PipeSSD: A Lock-free Pipelined SSD Firmware Design for Multi-core Architecture

DU Zelin
The Chinese University of Hong Kong
Hong Kong, China
zldu22@cse.cuhk.edu.hk

LI Shaoqi
Shenzhen University
Shenzhen, Guangdong, China
lishaoqi2023@email.szu.edu.cn

HUANG Zixuan,
XUE Jin
The Chinese University of Hong Kong
Hong Kong, China
{zxhuang,jinxue}@cse.cuhk.edu.hk

HUANG Kecheng
The Chinese University of Hong Kong
Hong Kong, China
kchuang21@cse.cuhk.edu.hk

WANG Tianyu*
The Chinese University of Hong Kong
Hong Kong, China
tywang@cse.cuhk.edu.hk

SHAO Zili
The Chinese University of Hong Kong
Hong Kong, China
shao@cse.cuhk.edu.hk

ABSTRACT

Modern SSD firmware is continuously optimized for higher parallelism to match the growing frontend PCIe bandwidth with more backend flash channels. Although a multi-core microprocessor is typically adopted to concurrently process independent NVMe requests from multiple NVMe queues, the existing one-to-many thread-request mapping model with each thread serving one or more incoming I/O requests has poor scalability due to severe lock contention problem, especially in cache management.

In this paper, we first conduct experiments on an open-channel NVMe SSD to exhibit the lock contention problem in the one-to-many thread-request mapping model when multiple threads compete for the cacheline, which occupies more than 50% of the long-tail request latency. To mitigate this, we propose PipeSSD, a lock-free pipeline-based SSD firmware design with a many-to-one thread-request mapping model that assigns multiple threads to serve different stages of each I/O request in a pipelined way. Three novel designs are introduced in PipeSSD: 1) a loop-free request processing pipeline with a postponed cache stage to reduce cacheline competition; 2) a lock-free cache management scheme to ensure consistent cache behavior; and 3) a FIFO-based inter-stage request flow for correct request dependencies. We implement PipeSSD on real hardware and evaluate its performance on a multi-core NVMe SSD prototype. The evaluation results show that PipeSSD has a significant throughput improvement compared to the state-of-the-art multi-core SSD firmware.

KEYWORDS

SSD Firmware, Flash Translation Layer, Multi-core Architecture, Pipeline Design

*Corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657384

ACM Reference Format:
DU Zelin, LI Shaoqi, HUANG Zixuan,, XUE Jin, HUANG Kecheng, WANG Tianyu, and SHAO Zili. 2024. PipeSSD: A Lock-free Pipelined SSD Firmware Design for Multi-core Architecture. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657384

1 INRODUCTION

The NVMe protocol with multiple request queues fully exploits the potential of the high-speed PCIe interface [9]. To catch up with the growing frontend PCIe bandwidth, NVMe SSDs incorporate multiple backend flash channels [3, 6] to achieve higher throughput. However, existing SSD firmware inefficiently utilizes valuable computation resources with a one-to-many thread-request mapping model in the multi-core SSD controller, a pivotal component bridging the gap between the growing frontend PCIe bandwidth and the increasing number of backend flash channels.

With each thread handling one or more incoming I/O requests, the SSD controller can readily incorporate additional cores to enhance thread-level parallelism. Although multiple NVMe requests are supposed to be concurrently processed with the multi-core SSD controller, we observe a significant lock contention issue caused by cacheline competition. This issue results in over 50% of the computation resource wastage and hinders SSD firmware's scalability.

To mitigate lock contention, we propose **PipeSSD**, a lock-free pipelined SSD firmware on a multi-core controller. Different from the traditional one-to-many thread-request mapping model, we introduce a many-to-one thread-request mapping model by segmenting the process of handling each NVMe request into four stages (Request Fetching, Cache Management, Flash Translation Layer, Flash Interface Layer) and distributing them across different cores. Although this approach allows the four stages to be processed in a pipeline, it still faces several challenging issues.

