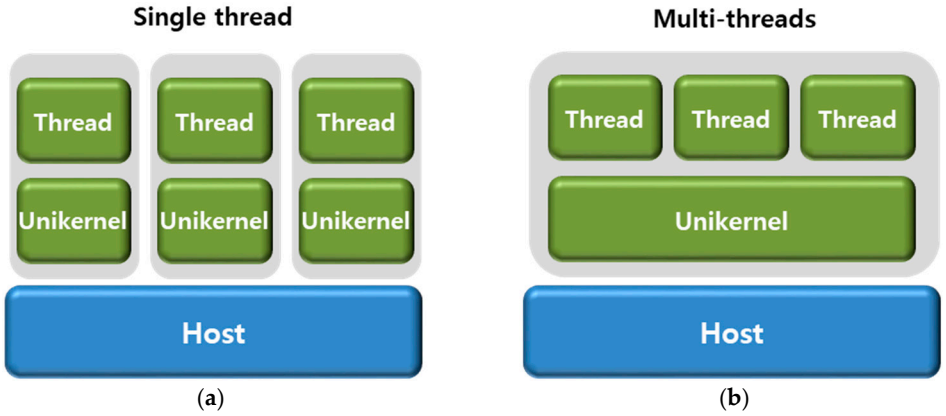

Figure 5. Execution of single thread and multi-threads in each Unikernel. (**a**) Single thread. (**b**) Multi-threads.

5. Experiment Results

In this section, we evaluate the file system performance of major Unikernels, including ULFS, through experimental runs. We compare the performance of ULFS with those of Rumpvfs used in the Nabla project, Ramfs and 9pfs used in Unikraft, and HermitCore fs (Hcfs) from HertmiTux. In the case of Ramfs, we consider two versions, Ramfs-u and Ramfs-q, to compare cases executed on Unikraft's Linuxu platform and Qemu platform, respectively. Note that Rumpvfs and Ramfs are in-memory file systems, whereas 9pfs and Hcfs are host-based file systems.

To evaluate the performance of a file system, it is crucial to measure how workload execution time varies in relation to access patterns and varying data volumes. In this paper, as we evaluate file system performance within a Unikernel environment, examining the memory usage of the host is also important. For performance measurement, we utilize both synthetic workloads, which simulate file access at a micro level, and real-world workloads, representative of macro-level usage. Figures 6–12 depict experiments utilizing synthetic workloads, whereas Figures 13 and 14 illustrate those conducted with real-world scenarios. Each experiment aims to explore the impact of varying data volumes on performance outcomes.

Our first experiment was conducted to compare the read-after-write performance of each file system implementation. The read-after-write is a common microbenchmark used to assess file I/O performance. It first splits a given set of storage data into chunks and then randomly determines the order of these chunks for I/O operations. Following this order, a read operation is conducted after a write operation for each chunk. Figure 6 shows the execution time of the read-after-write operations for the six file system implementations as the chunk size is varied. In this experiment, the total data size is set to 1MB, and the number of I/Os decreases as the chunk size increases, ensuring that the total I/O data remains consistent across all experiments.

As shown in the figure, ULFS provides the best performance for a wide range of chunk sizes. This is due to the simple structure of the ULFS design and the efficiency of file system I/O through `mmap` on the non-hypervisor structure. This performance gap implies that API and some other overhead rather than pure I/O processing account for a large portion of elapsed time in file systems other than ULFS.

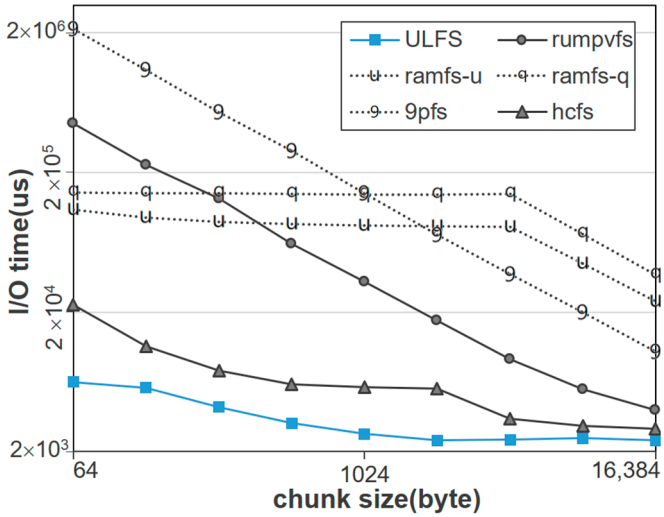

Figure 6. Performance comparison of Unikernel file systems in read-after-write.

