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

A novel non-volatile memory storage system for I/O-intensive applications*

Wen-bing HAN[†1,2], Xiao-gang CHEN[†‡1], Shun-fen LI[1], Ge-zi LI[1,2], Zhi-tang SONG[1], Da-gang LI[3], Shi-yan CHEN[3]

[1]*Shanghai Institute of Micro-system and Information Technology, Chinese Academy of Sciences, Shanghai 200050, China*

[2]*University of Chinese Academy of Sciences, Beijing 100080, China*

[3]*School of Electronics and Computer Engineering, Peking University, Shenzhen 518055, China*

[†]E-mail: hwbx@mail.sim.ac.cn; chenxg@mail.sim.ac.cn

Received Jan. 20, 2017; Revision accepted Aug. 8, 2017; Crosschecked Oct. 15, 2018

Abstract: The emerging memory technologies, such as phase change memory (PCM), provide chances for high-performance storage of I/O-intensive applications. However, traditional software stack and hardware architecture need to be optimized to enhance I/O efficiency. In addition, narrowing the distance between computation and storage reduces the number of I/O requests and has become a popular research direction. This paper presents a novel PCM-based storage system. It consists of the in-storage processing enabled file system (ISPFS) and the configurable parallel computation fabric in storage, which is called an in-storage processing (ISP) engine. On one hand, ISPFS takes full advantage of non-volatile memory (NVM)'s characteristics, and reduces software overhead and data copies to provide low-latency high-performance random access. On the other hand, ISPFS passes ISP instructions through a command file and invokes the ISP engine to deal with I/O-intensive tasks. Extensive experiments are performed on the prototype system. The results indicate that ISPFS achieves 2 to 10 times throughput compared to EXT4. Our ISP solution also reduces the number of I/O requests by 97% and is 19 times more efficient than software implementation for I/O-intensive applications.

Key words: In-storage processing; File system; Non-volatile memory (NVM); Storage system; I/O-intensive applications

https://doi.org/10.1631/FITEE.1700061 **CLC number:** TP333

1 Introduction

As datasets increase at an exponential rate (Szalay and Gray, 2006), we have entered the data-centric era of big data. Because the speed of information growth exceeds Moore's law (Chen and Zhang, 2014), excessive I/O-intensive application data present enormous challenges to computer storage, including storage devices, storage architectures, and data access mechanisms, which require extremely-low-latency, effective, and random-access I/O data requests.

The general approach for obtaining good I/O performance is caching the entire dataset in main memory. However, it is not reasonable or satisfactory for big data applications because of their large-scale datasets, and high power consumption in DRAM, caused mainly by refresh power, is more severe with its growing capacity. Furthermore, DRAM has already reached its scaling limits. Instead, we can

‡ Corresponding author

* Project supported by the National Basic Research Program of China (No. 2017YFA0206101), the National Defense Innovation Fund of Chinese Academy of Sciences (No. CXJJ-16M106), the "Strategic Priority Research Program" of the Chinese Academy of Sciences (No. XDA09020402), and the Science and Technology Council of Shanghai, China (Nos. 14DZ2294900, 13ZR1447200, and 14ZR1447500)

ORCID: Wen-bing HAN, http://orcid.org/0000-0002-0370-7023

store datasets in cheap, high-density secondary storage, such as hard disk drives (HDDs), whose random I/O performance is much poorer than sequential I/O performance. Considering its characteristics, the storage system has to reorder and merge incoming randomly ordered requests to minimize the search time wasted by the hard disk. However, the storage system still has average latency in milliseconds. As a result, the primary performance bottleneck is poor I/O storage devices.

The recent development of high-performance non-volatile memory (NVM), represented by the phase change memory (PCM) technique, provides new choice for improving I/O performance. Compared with DRAM, PCM can store persistent data after a power failure. It has lower cost, smaller device scaling, and higher density. On the other hand, PCM is byte-addressable and offers superior random-access performance and write endurance compared to flash, let alone HDDs. Such advantages make PCM an alternative to flash devices and HDDs, which helps bridge the access-time gap between memory and storage.

Instead of storage devices, the remainder of the storage system becomes the key bottleneck in I/O performance, such as software stack overhead. A recent study reported that the costs of software and block transport layers represent between 1.2% and 3.8% of total latency for a disk-based storage area network (SAN). For flash-based solid state disks (SSDs), the percentage climbs as high as 59.6%, whereas software overhead of more advanced memories accounts for 98.6% to 99.5% of latency (Caulfield and Swanson, 2013). Because of different storage device characteristics, traditional software designed for flash or HDDs, including file systems and I/O subsystems, is never applicable for PCM. As a consequence, redefining a file system, simplifying the I/O stack, and promoting I/O efficiency to realize the full potential of PCM devices, are the main motivations for our work.

Another approach for enhancing system performance is minimizing the number of I/O requests. In the modern computer system, moving computation is cheaper than moving data. In-storage processing (ISP) adds a computation fabric on top of the storage devices to filter data, which is regarded as the ultimate solution for accelerating I/O-intensive applications (SAMSUNG, 2015). It transports the tedious data manipulation tasks to the storage itself, rather than moving a large amount of data from storage devices to the CPU. Although the ISP concept can be traced back to the 1970s (Jun et al., 2015), it has no unitary specifications so far. In fact, there are two major issues in the ISP research field now. One issue is how to establish the ISP channel between hosts and computation fabrics. Sending commands and returning results are significant portions of ISP, and no detailed discussions have been given in the literature. Another issue is that the computation fabric is not able to figure out data. Most prior work accesses data through a specific address and does not know what the data mean. Also, previous research into ISP was always based on SSDs, which implement the computation fabric on an ARM controller in the SSD. However, programmable, highly parallel, high-speed hardware fabric such as FPGA is a more effective choice for ISP. Thanks to the development of ASIC chips, the requirement of low power consumption can be met.