The first challenge is how to construct a loop-free pipeline. Simply following the aforementioned request processing sequence still encounters a loop inside the pipeline, which is due to the cacheline update. Specifically, when a request triggers a cacheline miss, a lock is necessary to ensure data consistency until the related cacheline is updated with the correct data, blocking all subsequent requests that need to access this cacheline. If it is a read miss, it needs to re-enter the cache management stage after completing the FIL stage

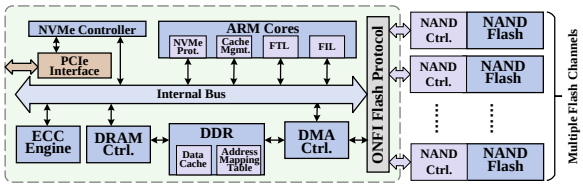


Figure 1: The overall architecture of an NVMe SSD.

(i.e., reading the data from flash chips), introducing a loop in the pipeline. To address this, we counter-intuitively reorganize pipeline stages by **deferring the cache management stage** to the final step. With such a request processing sequence (Request Fetching → FTL → FIL → Cache Management), all cache modifications are postponed to the final stage, which fundamentally eliminates loops inside the pipeline. Without pipeline loops, we can safely remove cacheline locks since all cacheline modifications are only from the preceding pipeline stage, which can be processed in sequence.

The second challenge related to the cache consistency issue arises after postponing the cache management stage. Once a request arrives and checks the cache, the result (hit/miss) may change because previous requests must reach the cache management stage earlier than this request. This can modify the target cacheline and introduce a false cache state for this request. To tackle this issue, we assign a pilot of the cache manager at the beginning to gather the cache lines' state and provide each request with a roadbook. The roadbook records the critical information of the target cache line, such as the cache hit condition and the *Tag*. Each request proceeds the pipeline with its roadbook. At the FTL stage, the necessary physical page number to write back the dirty cacheline is prepared and recorded in the roadbook, as well as the missed page number that needs to be read from flash channels. During the FIL stage, each request can compare the cacheline *Tag* in the roadbook with the **actual** cacheline *Tag* to ensure the correct page is being written back. With pilot assistance, we can achieve lock-free cache management with consistent cache behavior.

The third challenge is to ensure correct request dependencies. We achieve this by adopting FIFO queues between cores. Thus, communications between the four pipeline stages inherently follow a sequenced manner, which simplifies dependency maintenance.

We implement PipeSSD on a real hardware platform with a 4-core ARM CPU, 4GB DDR4 DRAM, and 512GB NAND flash chips. Experimental results show that PipeSSD outperforms the state-of-the-art multi-core SSD firmware and achieves up to 46% higher throughput than the one-thread-multi-request model.

2 BACKGROUND AND MOTIVATION

2.1 NVMe SSD Architecture

As shown in 1, a SSD has computation and memory resources to manage the storage space that is composed of many NAND flash chips. The flash chips are grouped in multiple parallel channels. The NVMe protocol is adopted to fully utilize the high bandwidth PCIe link. With increasing PCIe bandwidth (e.g., 2GB/s @ PCIe Gen2 x4 to 16GB/s @ PCIe Gen5 x4), SSDs employ more flash channels (e.g., from 2 to 16 or more) to increase the I/O parallelism to catch up with the high-speed frontend interface. The bandwidth increment places high demands on the power of the processors, especially the ability to process tasks in parallel.

2.2 NVMe Request Workflow

As shown in Figure 2, host interface layer (HIL), flash translation layer (FTL), and flash interface layer (FIL) are integrated into the SSD's firmware. The NVMe request is fetched from the host memory to the SSD internal memory by the HIL. Then the FTL brings the request from the logical page space to the physical page space. In the physical space, the FIL performs the flash I/O transaction. Finally, the HIL notifies the host that the request has been processed.