As the chunk size increases, the execution time is improved in all file system cases, but the improvement varies depending on the details of the file system design. In the case of Ramfs, the execution time is almost the same until the chunk size is less than 4 KB and is improved significantly after that size. This is because memory allocation in units of 4 KB accounts for most of the execution time in Unikraft. Thus, the execution time decreases in

proportion to the chunk size after that size as the number of memory allocation requests decreases accordingly. Except for the two variants of Ramfs, the execution time decreases in proportion to the chunk size.

Figure 7 shows the execution time of read-after-write operations for the six file system implementations as the number of files is varied. Specifically, the file size is 1 MB as in previous experiments, but the number of files in this experiment is varied from 1 to 100, implying that the total I/O size changes from 1 MB to 100 MB. We set a sufficiently large chunk size of 4 MB in this experiment to minimize the overhead of API calls and see the exact I/O overhead of each file system. As shown in the figure, the execution time increases in proportion to the number of files, regardless of file systems. In our experiment, the performance of Ramfs is the worst, as it behaves inefficiently by copying memory data in byte units. In this experiment, the performance of Hcfs is similar to that of host I/O, and there is no significant difference between Hcfs and ULFS. This is because the chunk size of 4 KB is large enough, so there is almost no API overhead in this experiment.

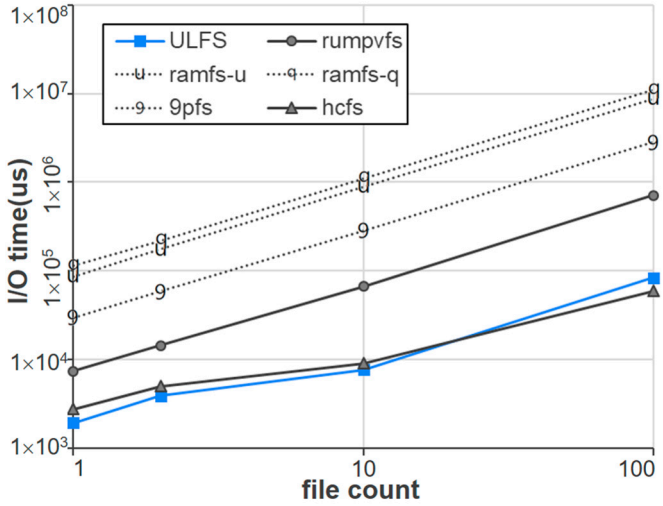

Figure 7. Performance comparison of Unikernel file systems as file counts are varied.

Figure 8 shows the memory usage of the six file system implementations in response to variations in the number of I/O files. In this experiment, all configurations, including the sizes and number of files, as well as chunk sizes, are set to identical as those in the experiment described in Figure 7. As shown in the figure, except for 9pfs and Hcfs, memory usage also increases as the number of I/O files increases. In the case of Hcfs, as the host memory is used for I/O processing, there is no additional memory consumed even when the number of files increases. 9pfs is also a host-based file system like Hcfs, but a certain additional memory space is internally used in the Unikernel during the mapping of host files.

Unlike general-purpose operating systems, Unikernels are not expected to support large-scale file systems, but the host may run a large number of Unikernel instances simultaneously. Thus, it is not desirable for a single Unikernel instance to use large memory space. In the results shown in Figure 8, ULFS uses almost 100MB of memory space when the number of files is 100, but we can simply limit the memory usage of a Unikernel by making use of `cgroups`. Figure 9 shows the execution time of ULFS when executing the same read-after-write workloads shown in Figure 8, with the memory usage of the Unikernel limited to less than 8MB. Even though the execution time increases due to memory constraints, ULFS performs reasonably well with small memory allocation. However, when the memory size is 1MB, the I/O performance is degraded significantly even for a single file. This is because the current implementation of ULFS utilizes host memory through `mmap` and uses LRU (Least Recently Used) as the memory page replacement algorithm. In a single application environment with workloads that involve simple sequential access, the

MRU (Most Recently Used) replacement algorithm is known to perform better than LRU. Therefore, in a Unikernel environment constrained by small memory, it would be desirable for ULFS to directly manage the page replacement policy.

