Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

NICFS: a file system based on persistent memory and SmartNIC*

Yitian YANG, Youyou LU‡

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

E-mail: yangyiti22@mails.tsinghua.edu.cn; luyouyou@tsinghua.edu.cn

Received Oct. 15, 2022; Revision accepted Feb. 22, 2023; Crosschecked Apr. 26, 2023

Abstract: Emergence of new hardware, including persistent memory and smart network interface card (SmartNIC), has brought new opportunities to file system design. In this paper, we design and implement a new file system named NICFS based on persistent memory and SmartNIC. We divide the file system into two parts: the front end and the back end. In the front end, data writes are appended to the persistent memory in a log-structured way, leveraging the fast persistence advantage of persistent memory. In the back end, the data in logs are fetched, processed, and patched to files in the background, leveraging the processing capacity of SmartNIC. Evaluation results show that NICFS outperforms Ext4 by about 21%/10% and about 19%/50% on large and small reads/writes, respectively.

Key words: Non-volatile memory; Persistent memory; Data processing unit; Smart network interface card (SmartNIC); File system
https://doi.org/10.1631/FITEE.2200469 **CLC number:** TP212

1 Introduction

In recent years, the emergence of new hardware, including persistent memory and smart network interface card (SmartNIC), has opened up new possibilities to file system design. Persistent memory, similar to phase change memory (PCM), resistive random access memory (RRAM), magnetic RAM (MRAM), and so on, provides data persistence at performance close to that of dynamic RAM (DRAM) and enables byte addressability. SmartNIC is a new class of network hardware equipped with a processing capacity that supports user programming and is helpful for file processing.

Recent research has intensively studied the persistent memory file systems, including both local file systems (Condit et al., 2009; Ou et al., 2016; Xu and Swanson, 2016; Chen et al., 2021) and distributed file systems (Lu et al., 2017; Anderson et al., 2019). Similarly, SmartNIC has been studied in a number of file systems (Schuh et al., 2021; Li et al., 2022). As far as we know, there is only one work that uses both persistent memory and SmartNIC in the file system, LineFS (Kim et al., 2021). In comparison to LineFS, which is a distributed system design, the NICFS proposed in this paper aims to study the impact of the new hardware on the local file system, without the interference of network inputs/outputs (I/Os).

In commercial computing clusters such as cloud data centers and high-performance computing centers, the economics of central processing units (CPUs) are very important since high-performance CPUs are sold and directly billed. We tend to expect high-performance CPUs (1) to waste as little computing power as possible by not doing other things and (2) to maximize revenue by completing only the user's computational tasks. For example, for storage tasks, the execution time is tied to the slow storage

‡ Corresponding author
* Project supported by the National Key R&D Program of China (No. 2021YFB0300500) and the National Natural Science Foundation of China (No. 62022051)
ORCID: Youyou LU, https://orcid.org/0000-0002-6214-5390

devices, and the fast CPUs will be idle due to polling and waiting for the slow devices, resulting in a loss of computing resources. Therefore, we considered introducing some lower-performance computing devices to handle these tasks instead of host CPUs, and SmartNIC caught our attention. SmartNIC was originally designed to offload host CPU tasks, which is in line with our thinking, so we tried to use SmartNIC to offload some of the work of maintaining the file system on the host CPU.

On the other hand, the high performance and byte-addressable nature of persistent memory make it ideal for storing data, such as logs and metadata. Therefore, we are also interested in exploring the enhancements that persistent memory can bring to the file system.

With the above considerations in mind, we design and implement a local file system, NICFS, to leverage persistent memory and SmartNIC. NICFS contains two modules, NICFS-Frontend and NICFS-Backend. NICFS-Frontend is located on the host and absorbs file system updates to the persistent memory in a log-structured way. Since persistent memory provides high-performance persistence and byte addressability, file system updates can be persisted quickly. The log-structured way also improves the performance, since all updates are appended in the log space of the persistent memory in a sequential write pattern. NICFS-Backend is located on SmartNIC and is responsible for merging the updates from NICFS-Frontend to the files in the data space in the background. It uses the computing capacity and resources of SmartNIC to offload the log-processing task of NICFS-Frontend, thus releasing the computing resources of host CPUs. Moreover, the log-processing tasks are performed in the background, which removes the data allocation and metadata processing tasks from the critical path of file writes and thus further improves performance.

We summarize our major contributions as follows:

1. We analyze the features of persistent memory and SmartNIC and design a local storage architecture in detail by combining the two new devices as the centerpiece with the classical storage system design.

2. We design a series of data structures and algorithms for logging, caching, merging, processing, and parallelizing to be friendly to persistent memory and SmartNIC and to maximize the performance of both.

3. We provide an implementation of the proposed storage architecture and a comprehensive evaluation on a machine that meets the conditions. The results are compared with those of the prototype to verify the improvement in read/write performance brought about by persistent memory and SmartNIC of the architecture.

2 Background

In this section, we present the background of persistent memory and SmartNIC, as well as the related works.

2.1 Persistent memory

The emergence of persistent memory has broken the long-standing structure of the storage pyramid. Situated between DRAM and solid-state drive (SSD), persistent memory has the characteristics of a memory device (byte addressability and high performance) while also having the features of a persistent device (high capacity and persistence). In all aspects of performance, cost, power consumption, and capacity, persistent memory fills the gap between the upper and lower levels in the storage pyramid. In the design of the storage architecture, persistent memory is poised to bring a huge change. More and more systems are now trying to leverage persistent memory (Condit et al., 2009; Ou et al., 2016; Xu and Swanson, 2016; Lu et al., 2017; Anderson et al., 2019; Liang et al., 2020; Chen et al., 2021); in this way, more data can be persisted close to host CPUs, thus speeding up the system start-up, reducing I/O latency, and improving storage performance. Persistent memory can also be used for efficient and particularly fine-grained data access due to its byte-addressable nature.

2.2 SmartNIC

SmartNIC, which is also known as the data processing unit (DPU), is a network interface card (NIC) with programming and computing capabilities; it was originally designed to offload general-purpose processing tasks from the CPU. Tasks such as encrypting and decrypting network traffic, checking firewalls, and routing—which are CPU-intensive,

transparent to upper-layer applications, and closely related to the network layer—are more straightforward and amenable to efficient execution by the NIC than by the host CPU. It is also a promising topic to use SmartNIC to effectively offload storage services, reduce the burden of the host CPU in providing storage services, and accelerate the processing of these tasks to improve storage system performance.