HIL. The NVMe communication is based on the NVMe queue pairs. Each queue pair consists of a submission queue (SQ) and a completion queue (CQ). The number of queue pairs normally equals the CPU cores of the host and can be scaled up to 64K. The SSD fetches requests from the SQ. The original request that covers a lot of pages should be segmented into a series of page-level sub-requests. when all sub-requests have been processed, a completion entry (Ce) is posted to the CQ to notify the host.

FTL. The out-of-place update feature of the flash arrays requires the isolation between the logical page address (LPA) and the physical page address (PPA). The FTL has an address allocation scheme [5] to assign physical addresses for write requests and maintains a translation table [4] to store the mappings. In addition, the storage space held by invalid physical addresses should be reclaimed by garbage collection [11]. The latency of flash transactions is two orders of magnitude higher than memory accesses. The efficiency of the data cache significantly improves the performance.

Limited by the memory capacity, sometimes the FTL just caches a portion of the mapping table in memory. With advances in address mapping technology, such as the learned index methods [8], the entire mapping table can be kept in memory. In our model, we load the entire mapping table into memory, so that the flash transactions are only used to manipulate user data.

FIL. The FIL dispatches the flash transactions to the channels according to their physical page addresses. A flash transaction is completed by the flash controller issuing a series of commands to the flash chip. The flash command sequence includes three phases – address setup(SET), data movement(DATA), and command execution(EXE). As shown in Table 1, the latency of DATA and EXE is long. If one dedicated processor polls all flash channels, when a channel is occupied by such long-latency instructions, it schedules the next channel to keep all channels busy.

2.3 Convoy Effect

Based on the SSD architecture and NVMe request workflow introduced above, we implemented a multi-threaded parallel firmware on our FPGA-based platform [10]. The platform's basic information is introduced in Section 4.1, which has 8 flash channels and supports 16 working threads. To increase the efficiency of multi-channel polling, FIL operations are performed with a dedicated thread. Once an NVMe request comes, an idle worker will be woken up to process it. The flash transactions generated in the FTL phase are submitted to the FIL thread through the inter-thread queue, and the FIL thread wakes up the corresponding work thread after processing the transaction. The worker will yield once it fails

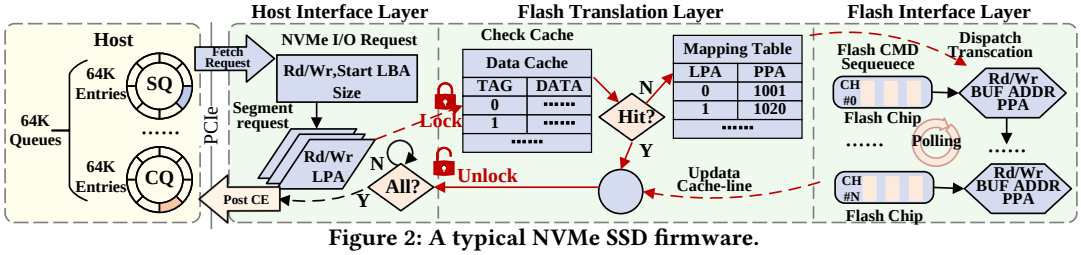
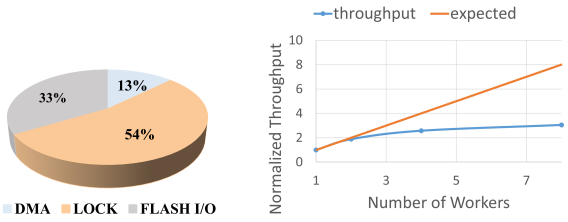


Figure 2: A typical NVMe SSD firmware.



(a) Lock contention. (b) Scalability of multiple workers.

Figure 3: Motivation: lock contention & scalability.

to acquire a cacheline's lock. One of the workers waiting for a lock will be woken up when the lock is released by its holder. We can analyze the bottleneck by counting how long each worker waits for various events. We perform the read-write mixed workload to collect all workers' time consumption information. The test contains 1000 cycles, in each cycle the host sends NVMe requests to read 10MB of data randomly and then writes 10MB of data. The result is shown in Figure 3(a).