In this paper, we propose a novel NVM-based storage architecture for big data applications. It leverages high I/O performance from two perspectives: enhancing I/O efficiency and minimizing the number of I/O requests. We integrate them into a novel file system called ISPFS, designed for NVM and enabling ISP. Our storage system aims to achieve the goals set out below:

1. Low latency and high bandwidth: To increase the efficiency of I/O-intensive applications, PCM devices should be closer to the CPU, whereas the file system and I/O stack need to be optimized and simplified.

2. Parallel and low-latency ISP engine: To reduce CPU consumption on data migration and achieve high utilization of storage internal bandwidth, a low-latency and parallel ISP engine should be provided to accelerate data processing.

3. Application compatibility: The applications that already exist can work on our storage system without any modification. Moreover, users can add a set of ISP instructions to their applications to see a tremendous performance gain with negligible overhead.

Due to high-performance PCM devices and specific architecture design, our storage system can provide low-latency random access to data. The proposed file system is established on the physical

address space and uses a memory management unit (MMU) to deal with address mapping. To reduce software overhead, the page cache and block device layer of the I/O subsystem are both eliminated. To further reduce the number of I/O requests, we implement the ISP computation fabric with a configurable FPGA. Because it is established beside the data path of the storage controller, through which data must travel, there is no extra overhead introduced from the ISP engine. Given that the file system manages data and the ISP engine processes data, it seems fairly natural to combine the file system with the ISP engine. We add the command file into our file system to establish the ISP channel. According to the structured instructions from hosts, the ISP engine can analyze the file's structure, find the address of the file content, and process data-intensive computation. Our ISP design focuses on encapsulating the data and computation into a file. It is notable that we only abstract the basic data processing instructions from I/O-intensive tasks to establish a simple data filter, instead of implementing a special algorithm in the ISP engine for a particular situation.

The key contribution of our work is a novel storage system for I/O-intensive applications, including a new file system (ISPFS) designed for NVM and a low-latency ISP engine. The file system improves I/O efficiency by reducing software overhead and applying a PCM storage device. Combined with the ISP engine, ISPFS also minimizes the number of I/O requests by reducing large volumes of data. As a result, our storage system offers high-performance storage and file-oriented ISP capabilities. We demonstrate the characteristics of our storage architecture on a prototype system.

To evaluate our design, we constructed the prototype system by coupling a commercially available ZYNQ platform (XILINX, 2014) with a custom PCM storage board. ISPFS was implemented on the host platform and was significantly faster than EXT4 by 1 to 9 times. With support from an FPGA-based ISP engine, we implemented a word query application based on the ISPFS. Our ISP solution achieved an almost 19-fold performance improvement over a pure software solution.

2 Related work

New nanostorage technology, represented by PCM, makes it possible for non-volatile storage chips to achieve high density, low latency, and high-performance random access. PCM chips offer access latency on the order of tens of nanoseconds, which is several orders of magnitude less than the 25- to 500-μs flash access latency (Li Z et al., 2016). The storage system can achieve high throughput by organizing multiple chips in parallel. Thus, the bottleneck of I/O efficiency shifts from storage devices to software and the I/O subsystem. It has been shown that software stack overhead accounts for only 0.3% of the total storage access latency in HDD environments, but up to 94.1% in NVM environments depending on the interface (Lee et al., 2014). As a consequence, modern storage systems need to optimize the software stack, especially the file system, to enhance I/O efficiency.

Researchers can often reduce software overhead by designing a specific file system for NVM. BPFS uses the short-circuit shadow paging (SCSP) technique, including in-place update, in-place append, and partial copy-on-write mechanisms, to provide fine-grained, atomic, and consistent updates on NVM storage medium (Condit et al., 2009). It can commit updates at any level in the file system tree through subtree modifications, which minimizes copy costs of conventional shadow paging. However, SCSP is guaranteed by special hardware supports, which is difficult to implement. SCMFS is established on the virtual address space, where the whole file system is mapped by the MMU (Wu and Reddy, 2011). To reduce the overhead of frequent allocation/de-allocation, it uses a null file to pre-allocate space and provide a garbage collection mechanism. Furthermore, SCMFS employs HugePage to decrease translation lookaside buffer (TLB) miss rates and improve performance (Cao, 2012). NVMFS is an implementation of storage class memory file system (SCMFS) on a hybrid architecture of NVM and SSDs (Qiu and Reddy, 2013). It stores metadata and hot data in the NVM, which is maintained by the LRU lists with dirty and clean marks, to increase cache hit rates and avoid unnecessary copies between NVM and SSDs. Unfortunately, as files are consistently added, removed, and changed in size, the free space of such file systems becomes

externally fragmented, leaving only small holes in which to put new data. To obtain efficient access into NVM, SIMFS incorporates file data into the virtual address space and bypasses traditional software layers in the I/O stack (Sha et al., 2015, 2016). The file data are organized in a file page table, which has contiguous virtual address space and the same structure to a process page table. The file virtual address space of an opened file is embedded into the calling process's address space when applications access file data. Nonetheless, SIMFS still suffers from file system fragmentation problems.