2.3 Related works

File systems have been studied for many years, from hard-disk drive file systems, flash file systems (Lu et al., 2013, 2014; Lee et al., 2015; Zhang et al., 2016; Chen et al., 2021), to persistent memory file systems (Condit et al., 2009; Ou et al., 2016; Xu and Swanson, 2016; Lu et al., 2017; Anderson et al., 2019; Chen et al., 2021). SmartNIC has also been studied in a number of computer systems (Schuh et al., 2021; Li et al., 2022). Among these, we describe two works that are mostly related to NICFS, i.e., StageFS and LineFS.

StageFS (Lu et al., 2019) divides a file system into two phases: staging phase and patching phase. In the staging phase, the synchronous inputs/outputs (I/Os) are appended to the log in a log-structured way. In the patching phase, these updated data are merged into the files in place. NICFS shares a similar design with StageFS, regarding the separated phases in a file system. Differently, StageFS aims to absorb the synchronous I/Os in flash file systems and is designed for open-channel SSDs (Lu et al., 2013). NICFS aims to mitigate the file system overhead in the critical path and uses persistent memory and SmartNIC.

A recent file system, LineFS (Kim et al., 2021), follows a similar idea to use different phases in a file system. It introduces SmartNIC into the system to actively assist the host CPU with storage tasks. LineFS offloads data publishing, replication, and consistency coordination in Assise from the host CPUs to SmartNIC and accelerates the execution in a pipelined manner. It effectively reduces CPU usage, significantly decreases the latency of storage operations, and greatly improves I/O throughput. Moreover, in this way, even if any host CPU is crashed or blocked, SmartNIC can still complete its own work and communicate with other storage nodes; thus, the fault-handling capability of the system is also improved. NICFS shares a similar architecture with LineFS, but the former focuses on local file system design and uses different designs in data structure and layout.

3 Design and implementation

In this section, we present the design and implementation of NICFS. NICFS aims to explore the benefits of the two new hardware systems, namely, persistent memory and SmartNIC. The key idea is to provide (1) fast persistence of the file system by exploiting the characteristics of persistence and high throughput of persistent memory and (2) low processing overhead by leveraging SmartNIC for background processing.

Fig. 1 shows the overall architecture of the NICFS file system. NICFS consists of two modules, namely, NICFS-Frontend on the host and NICFS-Backend on SmartNIC.

NICFS-Frontend is the top half of the file system and is the interface at which the file system exposes to processes, i.e., user applications. NICFS-Frontend appends the log updates to the file system performed by the process into the private log space in persistent memory and reads these logs when necessary. The log spaces of different processes are isolated and allocated from a preconfigured section on the persistent memory. The remainder of the persistent memory is the public data space, which is the normal file storage space. In NICFS, we borrow the file layout from the Ext2 file system.

NICFS-Backend is the bottom half of the file system and is a daemon running on SmartNIC, which is responsible for processing logs and persisting updates to the file storage space. After NICFS-Backend receives a request from NICFS-Frontend, with the communication tool of data center infrastructure-on-a-chip architecture (DOCA) (NVIDIA, 2022), it fetches the logs from persistent memory to the memory on SmartNIC through DOCA direct memory access (DMA). It then processes these logs, locates the original locations of the log data blocks, merges these logs, and finally patches them to the files in the public data space (only for index node (inode) and index block updates). After that, it notifies NICFS-Frontend to start I/O acceleration technology (IOAT) DMA to publish the data of the other part of the logs (data block updates) by copying them from the log section to the file section on persistent

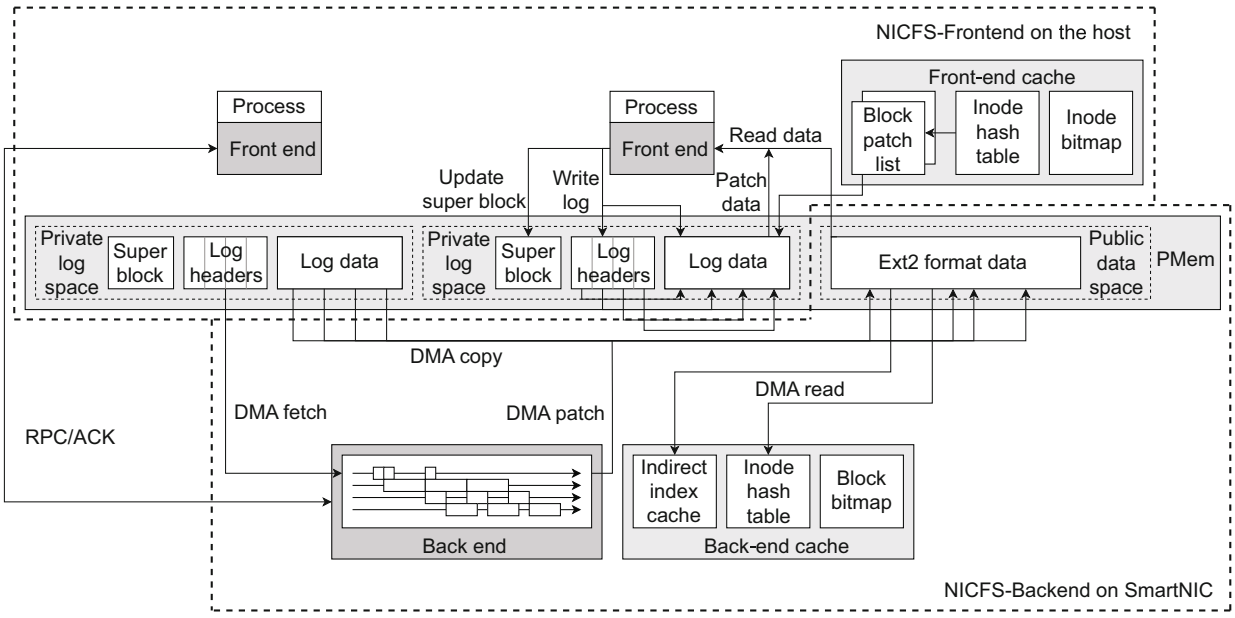

Fig. 1 The overall architecture of NICFS (ACK, acknowledgement; DMA, direct memory access; PMem, persistent memory; RPC, remote procedure call; SmartNIC, smart network interface card)

memory.

