ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices

Jiacheng Zhang, Jiwu Shu, and Youyou Lu, ***Tsinghua University***

https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIC ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices

Jiacheng Zhang Jiwu Shu* Youyou Lu
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
zhang-jc13@mails.tsinghua.edu.cn {*shujw, luyouyou*}*@tsinghua.edu.cn*

Abstract

File system designs are undergoing rapid evolution to exploit the potentials of flash memory. However, the internal parallelism, a key feature of flash devices, is hard to be leveraged in the file system level, due to the semantic gap caused by the flash translation layer (FTL). We observe that even flash-optimized file systems have serious garbage collection problems, which lead to significant performance degradation, for write-intensive workloads on multi-channel flash devices.

In this paper, we propose ParaFS to exploit the internal parallelism while ensuring efficient garbage collection. ParaFS is a log-structured file system over a simplified block-level FTL that exposes the physical layout. With the knowledge of device information, ParaFS first proposes 2-D data allocation, to maintain the hot/cold data grouping in flash memory while exploiting channel-level parallelism. ParaFS then coordinates the garbage collection in both FS and FTL levels, to make garbage collection more efficient. In addition, ParaFS schedules read/write/erase requests over multiple channels to achieve consistent performance. Evaluations show that ParaFS effectively improves system performance for write-intensive workloads by 1.6× to 3.1×, compared to the flash-optimized F2FS file system.

1 Introduction

Flash memory has been widely adopted across embedded systems to data centers in the past few years. In the device level, flash devices outperform hard disk drives (HDDs) by orders of magnitude in terms of both latency and bandwidth. In the system level, the latency benefit has been made visible to the software by redesigning the I/O stack [8, 43, 22, 11]. But unfortunately, the bandwidth benefit, which is mostly contributed by the internal parallelism of flash devices [13, 18, 10, 15], is underutilized in system softwares.

Researchers have made great efforts in designing a file system for flash storage. New file system architectures have been proposed, to either improve data allocation performance, by removing redundant functions between FS and FTL [20, 45], or improve flash memory endurance, by using an object-based FTL to facilitate hardware/software co-designs [33]. Features of flash memory have also been leveraged to redesign efficient metadata mechanisms. For instance, the imbalanced read/write feature has been studied to design a persistence-efficient file system directory tree [32]. F2FS, a less aggressive design than the above-mentioned designs, is a flash-optimized file system and has been merged into the Linux kernel [28]. However, the internal parallelism has not been well studied in these file systems.

Unfortunately, internal parallelism is a key design issue to improve file system performance for flash storage. Even though it has been well studied in the FTL level, file systems cannot always gain the benefits. We observe that, even though the flash-optimized F2FS file system outperforms the legacy file system Ext4 when write traffic is light, its performance does not scale well with the internal parallelism when write traffic is heavy (48% of Ext4 in the worst case, more details in Section 4.3 and Figure 6).

After investigating into file systems, we have the following three observations. First, optimizations in both FS and FTL are made but may collide. For example, data pages[1] are grouped into hot/cold groups in some file systems, so as to reduce garbage collection (GC) overhead. FTL stripes data pages over different parallel units (e.g., channels) to achieve parallel performance, but breaks up the hot/cold groups. Second, duplicated log-structured data management in both FS and FTL leads to inefficient garbage collection. FS-level garbage collection is unable

*Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).

[1] In this paper, we use *data pages* instead of *data blocks*, in order not to be confused with *flash blocks*.

to erase physical flash blocks, while FTL-level garbage collection is unable to select the right victim blocks due to the lack of semantics. Third, isolated I/O scheduling in either FS or FTL results in unpredictable I/O latencies, which is a new challenge in storage systems on low-latency flash devices.

To exploit the internal parallelism while keeping low garbage collection overhead, we propose ParaFS, a log-structured file system over simplified block-level FTL. The simplified FTL exposes the device information to the file system. With the knowledge of the physical layout, ParaFS takes the following three approaches to exploit the internal parallelism. First, it fully exploits the channel-level parallelism of flash memory by page-unit striping, while ensuring data grouping physically. Second, it coordinates the garbage collection processes in the FS and FTL levels, to make GC more efficient. Third, it optimizes the request scheduling on read, write and erase requests, and gains more consistent performance. Our major contributions are summarized as follows:

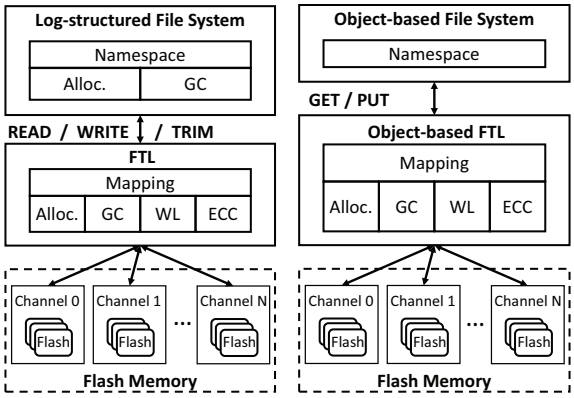
- We observe the internal parallelism of flash devices is underutilized when write traffic is heavy, and propose ParaFS, a parallelism-aware file system over a simplified FTL.
- We propose parallelism-aware mechanisms in the ParaFS file system, to mitigate the parallelism's conflicts with hot/cold grouping, garbage collection, and I/O performance consistency.
- We implement ParaFS in the Linux kernel. Evaluations using various workloads show that ParaFS has higher and more consistent performance with significant lower garbage collection overhead, compared to legacy file systems including the flash-optimized F2FS.

The rest of this paper is organized as follows. Section 2 analyses the parallelism challenges and motivates the ParaFS design. Section 3 describes the ParaFS design, including the 2-D allocation, coordinated garbage collection and parallelism-aware scheduling mechanisms. Evaluations of ParaFS are shown in Section 4. Related work is given in Section 5 and the conclusion is made in Section 6.

2 Motivation

2.1 Flash File System Architectures

Flash devices are seamlessly integrated into legacy storage systems by introducing a flash translation layer (FTL). Though NAND flash has low access latency and high I/O bandwidth, it has several limitations, e.g., erase-before-write and write endurance (i.e., limited program/erase cycles). Log-structured design is supposed to be a friendly way for flash memory. It is adopted in both the file system level (e.g., F2FS [28], and NILFS [27]) and the FTL level [10, 36]. The FTL hides the flash limitations by updating data in a log-structured way using an address mapping table. It exports the same I/O interface as HDDs. With the use of the FTL, legacy file systems can directly run on these flash devices, without any changes. This architecture is shown in Figure 1(a).



(a) Block Interface (b) Object Interface

Figure 1: FS Architectures on Flash Storage

While FTL is good at abstracting the flash memory as a block device, it widens the semantic gap between file systems and flash memory. As shown in Figure 1(a), the FS and FTL run independently without knowing each other's behaviours, resulting in inefficient storage management. Even worse, the two levels have redundant functions, e.g., space allocation, address mapping, and garbage collection, which further induce performance and endurance overhead.