As an effective solution to decrease the I/O request number and data migration, ISP has aroused extensive interest in academia and industry in recent years. Micron proposed an architecture enabling near-memory acceleration for the data center, called ScaleIn, which can offload tedious data manipulation tasks to the massively parallel SSD subsystem (Doller et al., 2014). The ScaleIn system with three drivers can rival the performance of MySQL on servers with cost and energy reductions. However, the lowest query latency of ScaleIn is always greater than that of the baseline system, which implies that block access to flash and traditional software stacks still introduce a large amount of overhead. The smart SSD model was proposed by Samsung to integrate in-storage compute (ISC) in SSD architecture (Do et al., 2013; Kang et al., 2013). Samsung then presented an ISC prototype, which is called ultimate close-to-data computing for high performance and low power (SAMSUNG, 2015). It is a software and hardware co-design framework in which developers can implement custom hardware accelerators on the SSDs' firmware while they develop the host applications. However, this programming model requires a new accelerator for every application. Although ARM is a low-power processor, it is not suitable for data-intensive applications because of its weak parallel arithmetic capability. BludDBM (Jun et al., 2014, 2015) provides an FPGA-based reconfigurable fabric for implementing hardware accelerators near storage. It starts using the file system, such as the FUSE virtual file, to transport ISP commands. However, its hardware accelerator is not aware of the structure of the file contents to be processed. We hope to further improve performance and compatibility through tight connections between ISP and the file system.

3 System architecture

Our storage system is composed of a PCM board and a host platform with heterogeneous SoC (FPGA and ARM). The PCM storage board is coupled with the host platform via the FPGA mezzanine card (FMC) interface. It includes PCM chips and an onboard FPGA. A specific PCM controller is designed on the reconfigurable FPGA fabric to organize raw PCM chips into buses, and an ISP engine is built on top of the PCM controller to perform computation right at the data source. It can work as both a raw storage device and a computing device with the ISP processor inside.

On the host board, the ARM core runs high-level applications that can generate read, write, or ISP commands to the proposed file system. Then ISPFS forwards all commands to the peripheral logic using the attached FPGA, which transports the requests to the PCM array or ISP engine registers. To take full advantage of the PCM's random access, we mount PCM storage on the memory bus and implement our file system in the addressing space of the CPU. In other words, the CPU can converting data access requests of the file system into load/store operations. As a result, the application could access byte-addressable persistent storage like memory, while instructing the ISP engine to process the data directly from the PCM controller.

This architecture fulfills our previous goal. First of all, we construct a low-latency, high-bandwidth storage system using fast random access PCM chips, the hardware frame which enables CPU direct addressing, and the optimized file system without block layers. Second, we implement a highly parallel ISP engine with reconfigurable FPGA. Our file system provides a hook for applications to invoke computation operations on data without passing through the host. Third, the proposed file system provides a generic POSIX interface for compatibility issues.

Fig. 1 illustrates the hardware and software stacks of our storage system. From the software perspective, we implement a novel file system that is specially designed for NVM and supports ISP. From the hardware perspective, there are four key components running on the FPGA fabric: external memory controller (EMC) interface, FMC connector, ISP engine, and PCM controller. The EMC interface handles the communication of data from the host

memory bus. Together with our file system on the host, it maps addresses of PCM data and ISP registers into the specific memory address space, which is directly exposed to the CPU. The FMC connector transfers ISP commands and data between the host board and storage board. The PCM controller provides a set of commands that are compatible with the LPDDR2-NVM protocol to access the raw PCM chips on the storage board. The innovative file system, ISP engine, and PCM controller are explained in detail below.

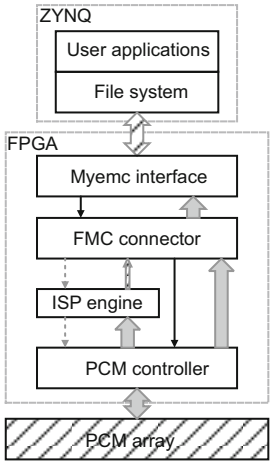

Fig. 1 Architecture of the proposed storage system

3.1 File system design

We have proposed a novel random-access file system that is specifically optimized for NVM (Zhou et al., 2016). To be compatible with software products, our file system exports an interface that is identical to the conventional file system. Therefore, the existing applications in the industry can be run on top of it without modifications. Our goal is to take full advantage of NVM and improve data access efficiency. We establish the associated hardware infrastructure on the embedded platform and implement the novel file system (Han et al., 2016). To explore the random-access property and low latency of PCM, we connect the PCM array to the memory bus through the PCM controller. As a result, the PCM storage device can be directly addressed by the CPU via the regular load/store instructions. Because of unified memory addressing of the DRAM and PCM, we can implement our file system on the physical address space of the PCM. Fig. 2 shows the space layout of our file system, including the super block, ISP registers, inode table, block in-use bitmap, and data space. ISP registers support ISP in the file system and will be discussed next. We omit the remainder because they are quite similar to the regular UNIX-like file system.

As depicted in Fig. 2, we use MMUs to manage the virtual memory addresses of the file system, which primarily performs the translation of virtual memory addresses to physical addresses. Because of the random-access property of PCM, we eliminate the original generic block layer, I/O scheduler, and block device drivers of the I/O subsystem. It shortens the data path between the CPU and storage devices in terms of the software stack and enhances the small file performance of our file system. Generally, applications always need two copy operations to access data in storage devices. The first operation is copying file data from storage to the page cache in memory; the second operation is copying data from the page cache to the address space of the process. We modify the page fault exception handler to bypass the complex page cache module and eliminate the first copy. The page table descriptor can obtain the physical address of data in PCM storage without copying it to memory.

In addition, we use MMAP, a method of memory-mapped file I/O, to map a file to the virtual address space of the calling process. It removes the second copy between the kernel space and user space. The process can manipulate files by the mapping pointer instead of invoking a read or write system call. As a result, we implement zero-copy techniques in the proposed file system. On the basis of this, our file system provides the execute in place (XIP) feature to execute programs directly from their storage location. It speeds up execution by omitting data migration and reducing the total amount of memory required.