The following subsections introduce the design of the architecture in detail from two aspects, namely, NICFS-Frontend and NICFS-Backend.

3.1 NICFS-Frontend

The top part of Fig. 1 shows the architecture of NICFS-Frontend. We describe the different parts of NICFS-Frontend, including the interface between processes and the file system, the private log space, and the front-end cache.

3.1.1 Interface

NICFS-Frontend is implemented in the userspace. Different from the legacy file system designs that are implemented in the kernel space using the virtual file system (VFS) layer, as shown in the left half of Fig. 2, NICFS-Frontend directly accesses the persistent memory from the userspace using the direct access (DAX) mode. The problem with userspace implementation is how to intercept the system calls of user applications (or processes). In NICFS-Frontend, we follow the implementation of our previous work (Lu et al., 2017; Chen et al., 2021) to leverage the system call intercept library. All system calls made by the application are intercepted and forwarded to NICFS-Frontend in the user mode. As shown in the figure, this approach completely bypasses the kernel and provides better performance.

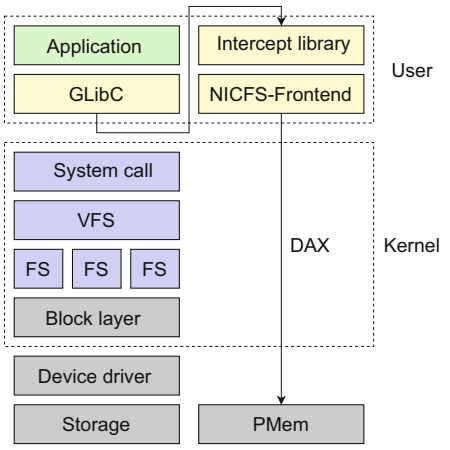

Fig. 2 Intercepting the Linux file system read/write call stack from user mode (DAX, direct access; FS, file system; GLibC, GNU C library; PMem, persistent memory; VFS, virtual file system)

3.1.2 Private log space

The private log space in NICFS-Frontend is the persistent storage for all incoming requests. NICFS-Frontend appends the data updates from the requests to the private log space in a log-structured

way to provide better performance. The performance benefits arise from two aspects. First, the persistent memory provides fast data persistence in a byte-addressable way. Compared to block-aligned updates, the overhead is reduced. Second, the updates are performed in a log-structured way. The log-structured way mitigates the overhead of reading file metadata and locating the data, in addition to transforming the random write into sequential write. The log-structured way also improves performance.

NICFS reserves a section on persistent memory to allocate the dedicated log space for each process, so as to reduce contention. Each private log space contains a super block, some log headers, and some log data. Different from the traditional log structure shown in Fig. 3a, we reorganize the log structure as shown in Fig. 3b for NICFS-Frontend.

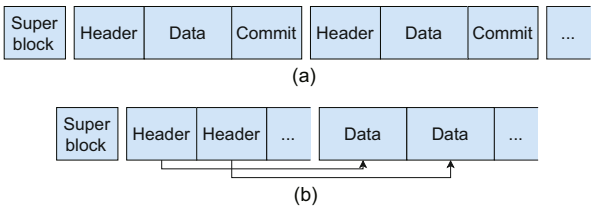

Fig. 3 Two different ways to organize logs: (a) traditional log structure; (b) NICFS-Frontend log structure

The goal of this log structure reorganization is to improve the efficiency of log processing in NICFS-Backend. The log data written by NICFS-Frontend are fetched, processed, and patched to the public data space (e.g., the Ext2 format part in the current NICFS implementation) on persistent memory by NICFS-Backend. The log header is necessary for NICFS-Backend because it contains meta-information, such as inode id, offset, and length of a write operation, which is needed for processing the log. However, the log data do not serve any purpose other than to be copied when performing the patch operation. Besides, if log data are fetched to SmartNIC, they must go through the slow peripheral component interconnect express (PCIe) bus, which is time-consuming. Therefore, for better performance, NICFS-Backend fetches only headers but not data when fetching logs. When log data need to be copied during the patch step, NICFS-Frontend is notified to do a DMA copy. In this way, NICFS-Backend eliminates reliance on log data. To support this strategy, NICFS-Frontend stores log headers separately from log data, which facilitates the fetching of the headers continuously by NICFS-Backend. The commit part of logs loses its role, to keep the atomicity of logs, and NICFS-Frontend atomically updates the head and tail pointers of the log cycle queue in the log super block.

3.1.3 Front-end cache

In NICFS, write operations are staged in the private log space and are not immediately applied to files in the public data space. One problem is how to provide consistent data reads, as some parts might be in the private log space, while others may be in the public data space. In NICFS-Frontend, we use the front-end cache to address the consistent read problem.

Specifically, there are two main types of consistency problems. These two types can be summarized from the perspective that the application will request two main pieces of information from the file system: inode and file content. Inode inconsistency can be solved by maintaining an inode cache using a hash table. Only a few maintenance times need to be introduced for each time of logging. As for file content inconsistency, it is not practical to solve it only by caching data blocks with a hash table. Because the log data can reach gigabytes, caching that much data will require a lot of memory. Thus, instead of caching entire blocks, NICFS-Frontend uses an update list for each block to track all the log data that cover it (Fig. 4).

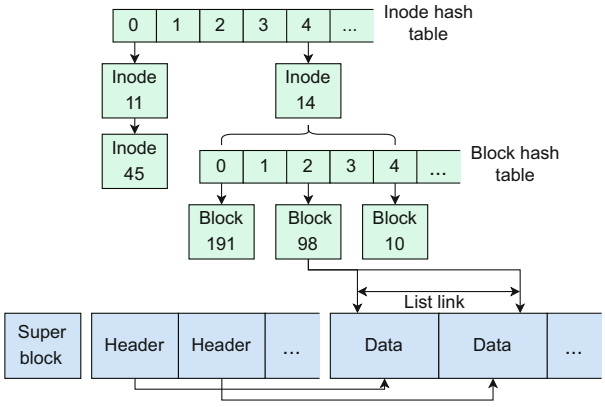

Fig. 4 Structure of file block cache

When NICFS-Frontend reads a file, it first reads data that have not been updated from the Ext2

section, and then iterates through the update list corresponding to the file blocks, overwrites the data read with newer log data to which the list items point, and finally obtains the newest data. The disadvantage of this method is that the performance is poor when there are too many list items, for example, when NICFS-Frontend repeatedly performs fine-grained writes to the same data block. In this case, page cache can be used in conjunction with the update list, and the two can complement each other to achieve high performance.

