MCMQ: Simulation Framework for Scalable Multi-Core Flash Firmware of Multi-Queue SSDs

Jin Xue, Tianyu Wang and Zili Shao
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
E-mail: {jinxue, tywang, shao}@cse.cuhk.edu.hk

***Abstract*—Solid-state drives (SSDs) have been used in a wide range of emerging data processing systems. To fully utilize the massive internal parallelism delivered by SSDs, manufacturers begin to utilize high-performance multi-core microprocessors in scalable flash firmware to process I/O requests concurrently. Designing scalable multi-core flash firmwares requires simulation tools that can model the features of a multi-core environment. However, existing SSD simulators assume a single-threading execution model and are not capable of modelling overheads incurred by multi-threading firmware execution such as lock contentions. In this paper, we propose MCMQ, a novel framework for simulating scalable multi-core flash firmware. The framework is based on an emulated multi-core RISC processor and supports executing multiple I/O traces in parallel through a multi-queue interface. Experiment results show the effectiveness of the proposed framework. We have released the open-source code of MCMQ for public access.**

***Index Terms*—solid-state drives, simulator**

I. Introduction

Solid-state drives (SSDs) have been used in a wide range of emerging data processing systems [7], [6] to provide high-performance storage. The design of modern SSDs has been optimized to reduce request latency and improve I/O throughput. Meanwhile, new interfaces, such as the NVM Express (NVMe) protocol, have been proposed to enable high I/O bandwidth and fully exploit the internal parallelism of an SSD. For example, with NVMe, the host can create up to 64k deep I/O queues which are assigned to different applications and threads to submit I/O commands independently in parallel. Internally, flash firmware processes I/O commands from multiple queues and dispatches them to an array of flash chips in a highly concurrent manner to maximize the storage utilization.

In order to handle the tremendous number of concurrent I/O requests arriving over multiple queues, flash firmware needs to perform SSD internal tasks such as cache lookup, address translation and flash transaction scheduling in parallel. Such internal tasks can introduce a high computation load when the number of concurrent I/O requests increases and the microprocessor on an SSD can quickly become a bottleneck if it cannot keep up with the heavy I/O workload. For this reason, manufacturers begin to employ high-performance multi-core microprocessors on SSDs to adapt to the massive internal parallelism.

Although a multi-core microprocessor has potentially more computational power than its single-core counterpart, it does not guarantee a higher I/O performance delivered by an SSD. While multiple cores can process I/O tasks in parallel, they still need to access a set of shared resources such as data cache and the mapping table for address translation. Synchronization mechanisms such as locks are required to provide coordinated accesses to these shared resources and avoid conflicts and inconsistencies. As the number of tasks handled by the SSD increases, the resource contention can be aggravated when processing them in parallel. Thus, the higher computational power brought by a multi-core microprocessor can be hindered by the poor scalability of flash firmware, if it is not carefully designed to reduce resource contention and synchronization overhead among cores [9].

Designing scalable flash firmware requires simulation tools that are capable of accurately modelling the features of a multi-core environment where the firmware is deployed. However, existing simulators [1], [5], [2], [3], [8], [4] assume a single-threading execution model and cannot model the complex thread interaction in a multi-threading flash firmware environment resulting from the high degree of internal parallelism. These simulators typically model only some parts that are considered to contribute the most to the final end-to-end latency, *e.g.* data transfer and flash command execution, while completely ignoring the thread synchronization and contention overhead. As such, scalability and flexibility issues in the flash firmware cannot be detected using these simulation tools.