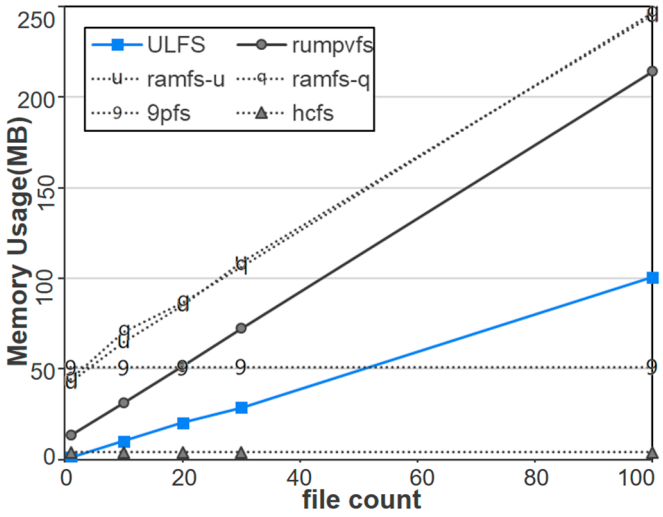

Figure 8. Memory usage as file counts are varied.

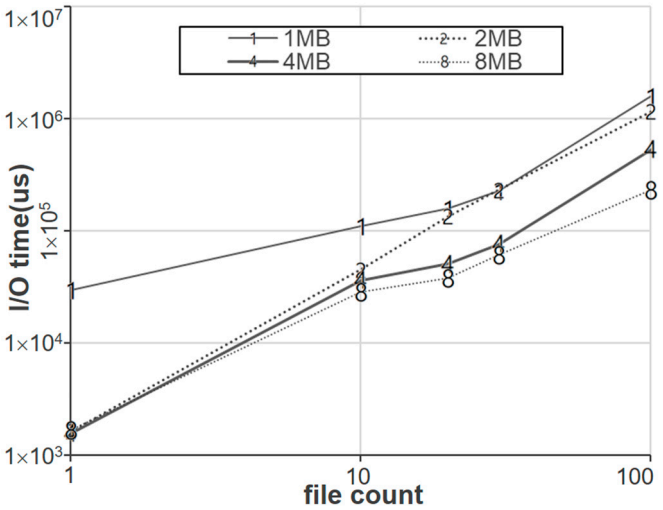

Figure 9. Execution time of ULFS as the memory size allocated to the Unikernel is varied.

Figure 10 shows the performance of the six file system implementations when directory entries are sequentially searched by the file systems. This experiment is conducted by configuring a specified number of files within a single directory and measuring the time taken to extract information from all file entries within that directory. To solely assess the API time, this process is carried out by simply referring to the inode numbers among the extracted information. We set the number of files within a directory to range from 10 to 1000 and invoke the `readdir` system call repeatedly. We perform this experiment 1000 times to eliminate the effect of average error in measurements. The results are presented with error bars showing standard deviations to illustrate the variability of the data. As shown in the figure, Rumpvfs and ULFS show reasonably good performance. Specifically, ULFS exhibits the most competitive performances as the number of directory entries increases. This is due to the simple and efficient implementation of ULFS for fast access to inode structures in directory items.

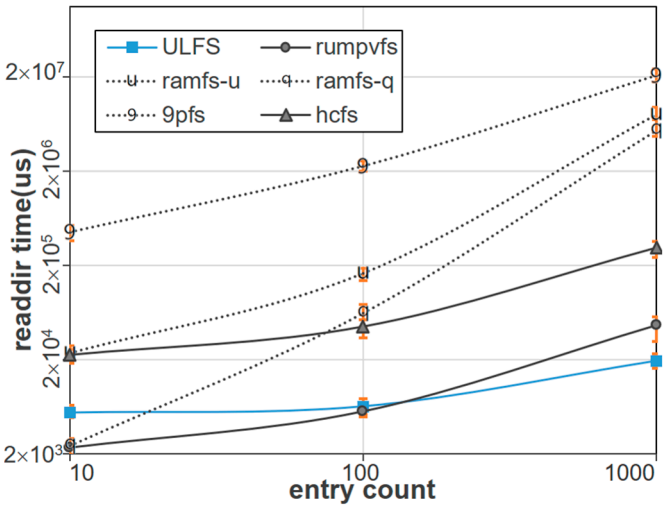

Figure 10. Performance comparison of Unikernel file systems in readdir.