3.2 NICFS-Backend

The bottom part of Fig. 1 shows the architecture of NICFS-Backend. NICFS-Backend is located on SmartNIC and is not able to do inter-process communication with NICFS-Frontend, nor can it access persistent memory directly. We use NVIDIA MBF2H516A-EENOT BlueField-2 DPU in the current NICFS-Backend implementation. With this SmartNIC, DOCA offers a high-speed data exchange between SmartNIC and the host, allowing NICFS-Backend to execute its functions. NICFS-Backend uses (1) DOCA Comm Channel for small transfers (such as remote procedure call (RPC)) with NICFS-Frontend and (2) DOCA DMA for large copies and persistent memory accesses.

3.2.1 Patch pipeline

NICFS-Backend reads the logs from the private log space and patches these updates to the public data space. This patch function consists of four main steps: (1) Log Fetch, (2) Data Locate, (3) Merge, and (4) Patch. In the Log Fetch step, NICFS-Frontend fetches the log headers from the private log space from different logs that are appended by NICFS-Frontend. In the Data Locate step, NICFS-Frontend reads the log headers to query the block location in each file of each updated data. In the Merge step, NICFS-Frontend tries to merge the file system updates in several logs, so as to merge the overlapped parts and reduce the data-moving overhead. In the Patch step, NICFS-Frontend patches the updated data blocks to the files in the public data space.

The challenge in NICFS-Backend is that the processing capacity in SmartNIC is weaker than that in the host because SmartNIC usually is equipped with weak cores. The complex execution logic of NICFS-Backend and the weak core performance of SmartNIC can easily lead to poor log-processing performance of NICFS-Backend. To address this issue, there are two possible ways. One is parallel computing using multicores. Processing logs in a data-parallel manner is not a promising approach, as ensuring the sequential nature of the logs will become problematic. The other way out is the pipeline way. We can pipeline the four steps, as shown in Fig. 5.

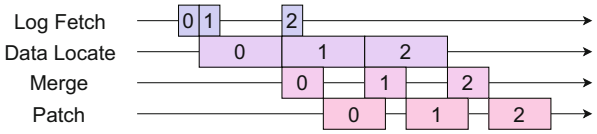

Fig. 5 Execution pipeline of NICFS-Backend

This approach can effectively improve the performance without disturbing the order of logs by covering up the execution time of other steps with the slowest step of the pipeline.

3.2.2 Back-end cache

In the patch pipeline, we find that the Data Locate step is the slowest among the four steps. Therefore, we aim to improve the performance of the Data Locate step, so as to improve the overall pipeline performance. In the Data Locate step, NICFS-Backend queries the block indices of files, causing the Data Locate step to repeatedly fetch the index block data from persistent memory. Since SmartNIC has to go through the PCIe bus to access the persistent memory on the host, the data transfer is less efficient. From this point of view, it is necessary to cache the index blocks locally on SmartNIC to reduce the cross-PCIe requests in the Data Locate step; this is what we call the back-end cache.

When updating a file, the address space to be updated should be continuous; this means that the index items in the index blocks to be queried during the Data Locate step are also likely to be continuous. Caching for continuous data can be done in the form of cache lines, which is what we do for back-end caching. The size of a back-end cache line is set according to the write granularity and page size. For instance, in a scenario wherein the granularity of write is generally $<$4 KB and the page size is 1 KB, setting the size of a cache line to four index items would be a reasonable choice.

3.2.3 Log merging algorithm

The Patch step is also time-consuming. One way to reduce the time spent on this step is to reduce the data to be patched. It is difficult to achieve this with a single log, but if multiple logs are processed at once, the data to be patched can be reduced by merging the logs. An effective algorithm for log merging is needed to merge logs as adequately as possible.

In general, logs make three types of modifications to the file system, namely, modifications to inodes, index blocks, and data blocks, and only the same type of modification has the opportunity to be merged.

Inode: The most easily merged type of modification is that to inodes. The system simply maintains a cache for the inodes being modified, updates this cache, and subsequently writes the cache to the persistent memory via DOCA DMA during the Patch step.

Index block: Another type is the modification of index blocks. When writing data to a location that exceeds the file size, new blocks need to be allocated to the file; at this point, new index items will be added to index blocks. At the same time, new holes may be generated in the file; then, it is necessary to empty the index items of the corresponding ranges. So, there are two types of operations: single-point modification and range clearing. NICFS-Backend uses the algorithm displayed in Fig. 6 to merge these operations.

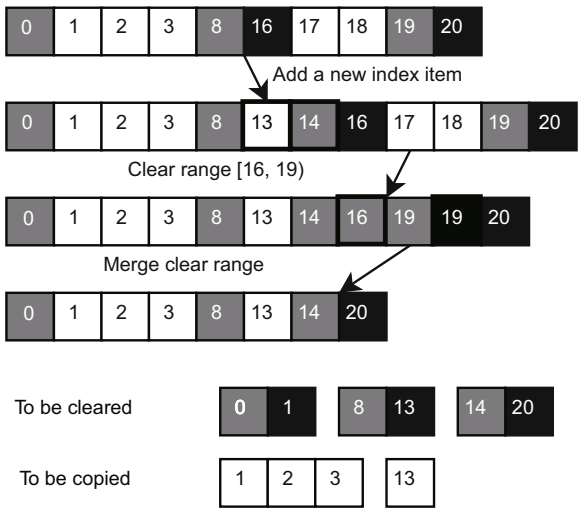

Fig. 6 Merge algorithm for modifications to an index block

The system creates an Adelson-Velsky and Landis (AVL) tree for each index block that is being modified. This AVL tree uses the in-block number as the key. The row of squares in Fig. 6 represents all nodes of the AVL tree, i.e., the modifications on an index block. The number in a square represents the key of the node. White squares stand for newly added index items. Gray and black squares indicate the left and right endpoints of ranges that need to be set to idle (called clear range), respectively; these nodes are specially marked in the AVL tree.