Object-based FTL (OFTL) [33] is a recently proposed architecture to bridge the semantic gap between file systems and FTLs, as shown in Figure 1(b). However, it requires dramatic changes of current I/O stack, and is difficult to be adopted currently, either in the device due to complexity or in the operating system due to rich interfaces required in the drivers.

2.2 Challenges of Internal Parallelism

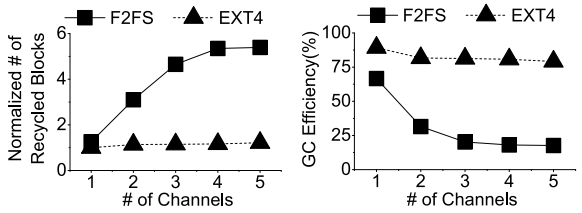
In a flash device, a number of flash chips are connected to the NAND flash controller through multiple channels. I/O requests are distributed to different channels to achieve high parallel performance, and this is known as the *internal parallelism*. Unfortunately, this feature has not been well leveraged in the file systems for the following three challenges.

Hot/Cold Grouping vs. Internal Parallelism. With the use of the FTL, file systems allocate data pages in a one-dimension linear space. The linear space works fine for HDDs, because a hard disk accesses sectors serially due to the mechanical structure, but not for flash devices.

In flash-based storage systems, a fine hot/cold grouping reduces the number of valid pages in victim flash blocks during garbage collection. Some file systems separate data into groups with different hotness for GC efficiency. Meanwhile, the FTL trys to allocate flash pages from different parallel units, aiming at high bandwidth. Data pages that belong to the same group in the file system may be written to different parallel units. Parallel space allocation in FTL breaks hot/cold data groups. As such, the hot/cold data grouping in the file system collides with the internal parallelism in the FTL.

Garbage Collection vs. Internal Parallelism. Log-structured file systems are considered to be flash friendly. However, we observe that F2FS, a flash-optimized log-structured file system, performs worse than Ext4 on multi-channel flash devices when write traffic is heavy. To further analyse this inefficiency, we implement a page-level FTL and collect the GC statistics in the FTL, including the number of erased flash blocks and the percentage of invalid pages in each erased block (i.e., GC efficiency). Details of the page-level FTL and evaluation settings are introduced in Section 4. As shown in Figure 2, for the random update workload in YCSB[14], F2FS has increasing number of erased flash blocks and decreasing GC efficiency when the number of channels increases.

The reason lies in the incoordinate garbage collections in FS and FTL. When the log-structured file system cleans a segment in the FS level, the invalid flash pages are actually scattered over multiple flash parallel units. The more channels the device has, the more diverse the invalid pages are. This degrades the GC efficiency in the FTL. At the same time, the FTL gets pages' invalidation only after receiving the *trim* commands from the log-structured file system, due to the no in-place update. As we observed in the experiments, a large number of pages which are already invalidated in the file system are migrated to the clean flash blocks during the FTL's erase process. Therefore, garbage collections in FS and FTL levels collide and damage performance.



(a) # of Recycled Blocks (b) GC Efficiency

Figure 2: GC Statistics under Heavy Traffic

Consistent Performance vs. Internal Parallelism. Read or write requests may be blocked by erase operations in the FTL. Since erase latency is much higher than read/write latency, this causes latency spikes, making I/O latency unpredictable. It is known as the inconsistent performance, which is a serious issue for low-latency storage devices. Parallel units offer an opportunity to remove latency spikes by carefully scheduling I/O requests. Once a file system controls read, write and erase operations, it can dynamically schedule requests to make performance more consistent.

To address the above-mentioned challenges, we propose a new architecture (as shown in Figure 3) to exploit the internal parallelism in the file system. In this architecture, we use a simplified FTL to expose the physical layout to the file system. Our designed ParaFS maximizes the parallel performance in the file system level while achieving physical hot/cold data grouping, lower garbage collection overhead and more consistent performance.

3 Design

ParaFS is designed to exploit the internal parallelism without compromising other mechanisms. To achieve this goal, ParaFS uses three key techniques:

- *2-Dimension Data Allocation* to stripe data pages over multiple flash channels at page granularity while maintaining hot/cold data separation physically.
- *Coordinated Garbage Collection* to coordinate garbage collections in FS and FTL levels and lower the garbage collection overhead.
- *Parallelism-Aware Scheduling* to schedule the read/write/erase requests to multiple flash channels for more consistent system performance.

In this section, we introduce the ParaFS architecture first, followed by the description of the above-mentioned three techniques.

3.1 The ParaFS Architecture

ParaFS reorganizes the functionalities between the file system and the FTL, as shown in Figure 3. In the ParaFS architecture, ParaFS relies on a simplified FTL (annotated as S-FTL). S-FTL is different from traditional FTLs in three aspects. First, S-FTL uses a static block-level mapping table. Second, it performs garbage collection by simply erasing the victim flash blocks without moving any pages, while the valid pages in the victim blocks are migrated by ParaFS in the FS level. Third, S-FTL exposes the physical layout to the file system using three values, i.e., the number of flash channels[2], the flash page size and the flash block size. These three values are passed to the ParaFS file system through *ioctl* interface,

[2]Currently, ParaFS exploits only the channel-level parallelism, while finer level (e.g., die-level, plane-level) parallelism can be exploited in the FTL by emulating large pages or blocks.

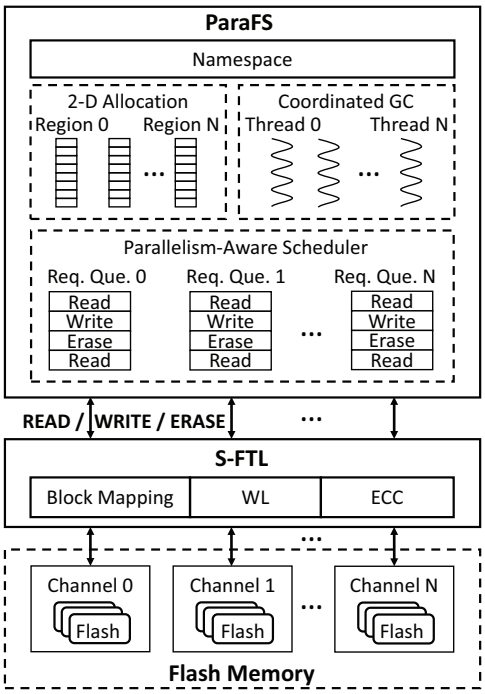


Figure 3: The ParaFS Architecture

when the device is formatted. With the physical layout information, ParaFS manages data allocation, garbage collection, and read/write/erase scheduling in the file system level. The ParaFS file system uses an I/O interface that consists of read/write/erase commands, which remains the same as the legacy block I/O interface. In ParaFS, the erase command reuses the *trim* command in the legacy protocol.

In S-FTL, the block mapping is different from those in legacy block-based FTLs. S-FTL uses static block-level mapping. For normal writes, the ParaFS file system updates data pages in a log-structured way. There is no in-place update in the FS level. For these writes, S-FTL doesn't need to remap the block. S-FTL updates its mapping entries only for wear leveling or bad block remapping. In addition, ParaFS tracks the valid/invalid statuses of pages in each segment, whose address is aligned to the flash block in flash memory. The file system migrates valid pages in the victim segments during garbage collection, and the S-FTL only needs to erase the corresponding flash blocks afterwards. As such, the simplified block mapping, simplified garbage collection and reduced functionalities make S-FTL a lightweight implementation, which has nearly zero overhead.