To reduce the number of I/O requests and data migration, we achieve ISP functionality based on the above features for our file system, called ISPFS. ISPFS is implemented on the basis of EXT2 and is written in GNU C in about 7000 lines of code. There is a command file in every ISPFS directory. The command file will be automatically created when its parent directory is created. Applications can access this command file through POSIX functions, including reading and writing. Reading ISP results and writing ISP instructions are the most fundamental

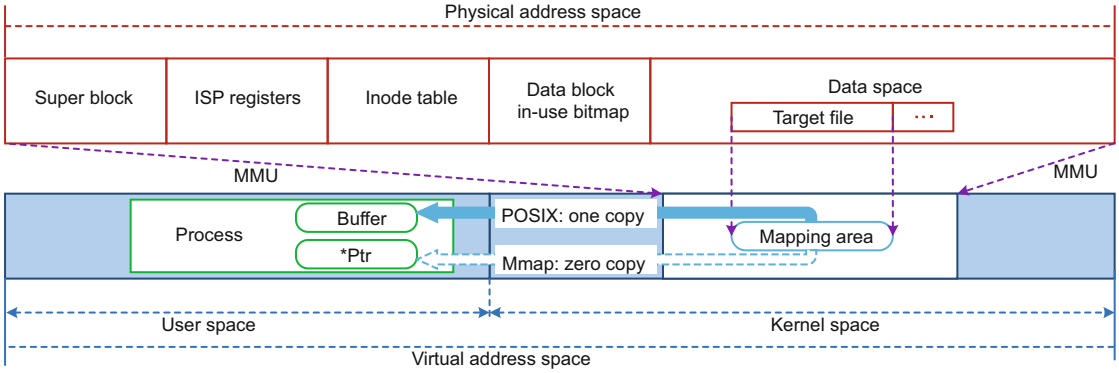

Fig. 2 Space layout of the in-storage processing enabled file system (ISPFS)

operations. As a virtual file, the command file does not have actual data space. Instead, it takes up only the space of an index node structure. There will be just one command file in a directory. As a consequence, it occupies an extremely small region of storage compared to the entire storage device.

The command file is similar to pipes to some degree. We modify its read and write function handler to enable ISP functionality in the file system, so ISPFS refactors read and write functions of the command file. The ISP mechanism of our file system is presented in Fig. 3. When applications write ISP instructions to the command file, the proposed file system will intercept and parse these instructions. ISPFS looks up its inode number in the directory entry cache according to the filename in the instructions. Then it passes the inode number and other parameters to the ISP engine registers. These ISP registers are mapped to the specific physical address when the file system is mounted. Finally, the file system sets the start register to run the ISP engine. If applications need to read results from the command file, our file system will copy the values of the ISP engine result registers to data buffers, and then return results to applications.

3.2 ISP engine implementation

We use an FPGA to design and implement the ISP engine. The FPGA is more and more popular as a hardware accelerator for big data applications because of its low power cost, flexibility, and high parallelism (Jun et al., 2014). When the start register is set, the ISP engine parses the received instructions according to the relevant register values. The address mapping relationship of these registers

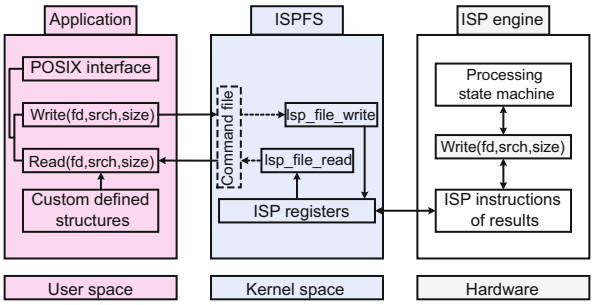

Fig. 3 In-storage processing mechanism

is specified and maintained by the ISP engine. As shown in Fig. 4, we implement a finite state machine (FSM) for every data processing instruction. The corresponding FSM will be triggered by the ISP instructions. The ISP engine can follow the file system's data management method, and calculate the file content's block address through the inode number. Coupled with the file offset, it can determine the physical address of the data actually needed. Therefore, our ISP engine is file-oriented rather than address-oriented, which is one of the fundamental reasons why we combine the file system and ISP. Data length can also be specified in the ISP instructions to offer flexible support. The corresponding FSM accesses file data by invoking the LPDDR2-NVM controller. Finally, it processes data in parallel and writes calculation results into the results buffer, waiting for reading operations of the file system.

As shown in Fig. 4, we implement the ISP engine beside the storage device data path, rather than as a separate appliance. The ISP engine can access data directly through the PCM controller. As a result, intensive operations on data are hidden in the storage devices, which dramatically reduces the overhead because a large quantity of data is migrated

between memory and storage. However, a separate appliance method is needed to transport data from storage to the ISP engine via the host, and then transport results back after processing. Although it reduces the amount of computations, it cannot free the CPU from the heavy work of data migration, which is exactly the opposite of the ISP concept.

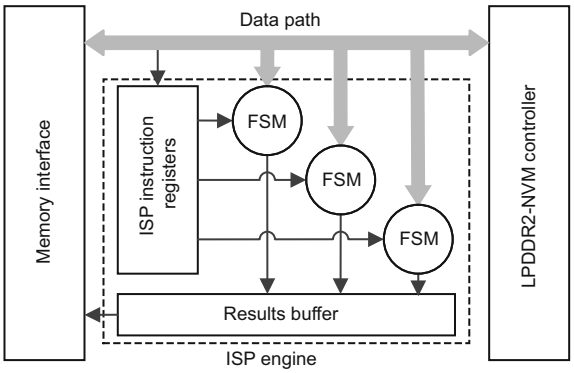

Fig. 4 In-storage processing (ISP) engine implementation

In addition, we handle read operations, write operations, and ISP operations at the same level. ISP operations have lower precedence than reading and writing operations. When the hosts need to access the storage device, the ISP engine will be bypassed from the data path. This guarantees that the access requests receive an immediate response. While the ISP engine is running, read and write operations can preempt the LPDDR2-NVM controller to handle requests, but the ISP engine continues running. So, ISP not only has no performance impact on data access, but also improves the inner bandwidth utilization of storage devices.