As shown in Fig. 6, when adding an index item, i.e., assigning a new block to the file, a white square is inserted into the AVL tree. If a gray square is to the left of this white square, it means that the white square divides a clear range into two, and a gray square should be inserted to the right of it. When doing range clear, i.e., punching the file, the AVL tree will delete a subtree and then try to add a black square first. When there is a gray square to the left of the black square, it means that the right side of the black square is also a clear range; thus, the addition of the black square can be dropped, which means merging the clear range to be added with the clear range to the right. Subsequently, after adding a gray square, the algorithm checks if the right side of some gray squares is a gray square, and if it is, then this gray square is removed for reasons similar to those when adding a black square.

Fig. 6 presents a direct index block of a file, meaning that its lower levels are pages. Assume that each page is 1 KB; thus, this index block manages the range [0, 20) KB of the file. At first, the file only holds data in the ranges [0, 4), [8, 16), and [17, 20) KB, and the data in the ranges [0, 1), [8, 16), and [19, 20) KB are going to be cleared. We then write data to the range [13, 14) KB and discard the data in the range [16, 19) KB. Obviously, in the end, only the ranges [1, 4) and [13, 14) KB have data, while the ranges [0, 1), [8, 13), and [14, 20) KB need to be cleared, which is consistent with what is shown in Fig. 6.

Finally, in the Patch step, the algorithm scans the squares from left to right to obtain a series of clear or copy ranges, which are later handed over to DMA for execution.

Data block: The last type of modification is writing to data blocks of files. This type of modification is similar to the modification to index blocks;

the difference is that this type applies only to ranges including range write and range clear. The specific algorithm to merge this type of modification is presented in Fig. 7.

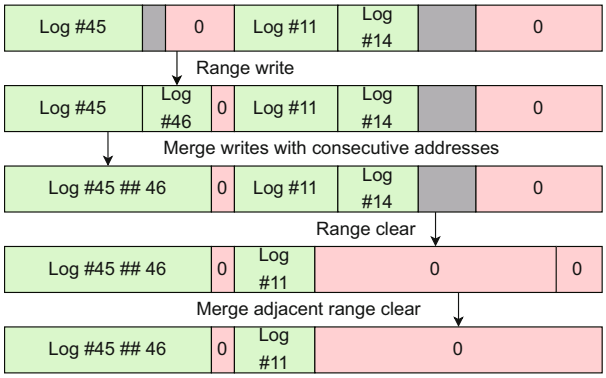

Fig. 7 Merge algorithm for modifications to a data block

For each block, we use an AVL tree to maintain the ranges to be modified. Because, in the end, there will not be two different pieces of data to be written to the same address, and the AVL tree stores multiple, nonoverlapping ranges; thus, the left endpoint of a range can be taken as the key. A row in Fig. 7 represents an element in the AVL tree, i.e., the modifications on a page. When adding a range, all overwritten ranges are removed, and the ranges on the left and right sides of the overlap are truncated and merged with the newly added range when it is possible to do so. In the Patch step, these modifications will be applied by DMA.

Fig. 7 presents a page in a file. This page is written by log 45, log 11, and log 14, separately, and some data are erased. Then, log 46 also writes to this page, right after log 45. Since log 46 and log 45 are contiguous, their log data are also contiguous in memory, which allows us to combine two contiguous writes of log 45 and log 46 into one contiguous write. Later, we need to delete some more data for some operations; we delete the overwritten logs, such as log 14 and the back part of log 11. Since there are consecutive purge operations, we will combine them into one continuous purge. This is easy to do with the AVL tree.

3.3 Concurrency

In NICFS-Frontend, each process (or user application) has its own logs. Logs are not shared among different processes for performance reasons. When multiple processes read and write the same file, each process can read or write part of the file data. The data are not consistent among different processes, and this is called the concurrency issue.

To address this issue, we use the lease semantic (Gray and Cheriton, 1989) to prohibit different processes from reading and writing to the same file at the same time. A lease can be thought of as a time-limited file read/write access given to a process, and it has semantics similar to a read/write lock. It is a reasonable choice for NICFS-Backend to manage leases. On one hand, it can reuse the RPC connections between NICFS-Backend and NICFS-Frontends. On the other hand, the reclamation of leases needs the cooperation of NICFS-Backend. NICFS-Frontend needs to check its own lease every time it reads/writes a file; if the lease has expired or is not available, it needs to contact NICFS-Backend. When the lease is reclaimed, NICFS-Backend has to make sure that all the logs produced by NICFS-Frontend of the file have been synchronized to Ext2, so that the next read of the file can be consistent.

4 Evaluation

To evaluate the design, experiments were conducted based on the following questions:

1. How is the read/write performance of NICFS? How do the persistent memory and SmartNIC bring improvements to it? (Sections 4.2.1 and 4.2.2)

2. What are the strengths and weaknesses of NICFS? How scalable is it? (Section 4.2.3)

3. Does NICFS-Backend have high enough processing power not to affect the efficiency of NICFS-Frontend? How scalable is it? (Sections 4.3.1 and 4.3.2)

4. Is the design of cache and pipeline necessary? How much improvement can be brought to NICFS-Backend? (Section 4.3.3)

4.1 Experimental setup

All experiments were done on a single machine with the configuration shown in Table 1.

4.2 NICFS tests

The experiment started with a series of tests on NICFS to evaluate the performance of NICFS,

Table 1 Configuration of the machine

Unit	Description
OS	Ubuntu 20.04.1 LTS with Linux kernel version 5.4.0-26-generic
CPU	Intel Xeon Gold 6240 at 2.60 GHz with 72 cores
Memory	Samsung M393A4K40DB2-CVF 64 GB DDR4-2993 DRAM
PMem	Intel Optane persistent memory 100 series 128 GB module
SmartNIC	NVIDIA MBF2H516A-EENOT BlueField-2 DPU with 8 Cortex-A72 cores and 16 GB DDR4-3200 DRAM

OS, operating system; CPU, central processing unit; DDR, double data rate; DPU, data processing unit; DRAM, dynamic random access memory; LTS, long-term support; PMem, persistent memory

examine the design of NICFS, and further suggest some possible optimization directions.