The ParaFS file system is a log-structured file system. To avoid the "wandering tree" problem [38], file system metadata is updated in place, by introducing a small page-level FTL. Since we focus on the data management, we omit discussions of this page-level FTL for the rest of the paper. In contrast, file system data is updated in a log-structured way and sent to the S-FTL. The ParaFS file system is able to perform more effective data management with the knowledge of the internal physical layout of flash devices. Since the file system is aware of the channel layout, it allocates space in different channels to exploit the internal parallelism and meanwhile keeps the hot/cold data separation physically (details in Section 3.2). ParaFS aligns the segment of log-structured file system to the flash block of the device. Since it knows exactly the valid/invalid statuses of data pages, it performs garbage collection in the file system with the coordination of the FTL erase operations (details in Section 3.3). With the channel information, ParaFS is also able to optimize the scheduling on read/write/erase requests in the file system and make system performance more consistent (details in Section 3.4).

3.2 2-D Data Allocation

The first challenge as described in Section 2.2 is the conflict between data grouping in the file system and internal parallelism in the flash device. Intuitively, the easiest way for file system data groups to be aligned to flash blocks, is using block granularity striping in the FTL. The block striping maintains data grouping while parallelizing data groups to different channels in block units. However, the block-unit striping fails to fully exploit the internal parallelism, since the data within one group needs to be accessed sequentially in the same flash block. We observe in our systems that block-unit striping has significantly lower performance than page-unit striping (i.e., striping in page granularity), especially in the small synchronous write situations, like mail server. Some co-designs employ a super block unit which contains flash blocks from different flash channels [42, 12, 29]. The writes can be striped in a page granularity within the super block and different data groups are maintained in different super blocks. However, the large recycle size of the super block incurs higher overhead in the garbage collection. Therefore, we propose 2-D allocation in ParaFS that uses small allocation units to fully exploit channel-level parallelism while keeping effective data grouping. Tabel 1 summaries the characteristics of these data allocation schemes and compares them with 2-D allocation used in ParaFS.

Figure 4 shows the 2-D allocation in ParaFS. With the device information from S-FTL, ParaFS is able to allocate and recycle space that aligned to the flash memory below. The data space is divided into multiple regions. A region is an abstraction of a flash channel in the device. The number and the size of regions equal to those of flash channels. Each region contains multiple segments and each segments contains multiple data pages. The segment is the unit of allocation and garbage collection

Table 1: Data Allocation Schemes

	Parallelism		Garbage Collecion		
	Stripe Granularity	Parallelism Level	GC Granulariy	Grouping Effect	GC Overhead
Page Stripe	Page	High	Block	No	High
Block Stripe	Block	Low	Block	Yes	Low
Super Block	Page	High	Multiple Blocks	Yes	Medium
2-D Allocation	Page	**High**	Block	Yes	**Low**

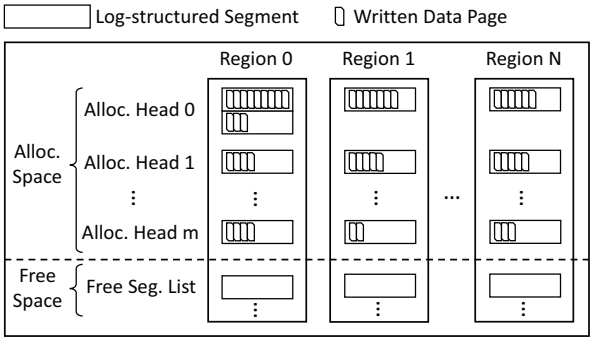


Figure 4: 2-D Allocation in ParaFS

in ParaFS. Its size and address are aligned to the physical flash block. The data page in the segments represents the I/O unit to the flash and is aligned to the flash page. There are multiple allocator heads and a free segment list in each region. The allocator heads point to data groups with different hotness within the region. They are assigned a number of free segments and they allocate free pages to the written data with different hotness, in a log-structured way. The free segments of the region are maintained in the free segment list.

The 2-D allocation consists of channel-level dimension and hotness-level dimension. ParaFS delays space allocation until data persistence. Updated data pages are first buffered in memory. When a write request is going to be dispatched to the device, ParaFS starts the 2-D scheme to allocate free space for the request.

First, in the channel-level dimension, ParaFS divides the write request into pages, and then stripes these pages over different regions. Since the regions match the flash channels one-to-one, the data in the write request is sent to different flash channels in a page-size granularity, so as to exploit the channel-level parallelism in the FS level. After dividing and striping the data of the write request over regions, the allocation process goes to the second dimension.

Second, in the hotness-level dimension, ParaFS groups data pages into groups with different hotness in a region, and sends the divided write requests to their proper groups. There are multiple allocator heads with different hotness in the region. The allocator head, which has similar hotness to the written data, is selected. Finally, the selected allocator head allocates a free page to the written data. The data is sent to the device with the address of the allocated page. Since the segments of allocator heads are aligned to the flash blocks, the data pages with different hotness are assigned to different allocator heads, thereby placed in separated flash blocks. The hotness-level dimension ensures the effective data grouping physically.

In implementation, ParaFS uses six allocators in each region. The data is divided into six kinds of hotness, i.e., hot, warm, and cold, respectively for metadata and regular data. The hotness classification uses the same hints as in F2FS [28]. Each allocator is assigned ten segments each time. The address of the segments and the offset of the allocator heads are recorded in the FS checkpoint in case of system crash.

Crash Recovery. After the crash, legacy log-structured file systems will recover itself to the last checkpoint, which maintains the latest consistent state of the file system. Some file systems[33][28] will do the roll-forward recovery to restore changes after last checkpoint. ParaFS differs slightly during the recovery. Since ParaFS directly accesses and erases flash memory through statically mapped S-FTL, it has to detect the written pages after last checkpoint. These written pages need to be detected so that the file system does not overwrite these pages, which otherwise causes overwrite errors in flash memory.

To solve this problem, ParaFS employs an *update window* similar to that in OFSS [33]. An update window consists of segments from each region that are assigned to allocator heads after last checkpoint. Before a set of segments are assigned to allocator heads, their addresses are recorded first. During recovery, ParaFS first recovers itself to the last checkpoint. Then, ParaFS does the roll-forward recovery with the segments in the update window in each region, since all written pages after last checkpoint fall in this window. After recovering the valid pages in the window, other pages will be considered as invalid, and they will be erased by the GC threads in the future.

3.3 Coordinated Garbage Collection

The second challenge as described in Section 2.2 is the mismatch of the garbage collection processes in FS and FTL levels. Efforts in either side are ineffective in reclaiming free space. ParaFS coordinates garbage collection in the two levels and employs multiple GC threads to leverage the internal parallelism. This tech-

nique improves both the GC efficiency (i.e., the percent of invalid pages in victim blocks) and the space reclaim efficiency (i.e., time used for GC).