Our prototyping device as a verification platform needs to support flexible software and hardware co-design, hence it makes trade-offs in performance and flexibility. To explore the behavior of the internal tasks when the bandwidth of NVMe-PCIe is fully utilized, we build up an emulator of our prototype. The emulated prototype is built on Pthread library [7], we bind each worker to a thread and assign multiple threads (one thread per flash channel) to perform the FIL. By scaling the number of workers, as Figure 3(b) shows, we can observe that the throughput is not growing as expected.

The availability of independent tasks limits task-level parallelism. For instance, in a system with four workers, one worker A waits for a long latency response and another worker B waits for the resources occupied by A. The parallelism of the system at that moment will be halved, i.e., only the remaining two workers are available to process the task. Since the latency of flash I/O transactions (hundreds of microseconds) is at least three orders of magnitude higher than accessing memory (tens of nanoseconds), a worker occupying a cache lock that is waiting for a FIL response will block all subsequent attempts to acquire the lock. The convoy effect that the entire system slows down owing to a few slower tasks is ubiquitous inside the SSDs. SSD's storage space is very huge (up to tens of TB), but its data cache is generally only 0.1% of the storage space (up to several GBs). Each cacheline must correspond to a wide address space so that the probability of multiple requests competing for the same cacheline is high. The contention on the cacheline is exacerbated if the granularity of the lock contains a set of cachelines. The other fact is that the worker occupying the cacheline may be waiting for the FIL responses for much longer than expected. Sometimes, one cache replacement operation requires two sequential flash transactions, the first one is to write back the dirty page and the second one is to read the requested data. Even with the state-of-the-art multi-core SSD firmware [12], the cacheline competition still exists and occupies a lot of computation resources.

3 DESIGN

Conventional SSD firmware employing the one-to-many thread-request mapping model suffers from the convoy effect, as each thread is fully functional and can exclusively occupy the shared resources. Another thread-request mapping model is many-to-one, which segments the processing of one request into several sequential stages and each thread executes one stage. Ideally, each stage execution would only need to access its local state, and requests are transferred between stages through the queues. The pipeline execution can realize the parallelism of the many-to-one mapping model, and the pipeline parallelism is limited only by inter-stage dependencies. Our goal is to design a pipeline with the minimum inter-stage dependencies, i.e. no loop and lock synchronization. The most challenging issue is solving the existence of multiple writers of the data cache, ensuring that the state of the data cache is updated by only one stage.

3.1 Non-blocking Pipeline

There are two data sources stored in the data cache: one is from the host, while the other comes from the flash channels. If we set the update phase of the cacheline before the FIL stage, there must be a loop back from the FIL stage to the update phase for data coming from the flash channel. This straightforward design presents two issues. Firstly, the loop in the pipeline implies inter-stage dependencies which will undermine the parallelism. Secondly, an updating cacheline blocks subsequent requests, especially preventing them from entering the FIL stage, which limits the utilization of parallelism in the multi-channel architecture.

We present PipeSSD to achieve our goal by post-placing the data cache to the final stage of the pipeline, which is a counter-intuitive but effective pipeline design to reduce lock contention. Specifically, the FTL stage and FIL stage can not get the cacheline state of the requests to determine their behavior. The post-placing data cache is essential for discarding the loops inside the pipeline. Besides, the above challenge can be overcome by our techniques including the pilot of the data cache and the data consistency scheduling.