4.2.1 Normal read/write

In the first test, the direct read/write performance of persistent memory and the file read/write performance of three different file systems were evaluated. Of the four objects, Raw PMem refers to persistent memory operating in the device DAX (DEVDAX) mode, wherein the persistent memory is driven as a character device, providing byte addressability and data persistence, thereby achieving the highest performance; NICFS also uses persistent memory in the DEVDAX mode; Ext4 is the typically used file system on SSD; Ext4-DAX refers to the Ext4 file system that uses FSDAX mode persistent memory. FSDAX mode persistent memory is treated as a block device, and the kernel page cache is bypassed when accessing the file system on FSDAX mode persistent memory.

This test started a single process to read/write the above four objects sequentially/randomly with a granularity of 4/16 KB, calculated the performance when processing different jobs, converted it into throughput, and obtained Fig. 8.

First, let us compare Ext4 with Ext4-DAX. It is clear that Ext4-DAX performed better than Ext4 on all tasks. This result suggests that using persistent memory as storage can bring better performance to file systems. However, there was not a large performance gap in the result between the two file systems. This is because the kernel page cache plays a significant role. It intercepts most operations in the cache, concealing the performance difference in storage.

Next, we compare NICFS with Raw PMem. Each operation on Raw PMem will perform only one persistent memory read/write without any redundant logic, which is actually equivalent to the ideal case of a file system on persistent memory storage.

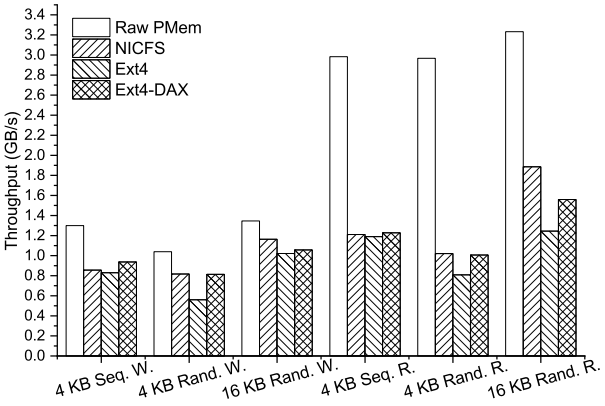

Fig. 8 Comparison of throughputs of different storage objects under different tasks (DAX, direct access; PMem, persistent memory; R., read; Rand., random; Seq., sequential; W., write)

Comparison between NICFS and Raw PMem shows whether NICFS still has room for improvement. As Fig. 8 shows, for write operations, NICFS was near ideal. Since NICFS-Frontend needs to maintain the block cache when performing write operations, which requires one hash table insertion and one queue insertion, there was an additional overhead compared to Raw PMem. The overhead is necessary unless the block cache structure can be further optimized. However, even without optimization, the overhead was still very small, so it can be said that NICFS is already satisfactory for write operations. For read operations, however, NICFS had a significant additional overhead over Raw PMem. The reason is that during read operations, NICFS-Frontend needs to repeatedly read the index blocks in the file system on persistent memory, and also needs to read the block cache and log data from persistent memory to patch the read data. Repeated reads into persistent memory are the cause of inefficiency; thus, the page cache in DRAM is still an essential design, which is not implemented in NICFS-Frontend.

Then, we compare NICFS with Ext4-DAX. While the throughputs of both were very close for

each task, NICFS outperformed Ext4-DAX for random and larger-granularity reads/writes. This is partly due to the append-only log, which converts writes to sequential writes, and partly due to NICFS-Backend, which offloads the file system write from NICFS-Frontend.

Finally, we compare NICFS with Ext4. In all tests, NICFS outperformed Ext4, reflecting the performance improvements (about 10%–21%) brought by persistent memory and SmartNIC at large read/write granularities.

4.2.2 Small read/write

The previous test had a read/write granularity of 4/16 KB, which is actually aligned with the kernel page cache; thus, the performance was good. In this test, to verify that the byte-addressable feature of persistent memory is well used and to reduce the acceleration effect of the kernel page cache, a smaller read/write granularity of 256 B was taken, and the throughputs of the three file systems were retested (Fig. 9).

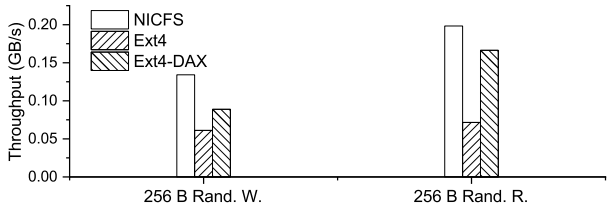

Fig. 9 Comparison of throughputs of different file systems at 256 B read/write granularity (Rand., random; W., write; R.: read)

As the results showed, the performances of NICFS and Ext4-DAX were significantly better than that of Ext4 due to the efficient support of persistent memory for fine-grained reads/writes, and the performance of NICFS was also significantly better than those of the two other because the impact of the kernel page cache was reduced.

Comparing the results of NICFS and Ext4, we can see that persistent memory and SmartNIC can bring ≥100% improvement for small-granularity reads/writes. Compared to Ext4-DAX, there is an improvement of 19% and 51% in reads and writes, respectively.

4.2.3 Scalability

Finally, to examine the ability of NICFS to read/write different files concurrently, this test started four processes for each of the four storage objects, performed random 4 KB reads/writes, and obtained the throughput (Fig. 10).

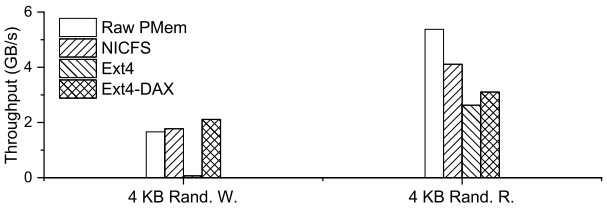

Fig. 10 Comparison of throughputs of different storage objects when doing concurrent reads/writes (PMem, persistent memory; DAX, direct access; R., read; Rand., random; W., write)

The results showed that NICFS performed even better than Raw PMem on random write with four processes. This is because, for NICFS, writes are sequential, and sequential writes are faster than random writes for persistent memory. Arguably, NICFS performed well when writing concurrently. Furthermore, for concurrent read, the performance gap between NICFS and Raw PMem was much smaller than in the single-process case, suggesting that concurrency does not bring much additional overhead to NICFS write but can cover up the overhead. Overall, the concurrency of NICFS was generally ideal. The comparison between Ext4 and Ext4-DAX illustrated that the concurrent read/write capability of persistent memory was excellent.