The coordinated garbage collection contains the GC process from FS level and FTL level. We use the terms "paraGC" and "flashGC" to distinguish them. In the FS level, the paraGC is triggered when free space drops below a threshold (i.e., foreground paraGC) or file system is idle (i.e., background paraGC). The unit of garbage collection is the segment, as we described in Section 3.2. The foreground paraGC employs greedy algorithm to recycle the segments quickly and minimize the latency of I/O threads. The background paraGC uses cost-benefit algorithm [38] that selects victim segments not only based on the number of invalid data pages, but also their "age". When the paraGC thread is triggered, it first selects victim segments using the algorithms above. Then, the paraGC thread migrates the valid pages in the victim segments to the free space. If the migration is conducted by foreground paraGC, these valid pages are striped to the allocator heads with similar hotness in different regions. If the migration is conducted by background paraGC, these valid pages are considered to be cold, and striped to the allocator heads with low hotness. After the valid pages are written to the device, the victim segments are marked erasable and they will be erased after checkpoint. The background paraGC exits after the migration while the foreground paraGC does the checkpoint and sends the erase requests to S-FTL by *trim*.

In the FTL level, the block recycling is simplified, since the space management and garbage collection are moved from FTL level to FS level. The segments in the FS level match the flash blocks one to one. After paraGC migrates valid pages in the victim segments, the corresponding flash blocks can be erased without additional copies. When S-FTL receives *trim* commands from the ParaFS, flashGC locates the flash blocks that victim segments mapped to, and directly erases them. After the flash blocks are erased, S-FTL informs the ParaFS by the callback function of the requests. The coordinated GC migrates the valid pages in FS level, and invalidates the whole flash block to the S-FTL. No migration overhead is involved during the erase process in the S-FTL.

Coordinated GC also reduces the over-provisioning space of the device and brings more user available capacity. Traditional FTLs need to move the valid pages from the victim blocks before erasing. They keeps large over-provisioning space to reduce the overhead of garbage collection. The spared space in FTL decreases the user visible capacity. Since the valid pages are already moved by ParaFS, the flashGC in S-FTL can directly erase the victim blocks without any migration. S-FTL needn't spare space for garbage collection. The over-provisioning space of S-FTL is much smaller than the traditional ones. The spared blocks are only used for bad block remapping and block mapping table storage. S-FTL also needn't track and maintain the flash page status and flash block utilization, which also reduces the size of spared space.

ParaFS optimizes the foreground GC by employing multiple GC threads. Legacy log-structured file systems perform foreground GC in one thread [28], or none [27]. Since all operations are blocked during checkpointing. Multiple threads cause frequent checkpoint and decrease the performance severely. However, one or less foreground GC thread is not enough under write intensive workloads which consume the free segments quickly. ParaFS assigns one GC thread to each region and employs an additional manager thread. The manager checks the utilization of each region, and wakes up the GC thread of the region when it's necessary. After GC threads migrate the valid data pages, the manager will do the checkpoint, and send the erase requests asynchronously. This optimization avoids multiple checkpoints, and accelerates the segment recycling under heavy writes.

3.4 Parallelism-Aware Scheduling

The third challenge as described in Section 2.2 is the performance spikes. To address this challenge, ParaFS proposes to schedule the read/write/erase requests in the file system level while exploiting the internal parallelism. In ParaFS, the 2-D allocation selects the target flash channels for the write requests, and the coordinated GC manages garbage collection in the FS level. These two techniques offer an opportunity for ParaFS to optimize the scheduling on read, write and erase requests. ParaFS employs parallelism-aware scheduling to provide more consistent performance under heavy write traffic.

Parallelism-aware scheduling consists of request dispatching phase and request scheduling phase. In the dispatching phase, it optimizes the write requests. The scheduler maintains a request queue for each flash channel of the device, shown in Figure 3. The target flash channels of read and erase requests are fixed, while the target channel of writes can be adjusted due to the late allocation. In the channel-level dimension of 2-D allocation, ParaFS splits the write requests into data pages, and selects a target region for each page. The region of ParaFS and the flash channel below are one-to-one correspondence. Instead of a Round-Robin selection, ParaFS selects the region to write, whose corresponding flash channel is the least busy in the scheduler. Due to the asymmetric read and write performance of the flash drive, the scheduler assigns different weights

(W_{read}, W_{write}) to read and write requests in the request queues. The weights are obtained by measuring the corresponding latency of the read and write requests. Since the write latency is 8× of the read latency in our device, the W_{read} and W_{write} are set to 1 and 8 in the evaluation. The channel with the smallest weight calculated by the Formula 1 will be selected. $Size_{read}$ and $Size_{write}$ represent the size of read and write requests in the request queue. The erase requests in the request queue are not considered in the formula, because the time, when they are sent to the device, is uncertain.

$$W_{channel} = \sum(W_{read} \times Size_{read}, W_{write} \times Size_{write}) \quad (1)$$

In the scheduling phase, the scheduler optimizes the erase requests scheduling. Considering the fairness, the parallelism-aware scheduler assigns a time slice for read requests and a same-size slice for write/erase requests. The scheduler works on each request queue individually. In the read slice, the scheduler schedules read requests to the channel. When the slice ends or no read request is left in the queue, the scheduler determines to schedule write or erase request in next slice by Formula 2. The f is the percent of free blocks in the flash channel. N_e is the percent of flash channels that are processing the erase requests at this moment. The a and b represents the weight of these parameters.

$$e = a \times f + b \times N_e \quad (2)$$

If e is higher than 1, the scheduler sends write requests in the time slice. Otherwise, the scheduler sends erase requests in that time slice. After the write/erase slice, the scheduler turns to read slice again. In current implementation, a and b are respectively set to 2 and 1, which implies that erase requests are scheduled only after the free space drops below 50%. This scheme gives higher priority to erase requests when the free space is not enough. Meanwhile, it prevents too many channels erasing at the same time, which helps to ease the performance wave under heavy write traffic.

With the employment of these two optimizations, the parallelism-aware scheduler helps to provide both high and consistent performance.

4 Evaluation

In this section, we evaluate ParaFS to answer the following three questions:

1. How does ParaFS perform compared to other file systems under light write traffic?
2. How does ParaFS perform under heavy write traffic? And, what are the causes behind?
3. What are the benefits respectively from the proposed optimizations in ParaFS?

4.1 Experimental Setup

In the evaluation, we compare ParaFS with Ext4 [2], BtrFS [1] and F2FS [28], which respectively represent the in-place-update, copy-on-write, and flash-optimized log-structured file systems. ParaFS is also compared to a revised F2FS (annotated as F2FS_SB) in addition to conventional F2FS (annotated as F2FS). F2FS organizes data into segment, which is the GC unit in FS and is the same size as flash block. F2FS_SB organizes data into super blocks. A super block consists of multiple adjacent segments. The number of the segments equals to the number of channels in the flash device. In F2FS_SB, a super block is the unit of allocation and garbage collection, and can be accessed in parallel.