As can be seen in Fig. 5a, an ISP instruction is composed of four parts: commands, file information, results, and ISP status. Commands specify the types of calculations with a command identifier and provide necessary parameters. File information consists of the filename, inode number, offset, and length, which limits the data scope. The results refer to computation results or their virtual address in the results buffer. The ISP status consists of the completion flag and time. An ISP instruction is packaged into a structure that will be invoked by reading or writing functions. Fig. 5b shows an example of an ISP instruction that queries four words in a file.

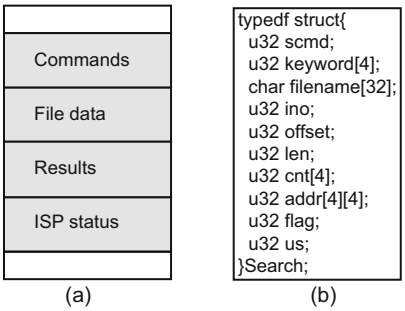

Fig. 5 In-storage processing (ISP) instruction organization: (a) four parts of an ISP instruction; (b) one example of an ISP instruction

3.3 LPDDR2 controller

We use PCM chips to build our fast random-access storage array. With PCM pins connected to the byte group I/Os of the FPGA, we implement the PCM controller to arrange these raw chips. The data bus of PCM chips is 16 bits, so we group two chips to act as a 32-bit bus. Each controller with a 32-bit bus manages two groups. PCM chips of different groups share the data and control buses, whereas they have their own chip selection signals. There are two PCM controllers on the board, and they operate in parallel to make up a 64-bit data bus. Our PCM controller has been modified from a Xilinx MIG controller, which has an LPDDR2-NVM interface. We are also in the process of implementing an original LPDDR2-NVM controller for PCM chips.

3.4 Example applications

Based on the above design, we implemented a word query application (Li GZ et al., 2016) on our proposed system. We defined a structure to represent the ISPFS query instruction, which consists of four words being queried, filename, offset, count values, etc. The proposed file system and ISP engine both support parsing that structure. Applications can pass it to the ISP engine through the write function as follows:

write(fd_cmd, srch, sizeof(search)).

In this statement, fd_cmd is the file descriptor of the command file and srch is an structure search instance. ISPFS will transform the filename into the inode number and pass all parameters to the ISP engine. Once the words are registered, the ISP engine accesses the physical address of the file data

to query the counts and addresses of four words in PCM storage devices. Applications can also read results from the ISP engine through the reading function as follows:

read(fd_cmd, srch, sizeof(search)).

This function informs the file system to read ISP engine result registers. When the computation is complete, the status register is set. The application needs to poll the status register to determine whether the return values are valid. Fortunately, this action does not take a long time due to the high performance of the ISP engine. We introduce hardware interrupts to optimize this mechanism in the next step.

To measure the performance of the proposed storage system, we need to time the latency of the ISP engine. However, the timing error of the embedded ARM core is always huge when Linux system resources are constrained. Consequently, we implement a hardware timer in the ISP engine, which can be accessed as described above.

4 Prototype system

Our prototype storage system consists of the low-cost development platform ZedBoard and the custom-built PCM storage board. It is shown in Fig. 6 that the PCM storage board is plugged into ZedBoard via the FMC connector.

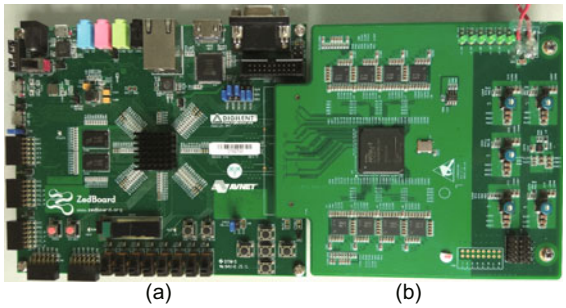

Fig. 6 Prototype system: (a) ZedBoard; (b) PCM board

ZedBoard uses a Zynq-7000 all programmable SoC as a processor, which integrates a feature-rich dual-core ARM CortexTM-A9 MPCore™ based processing system (PS) and Xilinx programmable logic (PL) in a single device (XILINX, 2014). We implement peripheral logic in the PL to handle FMC communications and exchange data with the custom PCM board. In the Zynq-7000 SoC, it is feasible for the CPU to load data into the L2 cache through the advanced extensible interface (AXI) bus, so we mount the peripheral logic on the AXI bus and specify the physical address space for it. As a result, the CPU can access PCM directly through the specific address space, which is part of the foundation of the proposed file system.

On our PCM storage card, there is 1 GB of PCM storage, which consists of eight 128-MB Micron PCM chips. With the LPDDR2-NVM JEDEC-compatible interface, Micron PCM chips achieve the throughput of each PCM chip to a maximum of around 150 MB/s at a clock rate of 250 MHz. An on-board FPGA handles the FMC communications, and implements the LPDDR2-NVM interface and ISP engine for all data buses. The detailed hardware configuration is as listed in Table 1.