Due to the complex nature of thread interaction, it is impossible to accurately model the contentions in flash firmware without actually running it on a multi-core environment. Although virtual platforms such as QEMU can provide emulation of multi-core microprocessors, there are several challenges in simulating SSDs based on these platforms. First, in order to fully utilize multiple cores, a thread model, which should be suitable for flash firmware, is required. Given specific functionalities of flash firmware, adopting existing thread libraries such as pthread, which may require operating system support, is an overkill. Second, a skeleton firmware module with basic functionalities and hardware abstractions is required so that new flash translation layer (FTL) algorithms can be easily implemented on top of it and deployed on the simulation platform for evaluation. The skeleton firmware should also provide timing-accurate simulation of the low-level NAND flash array with configurable parameters that can be tuned to match real SSDs. Third, the flash firmware running inside the virtual machine needs to expose a generic interface to the host so that storage protocols such as NVMe can be simulated

through the interface. Lastly, a host driver is required, by which concurrent IO requests generated from applications can be submitted based on real-world scenarios.

In this paper, we propose a novel framework, called MCMQ, for simulating multi-core flash firmware. MCMQ is directly built upon an emulated multi-core RISC-V microprocessor based on QEMU that enables to detect the scalability and flexibility of FTL designs, which cannot be solved by existing SSD simulators based on discrete event simulation. Specifically, MCMQ provides four levels of abstractions with flexible configurations to address the above challenges:

- First, we design a lightweight thread model by which various FTL threads can be created and executed concurrently without operating system support. In addition, our simple model includes several thread synchronization primitives that enables multiple FTL threads to synchronize accesses and process concurrent requests. We have implemented our thread model and released it as an open-source thread library.
- Second, we implement a skeleton flash firmware for our proposed simulation framework with basic functionalities such as timing-accurate low-level flash media simulation and garbage collection (GC) manager. New FTL components can be integrated with the skeleton flash firmware to generate new flash firmware, whose scalability and flexibility on a multi-core microprocessor can be evaluated and studied through the simulation framework. Internally, the flash firmware emulates low-level flash memory in a timing-accurate manner. The flash firmware is directly executed on a QEMU-based multi-core RISC-V processor in real time so that CPU execution latency and thread synchronization overhead can be modelled accurately.
- Third, we set up a communication mechanism based on virtual sockets and shared memory provided by the QEMU virtual platform, through which the host can send requests to and exchange data with the simulated SSD. Based on it, we implement a simulation of the NVMe protocol with which the host driver creates device queues and exchanges I/O commands. Various isolation techniques of the NVMe protocol, *e.g.* namespaces and NVM sets are supported to facilitate the studies of scalable flash firmware.
- Finally, on the host side, we implement a frontend driver, by which I/O requests from separate host threads can be submitted to the SSD simulator based on a flexible configuration interface. This effectively simulates real I/O scenarios where the host OS or application threads submit I/O commands to an NVMe SSD via multiple device queues in parallel.

We implement the proposed simulation framework and carry out performance evaluation. We conduct experiments on real hardware SSD platform to validate flash operation latencies and simulations with concurrent synthetic I/O traces to show the impact of non-scalability flash firmware on I/O performance with different levels of parallelism. We also evaluate how various isolation techniques can affect the scalability of the flash firmware. Experimental results show that our simulation framework is capable of modelling resource contention and synchronization overheads arising from different levels inside a multi-core SSD, thus providing a time-accurate simulator for investigating scalable multi-core flash firmware. MCMQ has been released as an open-source project at an anonymous repository https://github.com/Jimx-/mcmq.

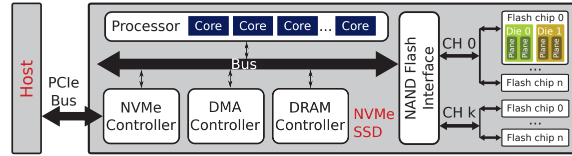

Fig. 1: The overall architecture of an NVMe SSD.

II. Background

A. NVMe SSD Devices

Figure 1 illustrates the overall architecture of a modern NVMe SSD equipped with a multi-core high performance processor. The SSD is connected to the host system through a PCIe (PCI Express) bus with the NVMe interface. The processor and controllers are connected to the flash memory components through an NAND flash interface which contains multiple bus *channels*. Each channel is connected to one or more NAND flash memory chips.