Figure 11 compares the performance of the six file system implementations when `lseek` operations are repeatedly performed. In this experiment, an empty file is first created, and then the time to move the offset to the initial position of the file is measured while the number of lseek operations is varied from 100 to 1,000,000 times. To perform the lseek operation, we make use of the SEEK_SET option, which moves the file offset. As shown in the figure, Ramfs-q and 9pfs behave very similarly, and they show competitive performance along with ULFS. Although 9pfs is a host-based file system implementation, it performs file offset operations without contacting the host, showing excellent performances. In contrast, Hcfs forwards all `lseek` requests to the host, resulting in the worst performances.

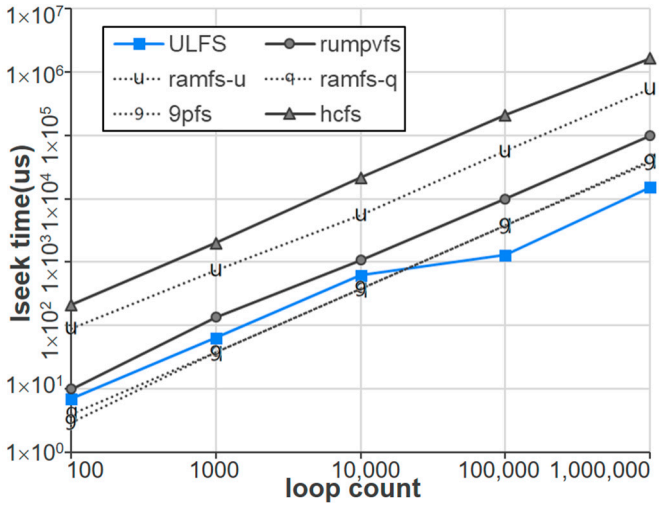

Figure 11. Performance comparison of Unikernel file systems in lseek.

Figure 12 illustrates the experimental results for another file system operation, `fstat`, for extracting file metadata. Specifically, the experiment measures the execution time of creating a single file and then invoking fstat repeatedly, from 100 to 1,000,000 times. To exclusively measure the API execution time, only the inode number is referenced from the fstat results, thereby minimizing the effect of executing the user code. The best performance is exhibited by ULFS, Rumpvfs, and Ramfs-q. However, it can be observed that as the number of iterations increases, the performance of ULFS, which is optimized for single-thread operations, becomes more pronounced. Specifically, we observe that ULFS performs better than other Unikernel file systems, particularly when the execution time exceeds around 1ms. We cannot quantify the exact reason for this, but it may be due to the ULFS's

lightweight design approach, which ensures that the working set fits within the CPU cache memory. Another potential factor contributing to the significant performance improvement of ULFS when the number of I/O requests exceeds a certain threshold is ULFS's utilization of the host's buffer cache. That is, a read-ahead function is triggered when the requested I/O size exceeds a certain threshold in ULFS, which fetches the subsequent block into the buffer cache before actually being requested, further enhancing performance.

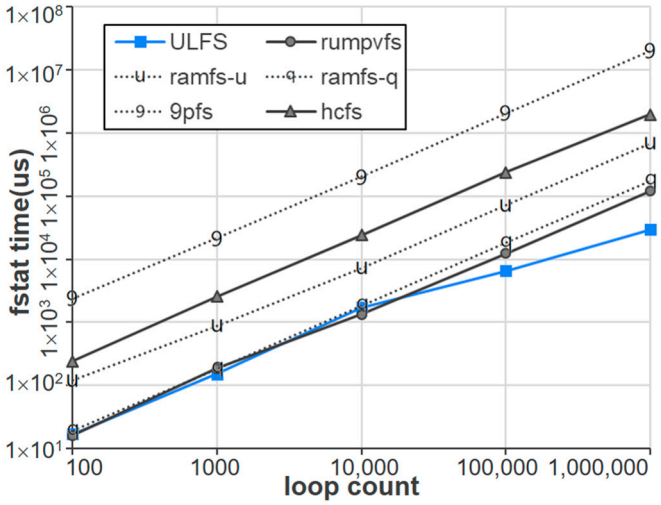

Figure 12. Performance comparison of Unikernel file systems in fstat.