Table 1 Hardware configuration of proposed system

Unit	Description
CPU	Dual ARM® Cortex™-A9, 667 MHz
L1 cache	32 KB instruction, 32 KB data
L2 cache	512 KB
Memory	512 MB DDR3 533 MHz
PCM chips	Micron LPDDR2-PCM (MT66R7072A), 16-bit data bus, 400 MHz
FPGAs	XILINX® ARTIX™-7 (XC7A35T), 33 280 logic cells, 1800 KB block RAM

To achieve extremely low latency for the ISP engine, we implement it beside the data path of the storage device, rather than as a separate appliance. For ordinary read or write operations, data will bypass the ISP engine and be transferred to the CPU directly. If there are ISP instructions, the ISP engine will decode the command signals and start corresponding with the state machine to handle this transaction. The ISP engine moves computation from the CPU to storage devices and reduces the amount of data being migrated. A separate appliance transports data from the storage device to the ISP engine via the host CPU and then returns results after processing. The fact that the FMC connector works at a clock rate of 100 MHz and in simplex mode is a performance bottleneck of our storage system. The ISP engine can help process more data than the FMC connector allows.

The Linaro distribution of Linux is run on top of the ZedBoard platform. We mount the ISPFS on the specific address space to interface with the PCM storage chips and implement data-intensive applications to invoke the ISP engine via the proposed file system.

5 Results

Based on the prototype system, we first compared ISPFS with EXT4, the typical existing file system, on throughput. Then we measured the performance of I/O-intensive applications, such as word query, compared with traditional software solutions. Finally, we examined CPU and memory utilization and the access time distribution of two solutions.

5.1 Throughput of the file system

To evaluate the effectiveness of our file system, we measured the throughput of ISPFS and EXT4 via the widely used benchmark IOZONE (Norcott and Capps, 2016). IOZONE can reflect the I/O efficiency improvements of ISPFS over the regular file system. Because it is hard to implement EXT4 on PCM chips, we used DRAM as a storage medium and mounted EXT4 on the RAMDisk block device to provide equivalent hardware conditions. Because EXT4 journaling leads to extra I/O, we disabled journaling in the experiment. In Fig. 7, we accessed the file with POSIX and MMAP interfaces and performed experiments with four workloads. The file size was specified as 256 MB, and the record size varied from 1 KB to 16 MB to test the performance trend of the file system.

Fig. 7 illustrates the throughput of ISPFS and EXT4 with various configurations of IOZONE. For reread and rewrite, ISPFS is 1.34 and 4.10 times faster than EXT4 on average, respectively. For random read and random write, ISPFS is 3.32 and 8.30 times faster than EXT4 on average, respectively. This is because the EXT4 file system still depends on the generic block layer and page cache mechanism, even though metadata and file data are already stored in memory. ISPFS is robust for different record sizes of the application. For all sizes of I/O requests, it constantly outperforms EXT4 on RAMDisk. As record size decreases, the performance of EXT4 degrades dramatically. In the worst cases, the throughput of ISPFS for random read and write can reach 26 and 39 times those of EXT4, respectively. This is because EXT4 still goes through all the software layers of traditional I/O stacks, which introduces more complexity and overhead, especially when accessing small files. In POSIX mode, because ISPFS saves one copy from disk to memory cache compared with EXT4, the throughput is twice that of EXT4. In MMAP mode, ISPFS offers a zero-copy technique by modifying the page fault handler. Consequently, the read throughput of the ISPFS is 1 to 4 times faster than that of EXT4 and the write throughput is 5 to 9 times faster than that of EXT4.

5.2 Word query application

In Fig. 8, the polylines indicate the search time and speed of ISPFS and a traditional software solution, with the file size ranging from 0.1 MB to 100 MB. The average speed of our proposed system reaches 74 MB/s, while that of the software solution is only 3.59 MB/s. The ISP solution is about 19 times faster than traditional software methods.

There is no fluctuation in the search speed of the proposed system with mixed file sizes. No matter how large the file is, the ISP solution spends the same amount of time in computing its physical address. The relationship between search speed and file size is approximately linear. The traditional software solution needs to open files and migrate data to memory first, which causes significant overhead for I/O-intensive applications. Small files are queried more slowly than large files because the time required to open small files takes up a larger portion of the whole search time. The CPU scheduling policy of the operating system also affects the search procedure; as a result, the software solution's search speed varies, especially for tiny files.

5.3 Resource usage and access request analyses

Fig. 9a presents the CPU and memory utilization rates of ISP and the software solution. Regardless of the file size, host CPU utilization varies significantly between the two systems. The software implementation uses 100%, whereas the ISP solution uses less than 3%. There is a linear relationship between memory consumption and file size in the software solution, but the ISP solution occupies only a small percentage of memory (less than 0.3%).