As shown in Figure 1, a flash chip is further organized in a hierarchy of execution units to maximize the I/O concurrency. Typically, a flash chip is divided into multiple *dies*, which are further divided into *planes*. The dies are allowed to execute different commands independently in parallel. Each plane consists of multiple *blocks*, which are the granularity for an erase operation. Blocks are divided into *flash pages* which are the basic data unit for read or write operations.

B. Flash Firmware

The main task of flash firmware is to retrieve commands from the host via the host interface layer (HIL), performing necessary transforms on the commands and dispatch them to the flash chips through the flash interface layer (FIL).

Specifically, for NVMe SSDs, the host and the device communicate with the NVM Express (NVMe) protocol over the PCIe bus. There are multiple pairs of *submission queues (SQ)* and *completion queues (CQ)* for submitting commands to and receiving command results from the SSD.

To submit an I/O command, the host selects an I/O submission queue and set the next empty slot with the command. After that, it writes the updated queue tail index to the corresponding SQ tail doorbell register mapped into the host memory to notify the device of the new incoming command. Upon the notification, the host interface layer fetches commands in the SQ from the host memory based on its internal SQ head index until it catches up with the tail index written by the host.

The FTL first dequeues a user request from the device-level request queue. The request is then segmented into multiple

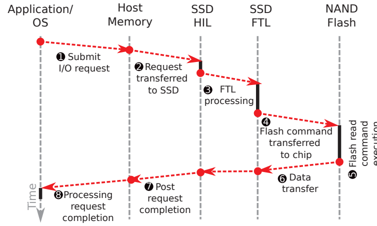

Fig. 2: End-to-end request latency diagram for a read request in an NVMe SSD.

flash transactions at the granularity of page size based on the start logical block address (LBA) and I/O size of the request. It then looks up the data cache for the requested pages. If a logical page is not found, it is then forwarded to the *address translation module*, which maintains a *mapping table* and maps the logical page addresses (LPAs) of the requested pages to their physical page addresses (PPAs).

After the target LPA of a flash transaction is translated to the physical location, the flash transaction is inserted into a transaction queue based on the flash chip. The transaction scheduling unit (TSU) reorders the transaction based on their priorities and schedules to send the transaction to execute on the flash chip in an interleaved manner via the FIL. When all transactions of a user request complete, the TSU notifies the HIL to post a completion entry containing the command result to the CQ paired with its corresponding SQ and notifies the host with an interrupt.

C. Motivation

Modelling end-to-end request latency is crucial in simulating the performance of a flash firmware design as it directly determines the time that an application need to stall on an I/O request. Because of the layered design of the flash firmware, a request needs to go through multiple stages and different components can contribute to the final end-to-end request latency. The end-to-end request latency includes several command/data transfers between the host and the SSD and between the SSD DRAM and flash chips, computation time (*e.g.* FTL address translation) and the execution time of actual flash commands on the flash chips. As illustrated in the request latency diagram in Figure 2, the end-to-end request latency includes several command/data transfers between the host and the SSD (❷, ❼) and between the SSD DRAM and flash chips (❹, ❻), computation time ❸ (*e.g.* FTL address translation) and the execution time of actual flash commands on the flash chips (❺). Traditionally, flash command execution and read data transfer from flash chip are considered to be the most time-consuming steps when handling a request. Thus, existing simulators [1], [5], [2], [3] tend to only take these two steps into account in the request latency model while ignoring the latency contributed by other parts in the request processing, *e.g.* computation.

However, with a multi-core processor, there are two major aspects prolonging the computation latency in the FTL. First, limited mapping table cache will make the FTL frequently issue extra flash transactions to read evicted entries, which stalls the user I/O requests. Second, with a multi-core processor, while multiple threads can perform FTL processing for more than one request in parallel, some in-memory resources (*e.g.* the data cache and the cached mapping table) are shared by the threads. Locks are needed to provide synchronized accesses to these shared resources. In case of coarse-grained locking, a FTL thread may need to hold a global lock during the entire FTL processing. Other FTL threads that are attempting to process requests have to wait until the thread releases the lock before they can proceed, resulting in longer FTL processing latencies. Thus, although a multi-core processor can provide more computational power than its uniprocessor counterpart, it does not guarantee a higher performance of the flash firmware due to the synchronization overhead. In fact, compute latency in the FTL can increase with a non-scalable flash firmware and take up most of the total I/O processing time [9].