Figure 13 compares the performance of the six file system implementations when the store and fetch operations for 4-byte data are repeatedly performed on the embedded database gdbm [23]. Specifically, the experiment sequentially inserts the designated data using 4-byte key values, followed by sequentially fetching the data using the same key values. Our gdbm benchmark measures the time taken to complete both insert and fetch operations as the number of items increases from 100 to 10,000. As shown in the figure, ULFS shows competitive performance in all cases, especially in heavy workloads where the number of operations exceeds 1000. When the workload is not heavy (i.e., the number of operations is less than 1000), in-memory file systems show better performance due to the caching effect inside gdbm. However, in-memory file systems have limitations in that they do not provide data persistence.

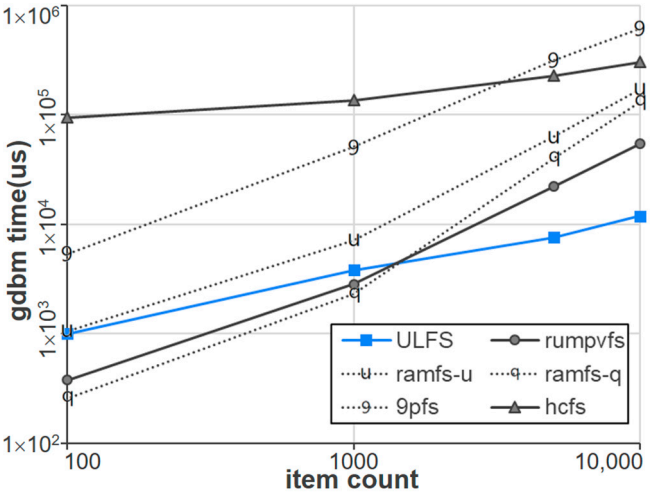

Figure 13. Performance comparison of Unikernel file systems in gdbm's store/fetch.

Figure 14 illustrates performance results for another practical storage engine, SQLite. Its journal mode is DELETE, known to incur substantial I/O overhead. In our experiment, each row added to the database follows a schema consisting of (id, string, float), and the string and float fields are filled with random values. The measured execution times are obtained by first inserting a designated number of items and subsequently selecting all inserted items. We measure the time taken to perform SQLite DELETE operations while varying the number of items from 1 to 1000. As shown in Figure 14, the results indicate a proportional increase in execution times across all file systems relative to item counts. While in-memory file systems (i.e., Rumpvfs and Ramfs) exhibit superior performance compared to host-based file systems (i.e., 9pfs and Hcfs), ULFS, benefitting from an optimized page cache mechanism and a simplified architectural design, facilitates the most effective execution times.

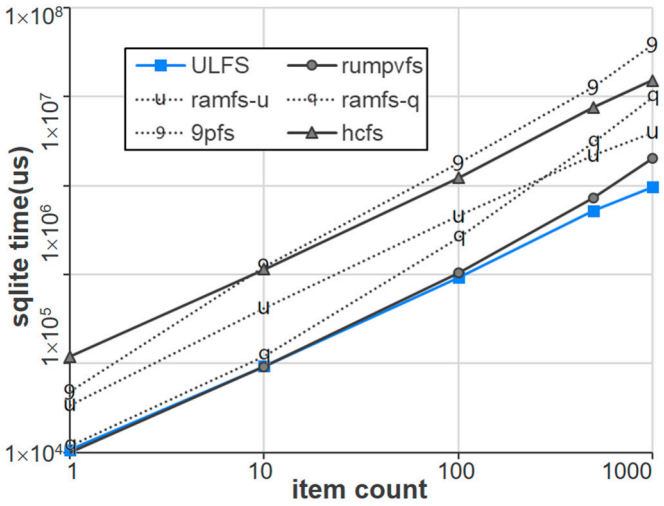

Figure 14. Performance comparison of Unikernel file systems for SQLite Insert/Select.

Before concluding this section, we briefly summarize the discussion of the performance results across different file system implementations by identifying the main reasons behind the performance differences observed.