We customize a raw flash device to support programmable FTLs. Parameters of the flash device are listed in Table 2. To support the file systems above, we implement a page-level FTL, named PFTL, based on DFTL [17] with lazy indexing technique [33]. PFTL stripes updates over different channels in a page size unit, to fully exploit the internal parallelism. S-FTL is implemented based on PFTL. It removes the allocation function, replaces the page-level mapping with static block-level mapping, and simplifies the GC process, as described in Section 3.1. In both FTLs, the number of flash channels and the capacity of the device are configurable. With the help of these FTLs, we collect the information about flash memory operations, like the number of erase operations and the number of pages migrated during garbage collection. The low-level information is helpful for comprehensive analysis of file system implications.

Table 2: Parameters of the Customized Flash Device

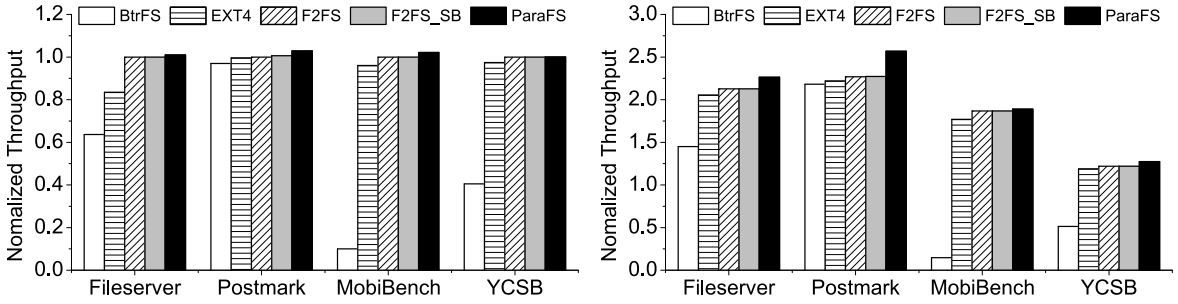
Host Interface	PCIe 2.0 x8
Number of Flash Channel	34
Capacity per Channel	32G
NAND Type	25nm MLC
Page Size	8KB
Block Size	2MB
Read Bandwidth per Channel	49.84 MB/s
Write Bandwidth per Channel	6.55 MB/s

The experiments are conducted on an X86 server with Intel Xeon E5-2620 processor, clocked at 2.10GHz, and 8G memory of 1333MHz. The server runs with Linux kernel 2.6.32, which is required by the customized flash device. For the target system, we back-port F2FS from the 3.15-rc1 main-line kernel to the 2.6.32 kernel. ParaFS is implemented as a kernel module based on F2FS, but differs in data allocation, garbage collection, and I/O scheduling.

Workloads. Table 3 summarizes the four workloads used in the experiments. Two of them run directly on file

Table 3: Workload Characteristics

Name	Description	# of Files	I/O size	Threads	R/W	fsync
Fileserver	File server workload: random read and write files	60,000	1MB	50	33/66	N
Postmark	Mail server workload: create, delete, read and append files	10,000	512B	1	20/80	Y
MobiBench	SQLite workload: random update database records	N/A	4KB	10	1/99	Y
YCSB	MySQL workload: read and update database records	N/A	1KB	50	50/50	Y



(a) channel = 8 (b) channel = 32

Figure 5: Performance Evaluation (Light Write Traffic)

systems and the other two run on databases. Fileserver is a typical pre-defined workload in Filebench [3] to emulate the I/O behaviors in file servers. It creates, deletes and randomly accesses files in multiple threads. Postmark [26] emulates the behavior of mail servers. Transactions, including create, delete, read and append operations, are performed to the file systems. Mobibench [19] is a benchmark tool for measuring the performance of file IO and DB operations. In the evaluation, it issues random update operations to the SQLite[9], which runs with FULL synchronous and WAL journal mode. YCSB [14] is a framework to stress many popular data serving systems. We use the workload-A of YCSB on MySql [7], which consists of 50% random reads and 50% random updates.

4.2 Evaluation with Light Write Traffic

In this section, we compare ParaFS with other file systems under light write traffic. We choose 8 and 32 channels for this evaluation, which respectively represent SATA SSDs [6, 4, 5] and PCIe SSDs [5, 41]. The device capacity is set to 128GB.

Figure 5 shows the throughput of evaluated file systems, and results are normalized against F2FS's performance in 8-channel case. From the figure, we have two observations.

(1) ParaFS outperforms other file systems in all cases, and achieves 13% higher over F2FS for postmark workload in the 32-channel case. In the evaluated file systems, BtrFS performs poorly, especially for database benchmarks that involve frequent syncs. The update propagation (i.e., "wandering tree" problem [38]) of copy-on-write brings intensive data writes in BtrFS during the sync calls (7.2× for mobibench, 3.0× for YCSB). F2FS mitigates this problem by updating the metadata in place [28]. Except BtrFS, other file systems perform roughly similar under light write traffic. F2FS only outperforms Ext4 by 3.5% in fileserver with 32 channels, which is consistent to the F2FS evaluations on PCIe SSD[28]. The performance bottleneck appears to be moved, due to the fast command processing and high random access ability of the PCIe drive. F2FS_SB shows nearly the same performance to F2FS. Since PFTL stripes the requests over all flash channels with a page-size unit, larger allocation unit in F2FS_SB doesn't gain more benefit in parallelism. The GC impact of larger block recycling is also minimized due to the light write pressure. The impact will be seen under heavy write traffic evaluations in Section 4.3. ParaFS uses cooperative designs in both FS and FTL levels, eliminates the duplicate functions, and achieves the highest performance.

(2) Performance gains in ParaFS grow when the number of channels is increased. Comparing the two figures in Figure 5, all file systems have their performance improved when the number of channels is increased. Among them, ParaFS achieves more. It outperforms F2FS averagely by 1.53% in 8-channel cases and 6.29% in 32-channel cases. This also evidences that ParaFS spends more efforts in exploiting the internal parallelism.

In all, ParaFS has comparable or better performance than the other evaluated file systems when the write traffic is light.

4.3 Evaluation with Heavy Write Traffic

Since ParaFS is designed to address the problems of data grouping and garbage collection while exploiting