In order to provide a simulation platform for evaluating the scalability of a flash firmware design on a multi-core SSD, an accurate model of both the FTL processing latency and the synchronization overhead incurred by complex thread interaction is needed. Although MQSim [8] and SimpleSSD [4] include FTL processing time in their latency model, they only estimate the latencies of some operations that can stall the request processing and CPU execution time based on instruction counts while still assuming a single-thread execution model.

III. MCMQ Design

The overall architecture of the proposed simulation framework is illustrated in Figure 3. The simulation framework consists of a frontend driver that runs on the host system and the flash firmware which is emulated in a QEMU RISC-V virtual machine. The frontend driver uses multiple host threads to generate I/O requests and submit them to the flash firmware in parallel through the host interface. Upon receiving requests, the flash firmware uses a set of FTL threads to process them concurrently.

A. Thread model

In a multi-core flash firmware module, requests arriving through multiple I/O queues exposed by the NVMe protocol are processed by multiple FTL worker threads. Thus, we need a thread model to specify how I/O requests are dispatched to the worker threads for processing. In MCMQ, we use a static assignment model where one NVMe queue is managed by one FTL thread exclusively and all requests arriving through a queue will be processed by its FTL thread. Each thread is pinned to run on one CPU core. Besides the FTL threads, the flash firmware also has a host interface layer (HIL) thread that handles host communication and a transaction scheduling unit/flash interface layer (TSU/FIL) thread that schedules to execute the flash transactions and simulates the low-level flash media in a timing-accurate manner.

Based on this thread model, MCMQ provides a lightweight thread library with thread synchronization primitives to simplify designing multi-threading flash firmware. All FTL

threads share the same data cache and address mapping table and fine-grained spin locks are utilized to provide coordinated accesses to these resources. The HIL thread and the TSU/FIL thread manage host NVMe queues and simulated NAND flash array exclusively; thus no locks are required for these resources.

In addition, all threads except the HIL thread have an event queue for inter-thread communication. The threads run an event loop that wait for events to arrive in its event queue and process them accordingly. The HIL thread uses the event queue to dispatch a user request arriving over the NVMe queue to its FTL thread. FTL threads submit flash transactions generated during the FTL processing to the event queue of the TSU/FIL thread for scheduling and execution.

B. Host interface

The host interface between the frontend driver and the emulated SSD simulates the host memory and the PCIe link required by the NVMe protocol. The frontend driver enqueues I/O commands to the queue pairs located on the simulated host memory and rings the doorbell registers on the SSD through the simulated PCIe link. When an I/O command completes, the SSD copies the response to the host memory and sends an interrupt request (IRQ) through the PCIe link to notify the frontend driver. The simulated host memory is implemented by the Inter-VM Shared Memory device (ivshmem) provided by QEMU. The shared memory device creates a POSIX shared memory object on the host system and exposes it through a virtual PCI device to the emulated flash firmware. The shared memory can be mapped into both the physical memory of the virtual machine and the process memory of the frontend driver. As such, the frontend driver and the simulated SSD can exchange I/O commands and data based on queue pairs allocated on the shared memory.

The flash firmware can map the shared memory region into its physical memory address space by writing the base address register (BAR) of the virtual PCI device and accesses it through the physical memory region. Similarly, the frontend driver can use the `mmap` system call to map the shared memory object into its process memory space. After that, the frontend driver can allocate queue pairs in the shared memory region and exchange their base offsets with the flash firmware with the NVMe protocol. Updates made by the frontend driver and the flash firmware to shared memory region will be visible to the other end.

The simulated PCIe link is implemented by a virtual socket (`virtio-vsock`). The virtual socket allows the frontend driver and the flash firmware to send messages to each other without involving network protocols. Through the virtual socket, the frontend driver encodes PCIe register writes into messages and sends them to the flash firmware. The flash firmware will be notified by a hardware interrupt emulated by QEMU and updates the doorbell registers accordingly.