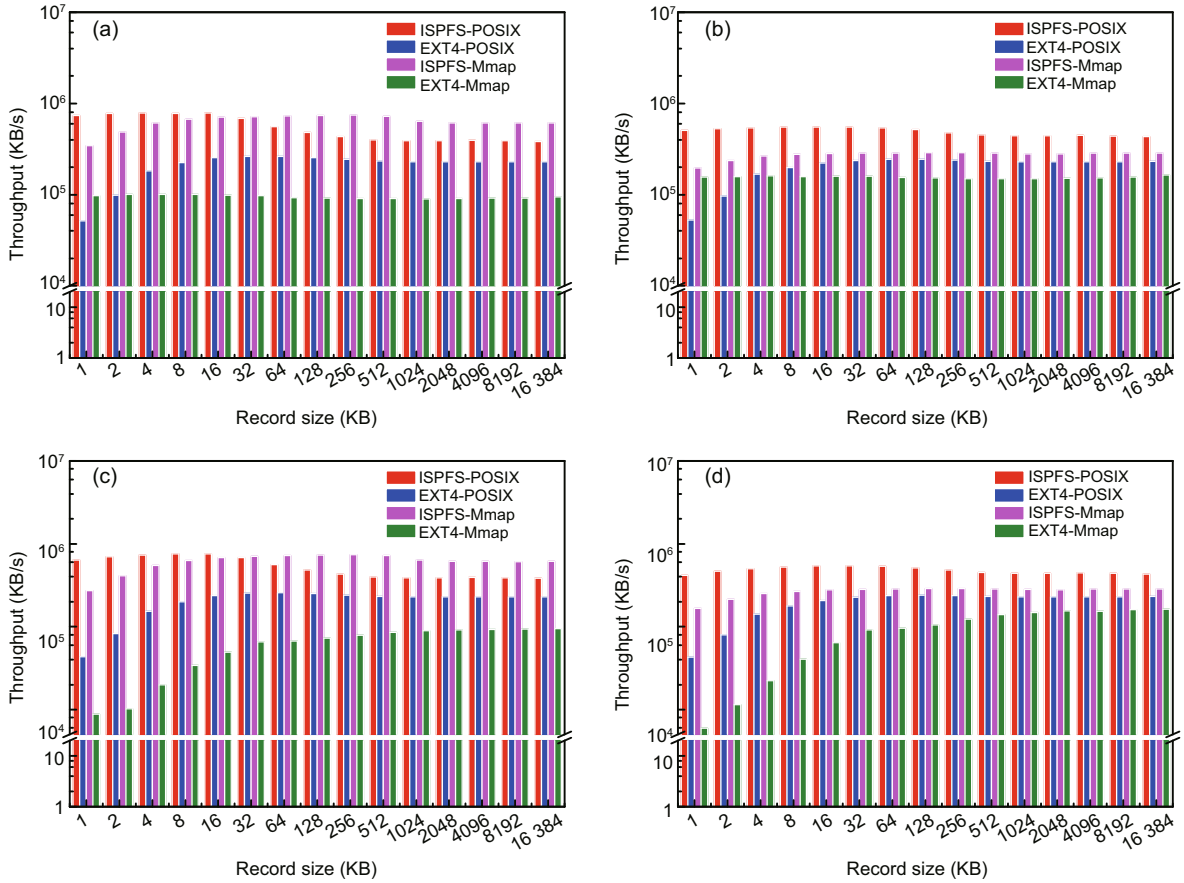

Fig. 7 Comparing throughput for ISPFS using IOZONE: (a) rewrite; (b) reread; (c) random write; (d) random read

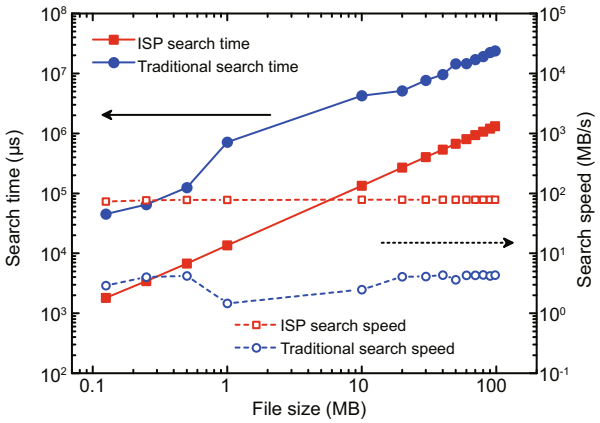

Fig. 8 Word query application performance of ISPFS and software solution

This significantly impacts the energy consumption and performance of the whole system.

A large number of I/O requests are reduced in the ISP solution. In the past, we have used blktrace, a block layer I/O tracing mechanism, to count the number of I/O requests of the software solution (Axboe et al., 2006), but the generic block layer has been removed in ISPFS, so we regard system calls as I/O requests in our file system. The software solution generates 833 I/O requests, whereas the ISP solution has only 17 system calls (Fig. 9b). The I/O requests are reduced by 97% with ISPFS. Fig. 9b also illustrates that file loading time has the same order of magnitude as search time in the software solution, which is 20.7 s in total. However, the ISP solution processes a search operation in the storage devices where the data is stored and saves the data migration time. The application spends about 2 ms sending ISP instructions to the hardware engine by ISPFS. The experimental results indicate that our proposed method introduces negligible overhead to the existing system. In fact, the words query task is already finished in the ISP solution in the time required for the software implementation to load file data to the buffer cache.

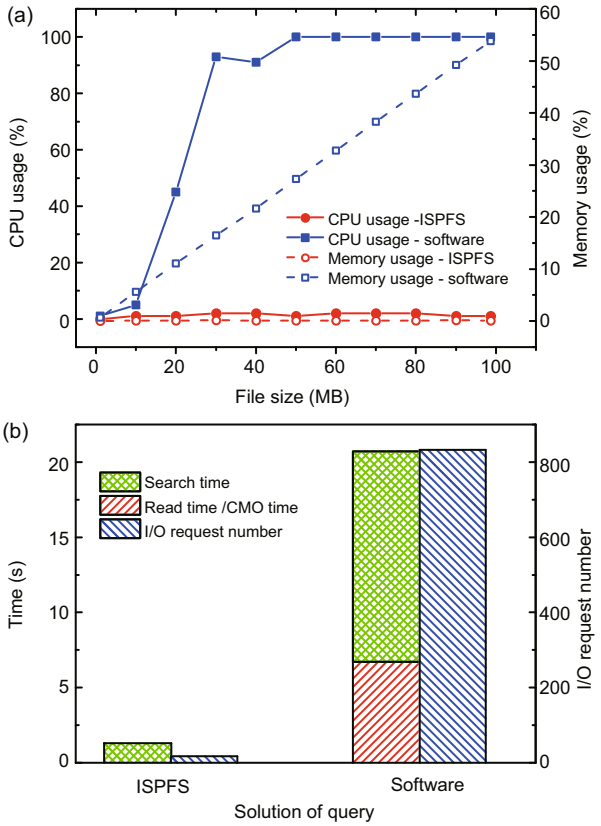

Fig. 9 The resource usage (a) and access time and I/O requests (b)