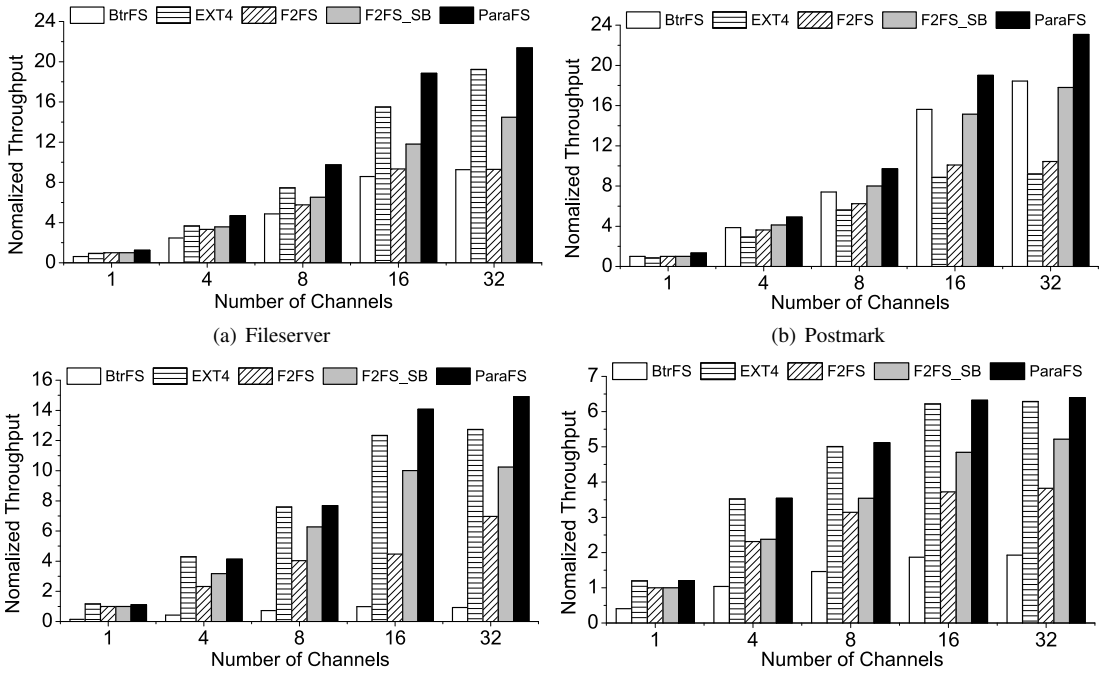


Figure 6: Performance Evaluation (Heavy Write Traffic)

the internal parallelism, evaluations using heavy write traffic are much more important. In this evaluation, we limit the capacity of flash device to 16GB and increase the benchmark sizes. The write traffic sizes in the four evaluated benchmarks are set to $2\times \sim 3\times$ of the flash device capacity, to trigger the garbage collection actions.

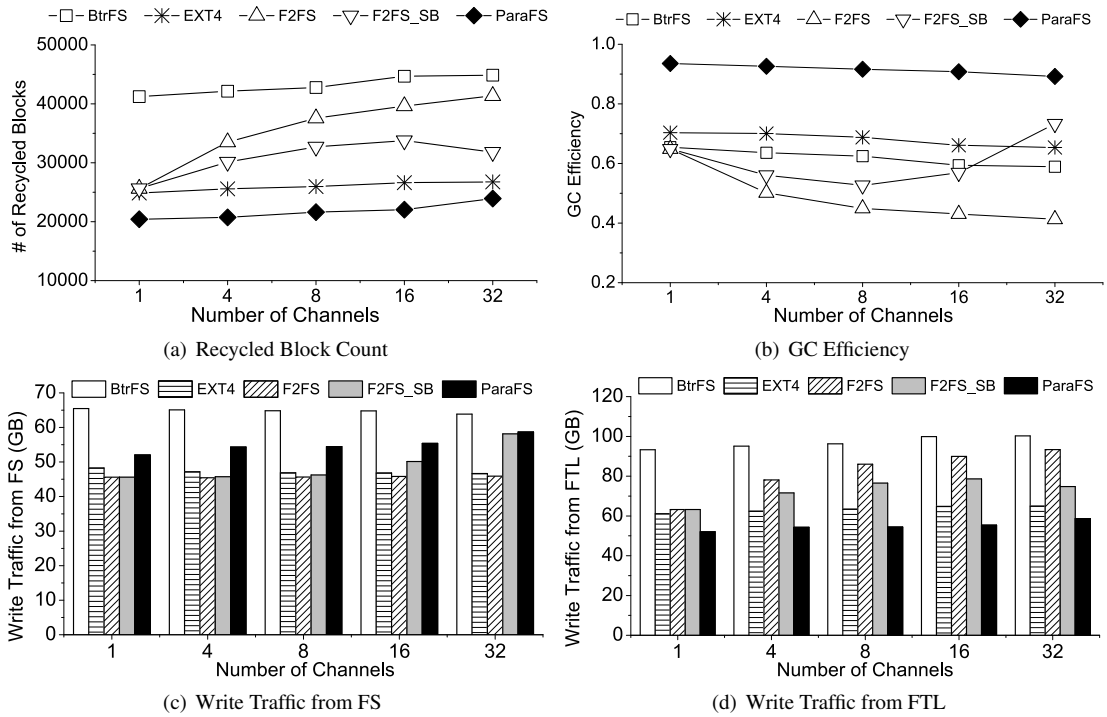
4.3.1 Performance

Figure 6 shows the throughput of evaluated file systems for the heavy write traffic cases, with the number of channels varied from 1 to 32. Results are normalized against F2FS's throughput in 1-channel case. From the figure, we have three observations.

(1) Ext4 outperforms F2FS for three out of the four evaluated workloads, which is different from that in the light write traffic evaluation. The performance gap between Ext4 and F2FS tends to be wider with more flash channels. The reason why the flash-optimized F2FS file system has worse performance than Ext4 is the side effects of internal parallelism. In F2FS, the hot/cold data grouping and the aligned segments are broken when data pages are distributed to multiple channels in the FTL. Also, the invalidation of a page is known in the FTL only after it is recycled in the F2FS, due to the no in-place update. Unfortunately, a lot of invalid pages have been migrated during garbage collection before their statuses are passed to the FTL. Both reasons lead to high GC overhead in FTL, and the problem gets more serious with increased parallelism. In Ext4, the in-place update pattern is more accurate in telling FTLs the page invalidation than F2FS. The exceptional case is the postmark workload, which contains a lot of create and delete operations. Ext4 spreads the inode allocation in all of the block groups for load balance. This causes the invalid pages distributed evenly in the flash blocks and results in higher garbage collection overhead than F2FS (18.5% higher on average). In general, the observation that flash-optimized file system is not good at exploiting internal parallelism under heavy write traffic motivates our ParaFS design.

(2) F2FS_SB shows improved performance than F2FS for the four evaluated workloads, and the improvement grows with more channels. This is also different from results in the light write traffic evaluation. The performance of F2FS improves quickly when the number of channels is increased from 1 to 8, but the improvement is slowed down afterward. For fileserver, postmark and YCSB workloads, F2FS gains little improvement in the 32-channel case over the 16-channel case. The main cause is the increasing GC overhead, which will be seen in next section. In contrast, allocation and recycling in super block units of F2FS_SB ease the GC overhead caused by unaligned segments. The *trim* commands sent by F2FS_SB contain larger address space, and are more effective in telling the invalid pages to the FTL. However, selecting victims in larger units also decreases the GC efficiency. As such, the internal parallelism using super block methods [12, 42] is still not effective.

(3) ParaFS outperforms other file systems in all cases.



(a) Recycled Block Count

(b) GC Efficiency

(c) Write Traffic from FS

(d) Write Traffic from FTL

Figure 7: Garbage Collection and Write Traffic Evaluation (Fileserver)

ParaFS outperforms Ext4 from 1.0× to 1.7× in the 8-channel case, and to 2.5× in the 32-channel case. ParaFS outperforms F2FS from 1.6× to 1.9× in the 8-channel case, and from 1.7× to 3.1× in the 32-channel case. ParaFS outperforms F2FS_SB from 1.2× to 1.5× in both cases. ParaFS keeps the aligned flash block erase units while using page-unit striping in 2-D allocation. The coordinated multi-threaded GC is also helpful in reducing the GC overhead. And thus, ParaFS is effective in exploiting the internal parallelism under heavy write traffic.