- In most scenarios, ULFS delivers the best performance regardless of workload characteristics and I/O size, while other file systems give and take amongst each other under specific workload conditions.
- ULFS's superior performance can be attributed to its streamlined file system design and efficient file I/O operations via `mmap`. In contrast, other file systems suffer from significant overhead in software stacks (e.g., API calls) rather than real I/O processing.
- In all file systems, the latency required to access a given amount of data decreases as the I/O size increases. However, for in-memory file systems, which allocate memory in 4 KB units, this improvement becomes noticeable only for I/O sizes larger than 4 KB.
- In-memory filesystems (i.e., Ramfs-u, Ramfs-q, Rumpvfs) generally perform well, as they bypass storage access during file system operations. However, their performance significantly drops with frequent memory allocations, such as those required for small-sized read-after-write operations.
- Hcfs redirects API calls to the host for file system operations, leading to substantial overhead. Consequently, Hcfs performs well only when API calls are infrequent.
- 9pfs tends to show relatively lower performance since it is a host-based file system requiring a hypervisor, resulting in API call overhead. However, it excels in `lseek` operations where requests are managed directly by the guest and not transferred to the host, highlighting specific operational efficiencies.

6. Conclusions

Most existing Unikernel projects have focused on the compatibility issue of file system related system calls rather than performance optimizations. In particular, they tried to bind Unikernel file systems to host file systems or slightly modified in-memory file systems for customizing Unikernel environments. Thus, existing Unikernel file systems have limitations in performance, security, volatility, and/or resource efficiency. In this paper, we designed and implemented a new file system for Unikernels called ULFS, which supports system call services for Unikernel applications the same as general-purpose OS kernels and provides superior performance and security with minimal system resources. In particular, we developed ULFS as a lightweight file system based on the principle of Unikernel design, simplifying system calls, eliminating unnecessary locking and permission checks for multiple users, and applying a non-hypervisor structure. This leads to reducing the memory footprint of file systems and improving booting and execution performances. Measurement studies showed that ULFS outperforms various existing file systems for Unikernels, including Rumpvfs, Ramfs-u, Ramfs-q, 9pfs, and Hcfs.

In future research, we would like to improve the ULFS design through a precise analysis of file access patterns in Unikernel workloads and confirm the performance improvement based on the analysis results. In particular, deep learning workloads exhibit a repetitive access pattern across each training epoch. When these access patterns occur in parallel across multiple Unikernels within a single machine, it leads to inefficiencies from the perspective of the host. Consequently, the implementation of efficient caching and read-ahead techniques is crucial. We aim to investigate a collaborative approach between hosts and Unikernel file systems to ensure effective caching support for deep learning workloads. Specifically, by analyzing the access patterns characteristic of Unikernel workloads, ULFS could leverage the fadvise system call to communicate these patterns to the host system. This would enable the host's buffer cache to make more informed decisions about data caching, thereby potentially avoiding unnecessary data storage and improving overall data access efficiency. Such a collaborative caching strategy between the host system and Unikernel file systems could significantly enhance performance by optimizing resource utilization based on actual workload requirements. This approach not only aims to boost the efficiency of ULFS in handling specific workloads but also sets the stage for integrating advanced caching mechanisms tailored to the unique operational dynamics of Unikernels. ULFS is implemented with a single-threaded, non-hypervisor structure, and single address space, which enables stable file system execution through memory-mapped back-end storage that is free from page-fault overhead. By utilizing these design characteristics, we also plan to expand the application area of our file system to IoT environments that require real-time constraints.

Author Contributions: K.C. 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 supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2016R1A6A3A11930295) and in part by RP-Grant 2022 of Ewha Womans University.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in ULFS (Ultra Lightweight File System) at https://github.com/oslab-ewha/ULV/tree/master/libulfs (accessed on 11 April 2024), reference number [19].