The workflow of PipeSSD is shown in Figure 4. We segment the layered firmware into 4 stages. The functions of each stage are briefly described as follows:

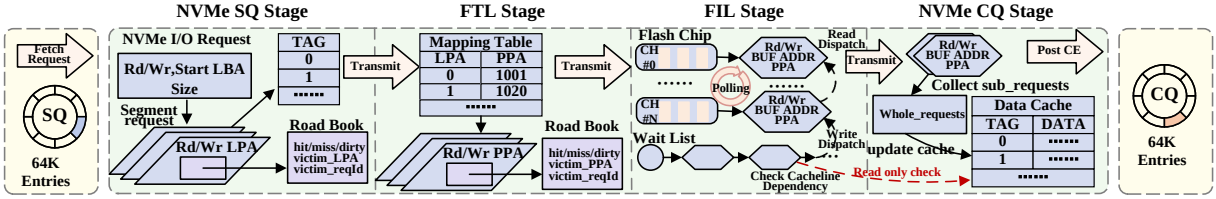


Figure 4: The pipeline design details in PipeSSD.

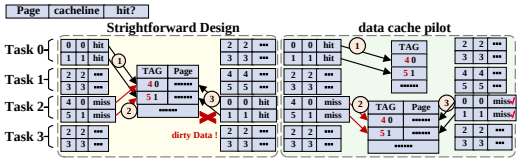


Figure 5: A Pilot example.

- NVMe Fetch stage fetches the NVMe requests from the host's SQs, segments each request into a series of page-level sub-requests, and gets the cacheline state for each sub-requests from the data cache pilot.
- FTL stage makes the necessary address translation, and submits the flash transactions for each sub-request.
- FIL stage submits the flash transactions to flash chips and schedules multiple channels to fully utilize the parallelism.
- The last stage is the NVMe Post stage by which the data needed to complete the request is ready. It updates the cacheline and sends the response.

The next part introduces the techniques in our pipeline design. With these techniques, our pipeline ensures data consistency without locking mechanisms and consistently maintains parallelism without encountering blocking issues.

3.2 Data Consistency without Lock

Data consistency is guaranteed by two paths, one is the correctness of the control path and the other is the sequentiality of the data path. We first address the consistency problem on the control path. The control path is mainly managed by the FTL which translates the LPA to PPA to bridge the gap between two address spaces. The requests with different cache statuses take distinct control paths. Physical addresses are only necessary for cache miss requests. For read requests, the physical address of the requested data is required, while for dirty cachelines, preparation of the victim data page's physical address is needed. Hence, before address translation, we need to confirm the cacheline status of the requests.

In our design, we get the state of the cacheline at the NVMe Fetch stage before the FTL stage. As shown in Figure 5, conducting a direct read-only check of the post-placed data cache tags would destroy data consistency, since previous requests can modify the target cacheline and introduce a false cache state for this request. For instance, Task0 (NVMe SQ) checks the tags of the cachelines for two requests. When the check is done, but before these requests reach Task2(FIL), Task2 finishes two flash transactions then modify the cacheline. Consequently, when these two requests arrive at the FIL stage, their cache status should be missed. However, their control path traveling along the hit state will eventually get dirty data from the cache.

The post-placed data cache can only reflect the local information of the last stage, the requests within the intermediate stages have not yet generated the footprint on the trace of the corresponding cacheline. For each request, if it can see the footprint of the previous request on its corresponding cacheline before accessing the data cache and it is guaranteed to follow the previous footprint into the cacheline, then the data consistency can be preserved.