4.3.2 Write Traffic and Garbage Collection

To further understand the performance gains in ParaFS, we collect the statistics of garbage collection and write traffic from both FS and FTL levels. We select the fileserver workload as an example due to space limitation. The other workloads have similar patterns and are omitted. For fairness, we revise the fileserver benchmark to write fixed-size data. In the evaluation, the write traffic size in fileserver is set to 48GB, and the device capacity is 16GB.

Figure 7(a) and Figure 7(b) respectively give the recycled block count and the garbage collection efficiency of evaluated file systems with varied number of flash channels. The recycled block count is the number of flash blocks that are erased in flash memory. The GC efficiency is measured using the average percentage of invalid pages in a victim flash block.

ParaFS has the lowest garbage collection overhead and highest GC efficiency among all evaluated file systems under four benchmarks. For the fileserver evaluation, it achieves the lowest recycled block count (62.5% of F2FS on average) and the highest GC efficiency (91.3% on average). As the number of channels increases, the number of recycled blocks in F2FS increases quickly. This is due to the unaligned segments and uncoordinated GC processes of both sides (as analyzed in Section 4.3.1). It is also explained with the GC efficiency degradation in Figure 7(b). The GC efficiency of Ext4, BtrFS, ParaFS trends to drop a little with more flash channels. Because the adjacent pages are more scattered when the device internal parallelism increases, and they tend to be invalidated together. F2FS_SB acts different from other file systems. In F2FS_SB, when the number of channels is increased from 8 to 32, the number of recycled blocks decreases and the GC efficiency increases. The reason is that the super block has better data grouping and alignments, and this advantage becomes increasingly evident with higher degree of parallelism. F2FS_SB also triggers FS-level GC threads more frequently with larger allocation and GC unit. More *trim* commands with larger address space help to decrease the number of invalid pages migrated by GC process in the FTL level. ParaFS further utilizes fine-grained data grouping and GC unit, and has the lowest garbage collection overhead.

Figure 7(c) shows the write traffic that file systems write to FTLs. Figure 7(d) shows the write traffic

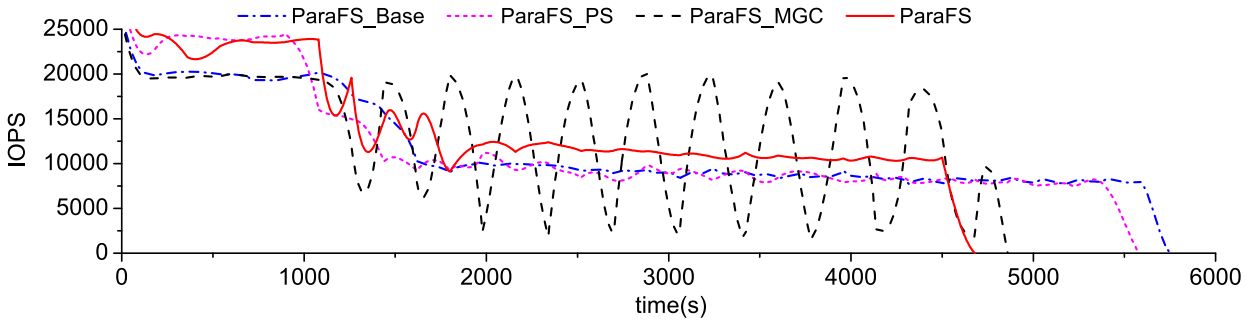


Figure 8: Performance Consistency Evaluation

that FTLs write to the flash memory. Write traffic in either level comes from not only the file system data and metadata but also the page migrated during garbage collection in FS or FTL level. The write traffic from FS in Ext4, BtrFS and F2FS are stable for different parallelism levels, because the increase of the flash channels in the device is transparent to them. ParaFS writes more in the file system than Ext4, F2FS and F2FS_SB, but less than them in the FTL. Because all the page migration during the garbage collection is done in the FS level. Similarly, F2FS_SB has higher write traffic from FS and lower from FTL than F2FS, as the number of channels is increased from 8 to 32. This results from the improved GC efficiency as mentioned above. The FTL write traffic in F2FS is higher than F2FS_SB, which explains why F2FS_SB has better performance than F2FS in Figure 6. ParaFS coordinates garbage collections in the two levels, and is more effective in space recycling. Compared to F2FS, ParaFS decreases the write traffic to the flash memory by 31.7% $\sim$ 54.7% in 8-channel case, and 37.1% $\sim$ 58.1% in 32-channel case. Compared to F2FS_SB, ParaFS decreases it by 14.9% $\sim$ 48.4% in 8-channel case, and 15.7% $\sim$ 32.5% in 32-channel case.

4.4 Performance Consistency Evaluation

To evaluate the performance consistency in ParaFS, we monitor the throughput wave during each run of experiments. ParaFS aims at more consistent performance using multi-threaded GC and parallelism-aware scheduling. In this evaluation, we use four versions of ParaFS. The baseline (annotated as ParaFS_Base) is the ParaFS version without the above-mentioned two optimizations. ParaFS_PS and ParaFS_MGC respectively stand for the version with parallelism-aware scheduling and multi-threaded GC. ParaFS is the fully-functioned version.

The fileserver and postmark have different phases in the evaluation, which also cause fluctuation in the aggregate throughput (in terms of IOPS). We choose mobibench as the candidate for performance consistency evaluation. Mobibench performs random asynchronous reads and writes to pre-allocated files. In this evaluation, the write traffic of mobibench is set to be 2$\times$ of the device capacity.

Figure 8 shows the process of each run using four versions of ParaFS. We compare ParaFS_MGC and ParaFS_Base to analyse the impact of multi-threaded GC. ParaFS_MGC and ParaFS_Base have similar performance in the first 1000 seconds, during which no garbage collection is involved. After that, the performance of ParaFS_MGC waves. The performance peaks appear after the GC starts in multiple threads. ParaFS_MGC finishes the experiments earlier than ParaFS_Base by 18.5%.

The parallelism-aware scheduling contains two major methods, the write request dispatching and the erase request scheduling. The effectiveness of write request dispatching can be seen by comparing ParaFS_PS and ParaFS_Base. For the first 1000 seconds when there is no garbage collection, ParaFS_PS outperforms ParaFS_Base by nearly 20%. This benefit comes from the write dispatching in the parallelism-aware scheduling technique, which allocates pages and sends requests to the least busy channels. The effectiveness of erase request scheduling can be observed between ParaFS and ParaFS_MGC. In the latter part of each run when the GC processes are frequently triggered, ParaFS using parallelism-aware scheduling performs more consistently than ParaFS_MGC.

In conclusion, the FS-level multi-threaded garbage collection as implemented in ParaFS is more effective in reclaiming free space, and the FS-level parallelism-aware scheduling makes performance more consistent.