In the flash firmware, a dedicated HIL thread manages all NVMe queues created by the frontend driver with admin commands. Upon receiving messages from the virtual socket,

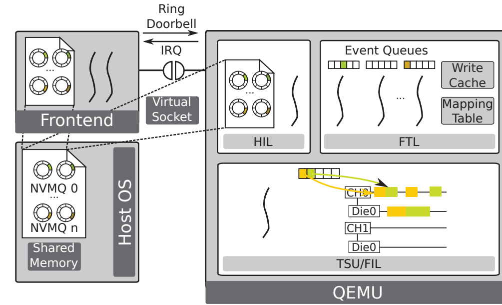

Fig. 3: The overall architecture of the proposed SSD simulation framework.

it decodes the message to extract the address of the target register the frontend driver is writing to and performs the updates accordingly. In case of an SQ tail doorbell write, it copies the incoming I/O requests from the shared memory region and parses the requests. The requests are then sent to the event queue of the FTL worker thread that manages the NVMe queue for further FTL processing.

C. Flash translation layer

The FTL in the proposed simulation framework has a number of FTL worker threads which match the number of NVMe queues created by the host. Each FTL worker thread has a local event queue and runs an event loop to wait for incoming events from its queue and process them. All threads use event queues and inter-processor interrupts (IPI) to communicate with each other. When a new user request arrives at the host interface, it is submitted to the corresponding FTL worker thread for FTL processing. An FTL worker handles all FTL tasks including cache lookup and address translation for a request. Resources related to FTL processing, *e.g.* data cache and cached mapping table, are shared among all FTL workers. Fine-grained spin locks are used to provide synchronized accesses to the shared resources. After FTL processing for the incoming flash transactions is finished, they are submitted to the event queue of the TSU/FIL thread.

The FTL also allows the frontend driver to set up *NVMe namespaces*. Namespaces are disjoint logical block address ranges on an NVMe SSD to provide logical isolation for multi-tenancy. In MCMQ, each namespace has separate data cache and address translation data structures so that contention between concurrent I/O requests to different namespaces is reduced. Furthermore, MCMQ also supports creating *NVM sets*. A NVM set is a set of hardware resources (*e.g.* flash chips) that can be assigned to a namespace. The FTL will only dispatch an I/O request in a namespace to the NVM set assigned to it. In this way, storage contention for different I/O requests can be further reduced.

D. Transaction scheduling and flash interface layer

SSDs have a hierarchical design that enables a high degree of internal parallelism. Dies in a flash chip can execute different commands at the same time. Even within the same die, planes can execute the same operation in parallel on the

same offset in parallel. This hierarchical design allows the flash firmware to execute flash operations in an interleaved manner to reduce request latencies. For instance, to handle a read transaction, the flash firmware first transfers the flash command to the flash chip through a channel and then triggers the command execution on the die. Finally, the page data are transferred back to the flash firmware. Because flash channels and dies can work independently, it is possible to overlap data transfers with the execution of other commands to reduce command delay, as shown in Figure 3. The flash firmware can also reorder flash transactions to fully utilize such interleaved execution. In MCMQ, we manage the NAND flash array simulation and transaction scheduling with the TSU/FIL thread.

The TSU/FIL thread fetches incoming flash transactions from its event queue and dispatches them to a set of user request and GC queues based on the physical locations of their target flash pages. The queues can be reordered based on the priorities of the transactions to reduce contention and maximize I/O throughput. The TSU/FIL thread is also responsible for simulating the low-level flash chips. It simulates the data transfer and command execution latencies with RISC-V timers based on configurable flash latency characteristics. It also keeps track of the busy/idle status of each channel and chip, and submits queued transactions to execute on the simulated flash chips in an interleaved manner. After a transaction is completed, the FTL worker that issued the transaction is notified, which will check whether the user request associated with the transaction is completed and generate response to the host if necessary.