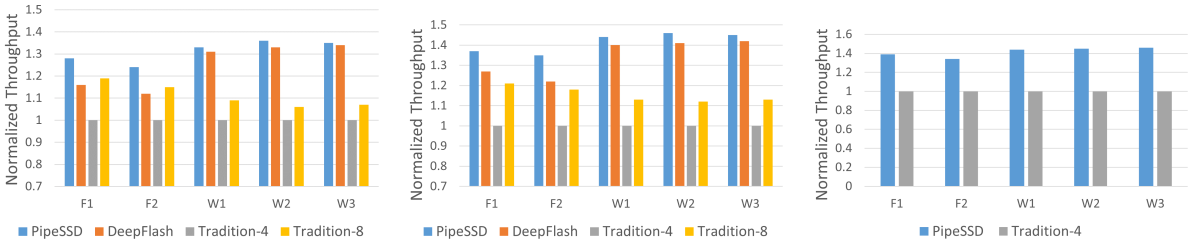
3.2.1 Pilot. We set a pilot of the data cache at the first stage to track the footprint of each cacheline. After segmenting the NVMe request into page-level sub-requests, the pilot assigns each sub-request a roadbook that records the cache state (e.g. hit/miss/dirty) and the previous request ID (i.e. footprint) associated with its corresponding cacheline. For a given sub-request, the pilot follows the cacheline mapping method to get its target cacheline and then compares the LPA of the sub-request with the LPA of the cacheline. If the two LPAs match, the cache state of the sub-request is classified as a hit; Otherwise, the state is missed. The cache state will be recorded in the roadbook to travel with the sub-request through the pipeline. In addition, the roadbook stores a copy of the pilot line. Once the roadbook is established, we update the pilot's information by storing the LPA, ID, and dirty bit (data from the host) of the current sub-request into the target pilot line.

The FTL stage and FIL stage refer to the cache state to handle each sub-request. As shown in Figure 4 The FTL stage checks the address mapping table to translate the requested LPA to PPA for the read-miss sub-requests and assigns a new PPA for the dirty page. The LPA of the victim cacheline can be found in the copy of the pilot line. If necessary, the FIL stage dispatches the sub-request to the flash channel according to the victim's PPA to write back the dirty page. Then the sub-request may be dispatched to the flash channel according to its PPA to read the requested page.

3.2.2 Data consistency scheduling. On the data path, two critical operations impacting data consistency are the write-back operation and the cacheline updating. We integrate the data consistency scheduling to keep the data consistent on the data path.

At the FIL stage, the write-back operation aims to flush the dirty cacheline, requiring accurate input to ensure persistence. Meanwhile, at the NVMe Post stage, the cacheline update operation involves overwriting the cacheline with the newly arrived data. To guarantee that the overwritten data is invalidated before rewriting, we introduce a waitlist at the FIL stage, ensuring that data operations obey the above constraints.

Throughout the pipeline, the data stream follows a First-In, First-Out (FIFO) pattern, except for the FIL stage, which handles multiple data streams concurrently via multiple flash channels. In our design, the requests from the FTL stage are not directly dispatched to the flash channels, but first go to the waitlist. During the scheduling



(a) 4 flash channels. (b) 8 flash channels. (c) 4 flash channels & half-sized data cache.

Figure 6: Performance comparison between PipeSSD and the state-of-the-art SSD firmware.

of flash commands, we periodically check whether the requests in the waitlist meet the constraints of cacheline dependencies. Each request gets the ID (i.e. victim ID) of the previous request on the same cacheline through the Pilot. The request exits the waitlist only if the victim ID recorded in the roadbook is already present in the current cacheline. This mechanism ensures sequential access to the cacheline. Naturally, if the request does not require flash I/O, it proceeds directly to the next stage after departing from the waitlist.

Waitlist-based data consistency scheduling does not enforce a global FIFO order but maintains the FIFO order for each cacheline. When a subsequent request on a cacheline is waiting in the queue, it does not affect the processing of requests on other cachelines. This approach ensures that the parallelism of the multiple flash channel is maintained as much as possible.

4 EVALUATION

4.1 Experimental Setup

4.1.1 Implementation platform. We build up an NVMe SSD prototype on an FPGA-based platform. The FPGA builds the data path with memory, DMA controllers, NAND Flash controllers, and a PCIe IP core. All of the crucial functions are defined by software, including NVMe connection, address translation, and flash channel scheduling. According to the data collected from the hardware platform, we build a twinned trace-driven multi-core and multi-flash channel emulation framework based on multi-threading. The most significant difference between the hardware prototype and its twinned emulation platform is the host interface layer. The block I/O trace is preloaded into the memory, so the overhead of fetching instructions is much smaller than the hardware prototype which is limited by the DMA rate. On this emulation platform, we can evaluate the performance of our design with a high level of connectivity (i.e. the request rate of NVMe queues) that is difficult to achieve on a hardware platform. Beyond the performance evaluation, we have verified the effectiveness of PipeSSD on the hardware prototype.