6 Conclusions

In this paper, we have proposed a novel storage architecture based on NVM, and on that basis, we design a new file system that enables in-storage processing with a reconfigurable fabric engine. To fully explore NVM's characteristics, our file system (called ISPFS) eliminates the page cache and modifies the page fault handler to implement zero-copy and XIP techniques. To move computation to storage devices and reduce data migration, ISPFS provides an ISP instruction channel by accessing the command file. The file system intercepts the message between the application and the command file and invokes an ISP engine to perform related data tasks right at the data source. Experimental results demonstrate that the throughput of ISPFS is consistently superior to that of EXT4, and that ISPFS provides higher I/O efficiency. Furthermore, the proposed ISP solution is about 19 times more efficient than the pure software implementation and reduces the number of requests for I/O-intensive applications by 97%.

References

Axboe J, Brunelle AD, Scott N, 2006. blktrace(8) - Linux man page. https://linux.die.net/man/8/blktrace [Accessed on Jan. 19, 2016].

Cao Q, 2012. SCMFS Performance Enhancement and Implementation on Mobile Platform. MS Thesis, Texas A&M University, College Station, Texas.

Caulfield AM, Swanson S, 2013. QuickSAN: a storage area network for fast, distributed, solid state disks. Proc 40th Annual Int Symp on Computer Architecture, p.464-474. https://doi.org/10.1145/2485922.2485962

Chen CLP, Zhang CY, 2014. Data-intensive applications, challenges, techniques and technologies: a survey on big data. *Inform Sci*, 275:314-347. https://doi.org/10.1016/j.ins.2014.01.015

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. https://doi.org/10.1145/1629575.1629589

Do J, Kee YS, Patel JM, et al., 2013. Query processing on smart SSDs: opportunities and challenges. ACM SIGMOD Int Conf on Management of Data, p.1221-1230. https://doi.org/10.1145/2463676.2465295

Doller E, Akel A, Wang J, et al., 2014. DataCenter 2020: near-memory acceleration for data-oriented applications. Symp on VLSI Circuits Digest of Technical Papers, p.1-4. https://doi.org/10.1109/VLSIC.2014.6858357

Han WB, Chen XG, Zhou M, et al., 2016. The storage system of PCM based on random access file system. Proc SPIE, 9818:98180G. https://doi.org/10.1117/12.2245028

Jun SW, Liu M, Fleming KE, et al., 2014. Scalable multi-access flash store for big data analytics. Proc ACM/SIGDA Int Symp on Field-Programmable Gate Arrays, p.55-64. https://doi.org/10.1145/2554688.2554789

Jun SW, Liu M, Lee S, et al., 2015. BlueDBM: an appliance for big data analytics. Proc ACM/IEEE 42nd Annual Int Symp on Computer Architecture, p.1-13. https://doi.org/10.1145/2749469.2750412

Kang Y, Kee YS, Miller EL, et al., 2013. Enabling cost-effective data processing with smart SSD. Proc IEEE 29th Symp on Mass Storage Systems and Technologies, p.1-12. https://doi.org/10.1109/MSST.2013.6558444

Lee E, Bahn H, Yoo S, et al., 2014. Empirical study of NVM storage: an operating system's perspective and implications. Proc IEEE 22nd Int Symp on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, p.405-410. https://doi.org/10.1109/MASCOTS.2014.56

Li GZ, Chen XG, Chen B, et al., 2016. An FPGA enhanced extensible and parallel query storage system for emerging NVRAM. *IEICE Electron Expr*, 13(4):20151109. https://doi.org/10.1587/elex.13.20151109

Li Z, Wang F, Liu JN, et al., 2016. A user-visible solid-state storage system with software-defined fusion methods for PCM and NAND flash. *J Syst Archit*, 71:44-61. https://doi.org/10.1016/j.sysarc.2016.08.005

Norcott W, Capps D, 2016. IOZONE filesystem benchmark. http://www.iozone.org/ [Accessed on Jan. 23, 2016].

Qiu S, Reddy ALN, 2013. NVMFS: a hybrid file system for improving random write in NAND-flash SSD. Proc IEEE 29th Symp on Mass Storage Systems and Technologies, p.1-5. https://doi.org/10.1109/MSST.2013.6558434

SAMSUNG, 2015. In-storage compute: an ultimate solution for accelerating I/O-intensive applications. http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150813_S301D_Ki.pdf [Accessed on Dec. 11, 2016].

Sha EHM, Chen XZ, Zhuge Q, et al., 2015. Designing an efficient persistent in-memory file system. IEEE Nonvolatile Memory System and Applications Symp, p.1-6. https://doi.org/10.1109/NVMSA.2015.7304365

Sha EHM, Chen XZ, Zhuge Q, et al., 2016. A new design of in-memory file system based on file virtual address framework. *IEEE Trans Comput*, 65(10):2959-2972. https://doi.org/10.1109/TC.2016.2516019

Szalay A, Gray J, 2006. 2020 computing: science in an exponential world. *Nature*, 440(7083):413-414. https://doi.org/10.1038/440413a

Wu XJ, Reddy ALN, 2011. SCMFS: a file system for storage class memory. Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-11. https://doi.org/10.1145/2063384.2063436

XILINX, 2014. Zynq-7000 all programmable SoC technical reference manual. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf [Accessed on Dec. 11, 2016].

Zhou M, Chen XG, Liu Y, et al., 2016. Design and implementation of a random access file system for NVRAM. *IEICE Electron Expr*, 13(4):20151045. https://doi.org/10.1587/elex.13.20151045