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

References

1. Madhavapeddy, A.; Mortier, R.; Rotsos, C.; Scott, D.; Singh, B.; Gazagnaire, T.; Smith, S.; Hand, S.; Crowcroft, J. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Comput. Archit. News* **2013**, *41*, 461–472. [CrossRef]
2. Ioini, N.; Majjodi, A.; Hastbacka, D.; Cerny, T.; Taibi, D. Unikernels Motivations, Benefits and Issues: A Multivocal Literature Review. In Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum (ESAAM), Ludwigsburg, Germany, 17 October 2023; pp. 39–48.
3. Kurek, T. Unikernel network functions: A journey beyond the containers. *IEEE Commun. Mag.* **2019**, *52*, 15–19. [CrossRef]
4. Fingler, H.; Akshintala, A.; Rossbach, C.J. USETL: Unikernels for serverless extract transform and load why should you settle for less? In Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSYS), Hangzhou, China, 19–20 August 2019.
5. Choudhuri, S. A Case for Unikernels in IoT: Enhancing Security and Performance. In *Internet of Things: Enabling Technologies, Security and Social Implications*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 85–91.
6. Yoo, S.; Jo, Y.; Bahn, H. Integrated scheduling of real-time and interactive tasks for configurable industrial systems. *IEEE Trans. Ind. Inform.* **2022**, *18*, 631–641. [CrossRef]
7. Chen, S.; Zhou, M. Evolving container to Unikernel for edge computing and applications in process industry. *Processes* **2021**, *9*, 351. [CrossRef]
8. Park, S.; Bahn, H. Trace-based Performance Analysis for Deep Learning in Edge Container Environments. In Proceedings of the 2023 Eighth International Conference on Fog and Mobile Edge Computing (FMEC), Tartu, Estonia, 18–20 September 2023; pp. 87–92.
9. Williams, D.; Koller, R.; Lucina, M.; Prakash, N. Unikernels as processes. In Proceedings of the ACM Symposium on Cloud Computing, Carlsbad, CA, USA, 11–13 October 2018; pp. 199–211.
10. Kantee, A.; Cormack, J. Rump kernels: No OS? no problems! *Login Mag. USENIX SAGE* **2014**, *39*, 11–17.
11. Kantee, A. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. Ph.D Thesis, Aalto University, Aalto, Finland, 2012.
12. Kuenzer, S.; Bădoiu, V.; Lefeuvre, H.; Santhanam, S.; Jung, A.; Gain, G.; Soldani, C.; Lupu, C.; Teodorescu, Ş.; Răducanu, C.; et al. Unikraft: Fast, specialized Unikernels the easy way. In Proceedings of the 16th ACM European Conference on Computer Systems (Eurosys), Edinburgh, UK, 26–28 April 2021; pp. 376–394.
13. 9pfs Setup Document. Available online: https://wiki.qemu.org/Documentation/9psetup (accessed on 11 April 2024).
14. Olivier, P.; Chiba, D.; Lankes, S.; Min, C.; Ravindran, B. A binary-compatible Unikernel. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Providence, RI, USA, 14 April 2019; pp. 59–73.
15. Edge, J. A Seccomp Overview. Linux Weekly News. 2015. Available online: https://lwn.net/Articles/656307/ (accessed on 11 April 2024).
16. Mmap System Call. Available online: https://man7.org/linux/man-pages/man2/mmap.2.html (accessed on 11 April 2024).
17. Lee, J.; Bahn, H. File Access Characteristics of Deep Learning Workloads and Cache-Friendly Data Management. In Proceedings of the 2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), Palembang, Indonesia, 20–21 September 2023; pp. 328–331.
18. Agreand, J.; Tripathi, S. Modeling reentrant and nonreentrant software. *ACM SIGMETRICS Perform. Eval. Rev.* **1982**, *11*, 163–178.
19. Ultra Lightweight File System (ULFS). Available online: https://github.com/oslab-ewha/ULV/tree/master/libulfs (accessed on 11 April 2024).
20. Ultra Lightweight Virtualization (ULV). Available online: https://github.com/oslab-ewha/ULV (accessed on 11 April 2024).
21. Mathur, A.; Cao, M.; Bhattacharya, S.; Dilger, A.; Tomas, A.; Vivier, L. The new ext4 filesystem: Current status and future plans. *Proc. Linux Symp.* **2007**, *2*, 21–33.
22. McKusick, M.; Joy, W.; Leffler, S.; Fabry, R. A fast file system for UNIX. *ACM Trans. Comput. Syst. (TOCS)* **1984**, *2*, 181–197. [CrossRef]
23. GDBM. Available online: https://www.gnu.org.ua/software/gdbm/ (accessed on 11 April 2024).

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.