5 Related Work

Data Allocation. In the FTL or flash controller, internal parallelism has been extensively studied. Recent researches [13, 18, 21] have conducted extensive experiments on page allocation with different levels (i.e., channel-level, chip-level, die-level and plane-level) of internal parallelism. Gordon [12] introduces a 2-D striping to leverage both channel-level and die-level parallelism.

But note that, 2-D striping in Gordon is different from 2-D data allocation in ParaFS. 2-D striping is designed in the flash controller, which places data to leverage multiple levels of parallelism. 2-D data allocation is designed in file system, which organizes data into different groups using metrics of parallelism and hotness. In addition, aggressive parallelism in the device level scatters data addresses, breaking up the data organization in the system level. P-OFTL [42] has pointed out this problem and found that increased parallelism leads to higher garbage overhead, which in turn can decrease the overall performance.

In the file system level, DirectFS [20] and Nameless Write [45] propose to remove data allocation functions from file systems and leverage the data allocations in the FTL or storage device, which can have better decisions with detailed knowledge of hardware internals. OFSS [33] proposes an object-based FTL, which enables hardware/software codesign with both knowledge of file semantics and hardware internals. However, these file systems pay little attention to internal parallelism, which is the focus of ParaFS in this paper.

Garbage Collection. Garbage collection has a strong impact on system performance for log-structured designs. Researchers are trying to pass more file semantics to FTLs to improve GC efficiency. For instance, *trim* is a useful interface to inform FTLs the data invalidation, in order to reduce GC overhead in migrating invalid pages. Also, Kang et al. [25] found that FTLs can have more efficient hot/cold data grouping, which further reduces GC overhead, if the expected lifetime of written data is passed from file systems to FTLs. In addition, Yang et al. [44] found log-structured designs in both levels of system software and FTLs have semantic gaps, which make garbage collection in both levels inefficient. In contrast, ParaFS proposes to bridge the semantic gap and coordinate garbage collection in the two levels.

In the file system level, SFS [34] uses a sophisticated hot/cold data grouping algorithm using both access count and age of the block. F2FS [28] uses a static data grouping method according to the file and data types. However, these grouping algorithms suffer when grouped data are spread out in the FTL. Our proposed ParaFS aims to solve this problem and keep physical hot/cold grouping while exploiting the internal parallelism.

A series of research works use large write block size to align the flash block and decrease the GC overhead[40, 31, 30, 35]. RIPQ[40] and Pannier[31] aggregate small random writes in memory, divide them into groups according to the hotness, and evict the groups in flash block size. Nitro[30] deduplicates and compresses the writes in RAM and evicts them in flash block size. Nitro proposes to modify the FTL to support block-unit striping that ensures the effective of the block-size write optimization. SDF[35] employs block-unit striping which is tightly coupled with key-value workloads. ParaFS uses page-size I/O unit and aims at file system workloads.

I/O Scheduling. In flash storage, new I/O scheduling policies have been proposed to improve utilization of internal parallelism [24, 23, 16] or fairness [37, 39]. These scheduling policies are designed in the controller, the FTL or the block layer. In these levels, addresses of requests are determined. In comparison, system-level scheduling can schedule write requests before data allocation, which is flexible.

LOCS [41] is a key-value store that schedules I/O requests in the system level upon open-channel SSDs. With the use of log-structured merge tree (LSM-tree), data is organized into data blocks aligned to flash blocks. LOCS schedules the read, write and erase operations to minimize the response time.

Our proposed ParaFS is a file system that schedules I/O requests in the system level. Different from key-value stores, file systems have irregular reads and writes. ParaFS exploits the channel-level parallelism with page-unit striping. Moreover, its goal is in making performance more consistent.

6 Conclusion

ParaFS is effective in exploiting the internal parallelism of flash storage, while keeping physical hot/cold data grouping and low garbage collection overhead. It also takes the parallelism opportunity to schedule read, write and erase requests to make system performance more consistent. ParaFS's design relies on flash devices with customized FTL that exposes physical layout, which can be represented by three values. The proposed design bridges the semantic gap between file systems and FTLs, by simplifying FTL and coordinating functions of the two levels. We implement ParaFS on a customized flash device. Evaluations show that ParaFS outperforms the flash-optimized F2FS by up to 3.1×, and has more consistent performance.

Acknowledgments

We thank our shepherd Haryadi Gunawi and anonymous reviewers for their feedbacks and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008, 61502266), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the China Postdoctoral Science Foundation (Grant No. 2015M580098).

References

[1] Btrfs. http://btrfs.wiki.kernel.org.

[2] Ext4. https://ext4.wiki.kernel.org/.

[3] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench.

[4] Intel dc s3500 480gb enterprise ssd review. http://www.tweaktown.com/reviews/5534/intel-dc-s3500-480gb-enterprise-ssd-review/index.html.

[5] Intel ssd 750 pcie ssd review. http://www.anandtech.com/show/9090/intel-ssd-750-pcie-ssd-review-nvme-for-the-client.

[6] Intel x25-m and x18-m mainstream sata solid-state drives. ftp://download.intel.com/newsroom/kits/ssd/pdfs/X25-M_34nm_ProductBrief.pdf.

[7] Mysql. https://www.mysql.com/.

[8] FusionIO Virtual Storage Layer. http://www.fusionio.com/products/vsl, 2013.

[9] SQLite. http://www.sqlite.org/, 2014.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC)*, Berkeley, CA, 2008. USENIX.

[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, page 22. ACM, 2013.

[12] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–228, New York, NY, USA, 2009. ACM.

[13] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277. IEEE, 2011.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[15] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[16] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[17] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, New York, NY, USA, 2009. ACM.

[18] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 96–107. ACM, 2011.

[19] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won. Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace. *Future Internet*, 5(4):591–610, 2013.

[20] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2010. USENIX.

[21] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, pages 9–9. USENIX Association, 2012.

[22] M. Jung and M. Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the fifteenth international joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 203–216. ACM, 2013.

[23] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 524–535. IEEE, 2014.

[24] M. Jung, E. H. Wilson III, and M. Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *Proceedings of the 39th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2012.

[25] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, pages 13–13. USENIX Association, 2014.

[26] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[27] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[28] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2015. USENIX.

[29] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[30] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 501–512, Philadelphia, PA, June 2014. USENIX Association.

[31] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, Vancouver, Canada, 2015. ACM.

[32] Y. Lu, J. Shu, and W. Wang. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 75–88, Berkeley, CA, 2014. USENIX.

[33] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2013. USENIX.

[34] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[35] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484, New York, NY, USA, 2014. ACM.

[36] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 301–311. IEEE, 2011.

[37] S. Park and K. Shen. Fios: a fair, efficient flash i/o scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[39] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 67–78, Berkeley, CA, 2013. USENIX.

[40] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[41] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, pages 16:1–16:14, New York, NY, USA, 2014. ACM.

[42] W. Wang, Y. Lu, and J. Shu. p-OFTL: an object-based semantic-aware parallel flash translation layer. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, page 157. European Design and Automation Association, 2014.

[43] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[44] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Dont stack your log on my log. In *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[45] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.