The cases of ≥4 processes were not tested, because the bandwidth of persistent memory has already reached the maximum, and the performance of NICFS cannot be improved by continuing to increase the number of processes.

4.3 NICFS-Backend

NICFS-Backend is responsible for processing logs, and if logs are not processed in a timely manner, they can cause an accumulation in the circular queue of NICFS-Frontend. When logs pile up too much, the read performance of NICFS-Frontend will be reduced. When logs fill up the circular queue, NICFS-Frontend will be unable to write any more, causing the file system to be unavailable. It is therefore logical to test NICFS-Backend, which helps understand

worst-case performance and whether the system can maintain stability.

4.3.1 Processing log

In this test, only one NICFS-Frontend was connected to NICFS-Backend; it generated a large number of sequential/random write logs with 4/16 KB granularity. The time NICFS-Backend spent on processing these logs was counted. The results were converted to throughput (Fig. 11).

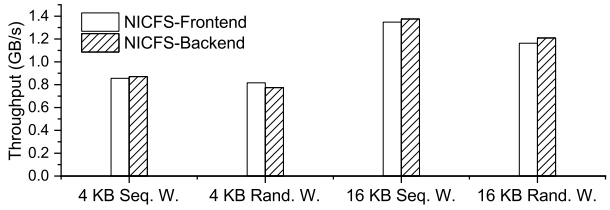

Fig. 11 Comparison of throughputs of NICFS-Frontend and NICFS-Backend on write (Seq., sequential; Rand., random; W., write)

The results suggested that NICFS-Backend generally happened to have slightly higher throughput than NICFS-Frontend. This is not really a coincidence, as the overhead outside the Patch step is hidden by the pipeline in NICFS-Backend. The work done in the Patch step is just writing data to persistent memory, which is the same as in NICFS-Frontend; thus, the two modules have comparable performance. Additionally, NICFS-Backend can effectively merge logs, which allows NICFS-Backend to write fewer data to persistent memory; thus, in some cases, NICFS-Backend performed even better than NICFS-Frontend.

Although the performance of NICFS-Backend was not much better than that of NICFS-Frontend, it was sufficient. This means that we have successfully accomplished migration of the log-processing step from the better CPU to the worse SmartNIC without losing performance. This means that we have successfully offloaded the tasks from the CPU, which is the target of our design.

4.3.2 Scalability

To test whether NICFS-Backend is still capable in the case of multiple NICFS-Frontends, this experiment started 1/2/4 NICFS-Frontends at the same time to continuously generate 4/16 KB random write logs to test the throughput of NICFS-Backend (Fig. 12).

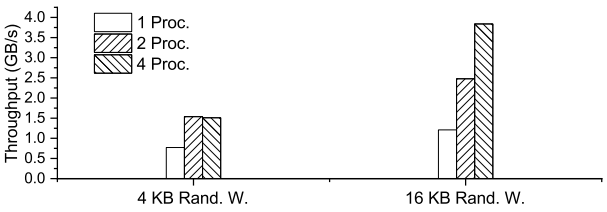

Fig. 12 Comparison of throughputs of NICFS-Backend connected to different numbers of NICFS-Frontends (Rand., random; W., write; Proc., process(es))

For 4 KB write, the throughput of NICFS-Backend grew to two times when there were two NICFS-Frontends. However, when there were four NICFS-Frontends, the throughput of NICFS-Backend decreased instead. The reason for this is not that the Patch step has reached the write throughput limit of persistent memory, but rather the lack of CPU cores on SmartNIC. To serve a single NICFS-Frontend, four threads are required for the pipeline in NICFS-Backend, and there are only eight cores on SmartNIC, which means that all the cores on SmartNIC will be exhausted when there are two NICFS-Frontends. At this point, if more NICFS-Frontends are added, the extra threads will not be able to work at the same time but will instead compete with the original threads for core resources, incurring context-switching overhead and thus leading to performance degradation. Thus, when there are $\geq$8 NICFS-Frontends, NICFS-Backend will still have only four sets of processing pipelines to avoid excessive lock overhead. In the stress test, the performance was comparable to those of four NICFS-Frontends; thus, more NICFS-Frontends were not listed. This is actually sufficient, because for NICFS-Frontend, the bandwidth of four is maxed out; thus, NICFS-Backend needs only to take into account four NICFS-Frontends as well.

The situation was better in the case of 16 KB write since most of the Patch step was occupied by DMA when the cores will be free for other threads. In summary, on one hand, the weak performance of cores leads to long thread-occupation time. On the other hand, the small number of cores leads to the small number of threads that can be served concurrently. These two factors limit the scalability of NICFS-Backend to a certain extent, resulting in the inability to fully use the persistent memory

throughput.

In summary, comparison of Fig. 10 with Fig. 12 shows that NICFS-Backend still keeps up with NICFS-Frontends in terms of throughput, which indicates that our design is able to offload tasks well under our experimental configuration.

4.3.3 Impacts of cache and pipeline

The final experiment was conducted for the index block cache and pipeline parts of the design and was aimed to illustrate the validity and importance of both parts. We removed the cache and pipeline from NICFS-Backend and processed the write logs from a single NICFS-Frontend with 4/8/16 KB granularity (Fig. 13).

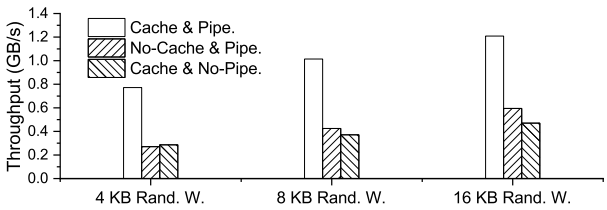

Fig. 13 Comparison of throughputs of NICFS-Backend under three different strategies (Rand., random; W., write; Pipe.: pipeline)

Unlike NICFS-Frontend, NICFS-Backend needs to pass through the PCIe bus when accessing persistent memory. This procedure is slow without a cache. As Fig. 13 shows, the performance slumped when there was no cache. Without the pipeline, the performance of NICFS-Backend dropped to $\leq 0.5\times$, which indicates that the pipeline conceals the cost spent in each step very well.