IV. Evaluation

A. Experiment setup

We run the simulation framework in a virtual machine with 8 logical cores (allowing up to 6 FTL workers) and 2GB RAM hosted by QEMU 6.0.0 for 64-bit RISC-V. The frontend driver is run on the same host machine as the QEMU virtual machine. All experiments are run on a host machine with a 8-core Intel(R) Core(TM) i7-9700 CPU with 32GB RAM. The simulation framework can be configured to simulate different SSDs. In the experiment, we simulate an MLC SSD with a capacity of 512GB and 8 flash channels.

B. Flash operation time

Flash operation time, *e.g.* read latencies for LSB and MSB flash pages, can be configured on MCMQ to match real hardware environment. To obtain realistic flash operation latencies and tune the simulator, we conduct preliminary experiments on a development board equipped with NAND flash chips. As shown in Figure 4, the development board has an ARM Cortex-A53 processor which is connected to 8 Micron MT29F1T08 flash memory chips (128GB each) through the Open NAND Flash Interface (ONFI). We run a program on the processor to repeatedly issue 100 flash commands to a chip and measure the flash operation latencies (*i.e.* the time after the command is issued and before the flash chip becomes ready again). Table I summarizes the latencies for different flash operations on real flash memory chips.

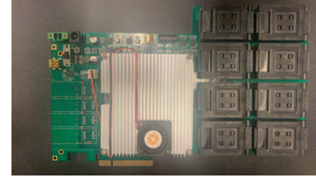

Fig. 4: Hardware SSD development board for measuring flash operation latencies.

TABLE I: Real and simulated flash operation latencies.

	Read (μ s)		Program (μ s)		Erase (ms)
	LSB	MSB	LSB	MSB	
Real	56.2 ± 1.6	71.5 ± 8.4	381.2 ± 157.3	1578.5 ±68.5	1.20 ± 0.008
MCMQ	60.7 ± 17.2	71.3 ± 26.6	381.3 ± 96.1	1584.6 ± 323.6	1.20 ± 0.3

Based on the measured operation time, we set the flash operation latencies to the real mean values accordingly. Because MCMQ simulates flash chips in real time, the actual flash operation latencies can be different from the configured values. We also measure the actual command latencies during the simulation, which is shown in the lower row of Table I. As the table shows, MCMQ can be configured to simulate latency characteristics of real flash memory chips accurately.

C. Multi-core firmware execution

We first evaluate the simulation framework with multiple synthetic I/O traces with uniform workloads. Each NVMe queue is managed by a thread of the frontend driver. The I/O threads generate requests to each logical page in the LPA space with equal probability. The size of each I/O request is set to 8KB. The request start address is aligned to page boundary. All I/O threads share the same namespace so that they are subject to contention in both the memory and the underlying storage. As shown in Figure 5, as the number of FTL threads increases, the I/O throughput does not scale linearly. Instead, the I/O throughput of each I/O trace decreases and the end-to-end request latency increases significantly, when more FTL threads are used. This shows that the computational power brought by a multi-core microprocessor cannot be fully utilized to

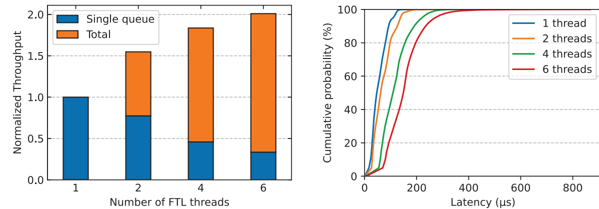

Fig. 5: I/O throughput and end-to-end request latency on uniform workload with a single namespace of MCMQ.

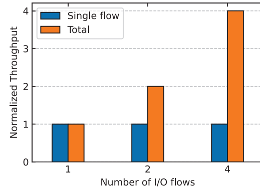

Fig. 6: Normalized I/O throughput of MQSim with different numbers of I/O flows.