4.1.2 Configuration. The hardware prototype is equipped with 8 flash channels (one flash chip per channel), and we currently use MLC-type NAND Flash chips (64GB per chip). The read and write performance data we collected for the flash chip are shown in Table 1. On our emulation platforms, the latency of flash operations is consistent with the hardware prototype. Our PipeSSD is compared with the DeepFlash which is the state-of-the-art SSD framework on many-core systems [12], and the traditional (Tradition-N, N is the thread number) one-to-many thread modeling. Typical SSD memory size is one-thousandth of the storage space, we follow this pattern to configure our platforms.

Table 1: Parameters setting of MLC-type NAND Flash chip.

	Address setup	Command execution	Data movement
read	3us	40us	60us
program	5us	400us	60us

On high-performance commercial NVMe SSDs, the controller is typically an ARM-based multi-core processor. For example, Samsung 980pro and 990pro are both 5-core, 32-bit ARM processors that support 8 flash channels. A quad-core 64-bit ARM processor is used on our FPGA core board. We use 4 cores as a standard, and the four stages of PipeSSD are pinned to one core each. DeepFlash supports scalability, which we have also scaled down to 4 cores. The traditional scheme starts with 4 workers.

4.1.3 Workloads. We use 5 block I/O traces from the UMassTraceRepository [1] to evaluate our design. Financial1 (F1) and Financial2 (F2) are collected from the OLTP application I/O, and WebSearch1 (W1), WebSearch2 (W2), and WebSearch3 (W3) are generated by a popular search engine. As shown in Table 2, it is obvious that the first group is highly localized, while the second group is not at all. Our design focuses on solving the convoy effect of cache and utilizing the parallelism of multiple flash channels. The performance of these two workloads on cache is very different, which on one hand can test the design of the cache system, and on the other hand, can test the parallelism of the flash level behind the cache.

4.2 Experimental Results

4.2.1 Throughput comparison. We first evaluate the throughput with 4 flash channels and then scale the flash channels to 8. The performance comparison results are shown in Figure 6.

In the evaluation, the baseline is Tradition-4, and the throughput of different methods is normalized to the throughput of Tradition-4 to make a fair comparison. From the evaluation result, we can get the following observations. First, our PipeSSD outperforms all of the other methods, PipeSSD's throughput is 31.2% higher than the baseline on average and 5% higher than DeepFlash on average. Second, when facing poorly localized workloads (e.g. W1, W2, and W3), the advantages of PipeSSD are even more pronounced, up to 36% ahead of baseline. Third, the traditional one-to-many thread model even doubling the number of workers (Traditional-8) does not significantly improve throughput.

Table 2: Data cache hit ratio.

	F1	F2	W1	W2	W3
PipeSSD	50.19%	51.03%	1.08%	0.03%	0.06%
DeepFlash	44.22%	46.17%	1.07%	0.03%	0.06%
Trandition	50.19%	51.03%	1.08%	0.03%	0.06%

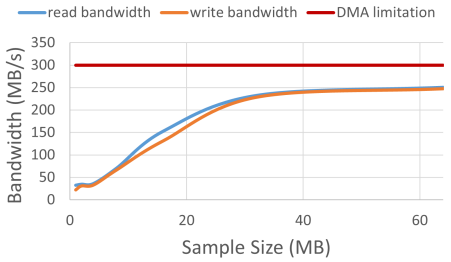


Figure 7: Bandwidth of hardware prototype.