To explore whether there is room for improvement of NICFS-Backend, inspections into the pipeline are demanded. The percentage of time spent on each step of the pipeline under different loads is presented in Fig. 14.

The entry point for optimizing NICFS-Backend should be to reduce the usage of cores so that better scalability can be achieved. From the results, it is easy to see that the Log Fetch step took almost negligible time, which guides the removal of this step from the pipeline, thus reducing the number of threads. As for the Patch step, although it took up a high percentage, it did not take a lot of core time and can therefore be ignored. For the Merge step, simplifying the log merging algorithm

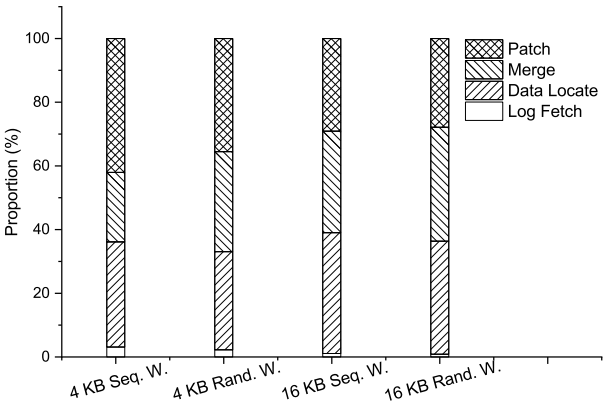

Fig. 14 The percentage of time spent on each step of the pipeline under different loads (Rand., random; Seq., sequential; W., write)

should be a good way to reduce the time consumption. The Merge step took a large percentage in all cases; the allocation of data blocks may be the point that can be optimized by replacing the bitset with a list. Distributing the allocatable data blocks to different NICFS-Frontends to manage without overlap also avoids the usage of mutexes on block allocation in multi-NICFS-Frontend cases.

5 Conclusions

In this paper, a storage architecture NICFS was proposed based on persistent memory for persistent storage and SmartNIC for offloading file processing. A set of data structures and algorithms were designed for logging, caching, merging, processing, and parallelizing suitable for NICFS. NICFS was tested on a machine with persistent memory and SmartNIC, and its performance was slightly better than that of Ext4 with kernel page cache on regular reads/writes, and significantly better than that of Ext4 on small data reads/writes. The experiments also proved that NICFS-Backend has sufficient offloading and processing capability to maintain the efficiency of NICFS-Frontend. However, the shortcoming is that NICFS-Backend is greatly affected by the computing power of SmartNIC itself, resulting in slightly weak scalability.

Contributors

Youyou LU proposed the general idea. Yitian YANG implemented and evaluated the system and drafted the paper. Youyou LU revised and finalized the paper.

Acknowledgements

We thank the members of the Storage Research Group at Tsinghua University for discussion.

Compliance with ethics guidelines

Yitian YANG and Youyou LU declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

Anderson TE, Canini M, Kim J, et al., 2019. Assise: performance and availability via NVM colocation in a distributed file system. https://arxiv.org/abs/1910.05106

Chen YM, Lu YY, Zhu BH, et al., 2021. Scalable persistent memory file system with kernel-userspace collaboration. Proc 19th USENIX Conf on File and Storage Technologies, p.81-95.

Condit J, Nightingale EB, Frost C, et al., 2009. Better I/O through byte-addressable, persistent memory. Proc ACM SIGOPS 22nd Symp on Operating Systems Principles, p.133-146.

Gray C, Cheriton D, 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Oper Syst Rev*, 23(5):202-210. https://doi.org/10.1145/74851.74870

Kim J, Jang I, Reda W, et al., 2021. LineFS: efficient SmartNIC offload of a distributed file system with pipeline parallelism. Proc ACM SIGOPS 28th Symp on Operating Systems Principles, p.756-771. https://doi.org/10.1145/3477132.3483565

Lee C, Sim D, Hwang JY, et al., 2015. F2FS: a new file system for flash storage. Proc 13th USENIX Conf on File and Storage Technologies, p.273-286.

Li JR, Lu YY, Wang Q, et al., 2022. AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing. Proc USENIX Annual Technical Conf, p.951-966.

Liang Z, Lombardi J, Chaarawi M, et al., 2020. DAOS: a scale-out high performance storage stack for storage class memory. Proc 6th Asian Conf on Supercomputing Frontiers, p.40-54. https://doi.org/10.1007/978-3-030-48842-0_3

Lu YY, Shu JW, Zheng WM, 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. Proc 11th USENIX Conf on File and Storage Technologies, p.257-270. https://doi.org/10.5555/2591272.2591299

Lu YY, Shu JW, Wang W, 2014. ReconFS: a reconstructable file system on flash storage. Proc 12th USENIX Conf on File and Storage Technologies, p.75-88.

Lu YY, Shu JW, Chen YM, et al., 2017. Octopus: an RDMA-enabled distributed persistent memory file system. Proc USENIX Annual Technical Conf, p.773-785.

Lu YY, Shu JW, Zhang JC, 2019. Mitigating synchronous I/O overhead in file systems on open-channel SSDs. *ACM Trans Stor*, 15(3):17. https://doi.org/10.1145/3319369

NVIDIA, 2022. DOCA. https://developer.nvidia.com/networking/doca [Accessed on Oct. 8, 2022].

Ou JX, Shu JW, Lu YY, 2016. A high performance file system for non-volatile main memory. Proc 11th European Conf on Computer Systems, Article 12. https://doi.org/10.1145/2901318.2901324

Schuh HN, Liang WH, Liu M, et al., 2021. Xenic: SmartNIC-accelerated distributed transactions. Proc ACM SIGOPS 28th Symp on Operating Systems Principles, p.740-755. https://doi.org/10.1145/3477132.3483555

Xu J, Swanson S, 2016. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. Proc 14th USENIX Conf on File and Storage Technologies, p.323-338.

Zhang JC, Shu JW, Lu YY, 2016. ParaFS: a log-structured file system to exploit the internal parallelism of flash devices. Proc USENIX Annual Technical Conf, p.87-100.