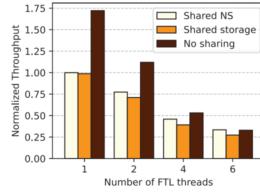

Fig. 7: I/O throughput with different isolation mechanisms.

improve the SSD performance, if the flash firmware is not carefully designed to reduce resource contention and improve scalability. Our simulation framework is capable of modelling various contention in an SSD under a multi-core environment and thus can provide a platform for designing highly scalable flash firmware.

Because no existing simulator supports simulating real multi-core firmware execution, we select MQSim [8] with the multi-queue feature for comparison. MQSim supports simulating submitting I/O requests concurrently but CPU latency and synchronization overhead are excluded from the latency model of MQSim. Thus, multiple requests can be dispatched and executed in parallel as long as there is no flash level contention between them, even if they share the same memory resources such as the write cache and the mapping table. Figure 6 shows the simulation results of running 1, 2 and 4 uniform I/O flows with the same configuration as MCMQ on MQSim. As the figure shows, the simulated I/O throughput for a single flow remains constant even when more flows are introduced, and the overall throughput of all flows scales linearly with the number of I/O flows. As write requests from different I/O flows are served with the same write cache, there can be synchronization overhead between flows that leads to slowdown of a single flow. These behaviour and thread interaction are not captured and simulated by existing simulators such as MQSim.

D. Isolation simulation

There are several logical and physical isolations supported by the NVMe protocol to reduce resource contention. Figure 7 shows the simulation results for different isolation mechanisms. In shared namespace, all I/O traces use the same namespace. With shared storage, I/O traces uses different namespaces; however the underlying flash storage is shared among all namespaces. With no sharing, I/O traces uses different namespaces, each assigned with a separate flash channel to avoid storage contention. As shown in Figure 7, although namespace isolation reduces synchronization overhead between different FTL threads, the I/O performance is worse than shared namespace because the data cache is partitioned. Each namespace has a smaller data cache and the number of issued cache writebacks increases. No sharing has a significantly higher performance than both shared namespace and shared storage. The reason is two-fold. First, with no sharing, each namespace is assigned a subset of the underlying flash resource and the capacity of each namespace is smaller so the requests are more concentrated and cache misses are reduced. Second, each I/O traces access only the in-memory cache and mapping table and the underlying flash media that are assigned to its namespace so contention is avoided. This shows that our simulation framework is capable of modelling different isolation mechanisms of SSDs, which is crucial to the scalability of flash firmware.

V. Conclusion

In this paper, we present a multi-core multi-queue SSD simulation framework called MCMQ to help investigate the scalability issue of flash firmware. We implement MCMQ based on an multi-core processor emulated by QEMU and simulate low-level flash media in a timing-accurate manner. Experimental results show that MCMQ is capable of modelling several synchronization overheads and underlying storage contention that can be incurred by the flash firmware under a multi-core execution environment and hinder SSD performance. The proposed simulation framework can be used as a basis for future research on highly scalable flash firmware.

VI. Acknowledgment

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 15224918), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151).

References

[1] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In EuroSys '17, pages 127–144, New York, NY, USA, Apr. 2017. Association for Computing Machinery.

[2] 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 ICS '11, pages 96–107, New York, NY, USA, May 2011. Association for Computing Machinery.

[3] M. Jung, W. Choi, S. Gao, E. H. Wilson III, D. Donofrio, J. Shalf, and M. T. Kandemir. NANDFlashSim: High-Fidelity, Microarchitecture-Aware NAND Flash Memory Simulation. *ACM Transactions on Storage*, 12(2):6:1–6:32, Jan. 2016.

[4] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir. SimpleSSD: Modeling Solid State Drives for Holistic System Simulation. *IEEE Computer Architecture Letters*, 17(1):37–41, Jan. 2018.

[5] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *2009 First International Conference on Advances in System Simulation*, pages 125–131, Sept. 2009.

[6] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX} Annual Technical Conference (USENIX} ATC} 19)*, pages 603–616, 2019.

[7] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, 2016.

[8] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[9] J. Zhang, M. Kwon, M. Swift, and M. Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, 2020.