To verify the scalability, we increased the number of flash channels to 8. As the number of flash channels rises, the advantages of the non-blocking design become more obvious. In pipeline-based designs, data streams are not blocked outside the FIL stage. DeepFlash bypasses the cache when the cacheline is occupied to avoid blocking subsequent requests. Our PipeSSD ensures both a continuous data stream into the flash channels and maintains the consistency of the cacheline through data consistency scheduling. As shown in Table 2, the cache hit ratio of PipeSSD does not drop. For workloads with high locality, cache hits can directly reduce the number of long-latency flash I/O operations, which is important for improving SSD performance. PipeSSD also has the advantage of minimizing inter-stage dependency throughout the pipeline, with no loopbacks and a unidirectional flow of data. Taking all these points together, PipeSSD outperforms the baseline in this set of experiments by about 40% in terms of throughput and is 5.2% ahead of DeepFlash.

Besides the regular configurations, we conduct an additional set of experiments with a memory-limited setting. The cacheline competition is much more dramatic, and the non-blocking pipeline is better adapted to this situation than the lock-based multi-thread methods. As shown in Figure 6(c), in such case, PipeSSD's average throughput is 42% higher than the baseline.

4.2.2 Hardware verification. Finally, the PipeSSD is verified on our hardware prototype. Based on the memory model supported by ARM SoC, read-only checking without lock protection is safe in the case of a unique writer. According to the lock-free queue design [2], the message can be safely transferred from the single writer to the single reader via a lock-free queue. Hence, the communications between stages are also based on lock-free queues. To improve memory efficiency, the request with a data field is a large payload, the memory space is statically preserved and the reference of this large payload is passed through the queues.

The performance limitation is not posed by software design but the hardware constraints. Behind the PCIe IP core is the DMA controller which is always listening to the data flows on the PCIe channels. Once the SSD fetches an NVMe request from the SQ, the data of the request will stream into the SSD's memory with the help of the DMA controller. The response data is also carried out by the DMA controller. According to [2], when the FPGA frequency is 100MHz, the bandwidth of DMA is 300MB/s. As shown in Figure 7, the bandwidth of PipeSSD on the hardware prototype can approach the DMA limitation when the sample size of the NVMe request increases. The gap between the theoretical bandwidth and the actual bandwidth is because that data movement requires some control overhead, such as interrupt handling and buffer setup.

5 CONCLUSION

In this paper, we propose the PipeSSD, a lock-free pipeline-based SSD firmware on a multi-core SSD controller. To fully explore the internal parallelism of the NVMe SSD, PipeSSD adjusts the NVMe request workflow by post-placing the data cache to make the pipeline non-blocking and loop-free. To guarantee data consistency, we integrate the pipeline with a cache pilot and data consistency scheduling. We verify the functional correctness of PipeSSD on our hardware prototype. The evaluation result shows that PipeSSD outperforms the state-of-the-art multi-core SSD firmware.

6 ACKNOWLEDGEMENT

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14219422 and GRF 14202123), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151).

REFERENCES

[1] 2023. Umass Traces Repository. http://traces.cs.umass.edu/.
[2] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 43–52.
[3] Jen-Wei Hsieh, Han-Yi Lin, and Dong-Lin Yang. 2013. Multi-channel architecture-based FTL for reliable and high-performance SSD. *IEEE Trans. Comput.* 63, 12 (2013), 3079–3091.
[4] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–12.
[5] Chun-Yi Liu, Yunju Lee, Wonil Choi, Myoungsoo Jung, Mahmut Taylan Kandemir, and Chita Das. 2021. GSSA: A resource allocation scheme customized for 3D NAND SSDs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 426–439.
[6] Renping Liu, Xianzhang Chen, Yujuan Tan, Runyu Zhang, Liang Liang, and Duo Liu. 2020. SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 966–975.
[7] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".
[8] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 442–456.
[9] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. {D2FQ}:{Device-Direct} Fair Queueing for {NVMe}{SSDs}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 403–415.
[10] Jin Xue, Renhai Chen, and Zili Shao. 2022. SoftSSD: Software-defined SSD Development Platform for Rapid Flash Firmware Prototyping. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 602–609.
[11] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in {SSD} based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.
[12] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 121–136.