Pipette: Efficient Fine-Grained Reads for SSDs

Shuhan Bai*
Huazhong University of Science and Technology
City University of Hong Kong

Hu Wan*
Yun Huang
Xuan Sun
City University of Hong Kong

Fei Wu
Changsheng Xie
Huazhong University of Science and Technology

Hung-Chih Hsieh
YEESTOR Microelectronics Co., Ltd

Tei-Wei Kuo
City University of Hong Kong
National Taiwan University

Chun Jason Xue
City University of Hong Kong

ABSTRACT

Big data applications, such as recommendation system and social network, often generate a huge number of fine-grained reads to the storage. Block-oriented storage devices tend to suffer from these fine-grained read operations in terms of I/O traffic as well as performance. Motivated by this challenge, a fine-grained read framework, Pipette, is proposed in this paper, as an extension to the traditional I/O framework. With an adaptive caching design, Pipette framework offers a tremendous reduction in I/O traffic as well as achieves significant performance gain. A Pipette prototype was implemented with Ext4 file system on an SSD for two real-world applications, where the I/O throughput is improved by 31.6% and 33.5%, and the I/O traffic is reduced by 95.6% and 93.6%, respectively.

CCS CONCEPTS

• **Software and its engineering → File systems management**;
• **Information systems → Flash memory**.

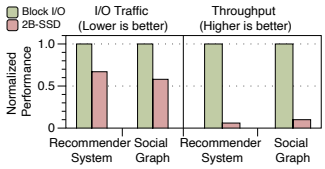
KEYWORDS

file system, solid-state drive, fine-grained reads

ACM Reference Format:
Shuhan Bai, Hu Wan, Yun Huang, Xuan Sun, Fei Wu, Changsheng Xie, Hung-Chih Hsieh, Tei-Wei Kuo, and Chun Jason Xue. 2022. Pipette: Efficient Fine-Grained Reads for SSDs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530467

1 INTRODUCTION

Fine-grained reads, each a few hundred bytes or less, are prevalent and widely reflected in many large-scale big data applications, such as recommendation system [7, 8], social network [4], and search engine [9]. For example, recommendation system handles sparse input features by looking up embedding vectors from SSDs [11]. The typical size of an embedding vector is 128B [6], while the minimal read granularity of block storage devices is often at 4KB, which leads to severe performance degradation. One reason is that the block-based interface induces a high read amplification, which indicates the ratio of the amount of data read from the storage device to the amount of data requested by the application. Another reason is that fine-grained reads are not adaptive to the read-ahead strategy and the page cache mechanism in current file systems.



Fine-grained reads are prevalent!

Applications	Small objects	Average size
Recommender System	Embedding vector	128B~2KB
Social Graph	Edge, Node	11.3B 87.6B

Figure 1: Fine-Grained Reads Dominated Applications [2, 7].

Recently, several works have been proposed to exploit the byte-accessibility of SSDs [1, 3, 10, 12]. 2B-SSD [3] proposes a dual-interface SSD that allows accessing the SSD in both byte and block granularity. Figure 1 shows the normalized I/O traffic and throughput of 2B-SSD compared to block I/O for the fine-grained read-dominated recommendation system and social graph. Although 2B-SSD reduces read amplification, it suffers from poor read performance. 2B-SSD requires a page fault or a DMA mapping before accessing through the memory interface. These operations are performed on the critical path and thus contribute to user-perceived latency. 2B-SSD also simply bypasses the I/O stack, without supporting data locality. The other recent work CoinPurse [12] exploits the byte addressability of SSDs for fine-grained writes only.

This paper proposes Pipette, a fine-grained read framework, to exploit byte-accessibility of SSDs and fine-grained caching to significantly reduce read amplification while achieving high read performance. Pipette exposes host memory buffer (HMB) to the device and establishes DMA mapping between the controller memory space and the host-assigned memory space. Thus, the host CPU can seamlessly access data in both byte and block granularity between the SSD and the host DRAM. Pipette includes a fine-grained read cache in the host side with an adaptive caching strategy to determine which fine-grained data should be included. Applications can transparently exploit the advantages of both the byte-addressable SSD and the main memory. The orchestrated fine-grained read cache dynamically adapts to different data sizes and maintains efficient memory utilization. Furthermore, a dynamic allocation strategy is designed to balance the memory usage of the page cache and the fine-grained read cache, to provide improved performance for

*Both authors contributed equally to this work.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530467

different access patterns. We implement the Pipette prototype with Ext4 file system on a real SSD development platform and evaluate it under a variety of synthetic and real workloads.

In summary, this paper makes the following contributions:

- We propose Pipette, a fine-grained read framework, as an extension to the traditional I/O framework (which has good I/O performance in handling the data spatial locality), offering a tremendous reduction in I/O traffic.
- We propose an adaptive caching design that explores a suitable mixture of temporal and spatial locality, avoiding coarse-grained reads from polluting the memory cache.
- We implement and evaluate the Pipette prototype with Ext4 file system on an SSD. Experimental results show that Pipette improves I/O throughput by 31.6% and 33.5% for real-world recommendation system and social graph, respectively.

2 BACKGROUND

2.1 Conventional Block-based Read Path

Current I/O stack requires many operations to service a single read request, as shown by the dotted box in Figure 2. A conventional read falls into the Virtual File System (VFS) layer to manage the information with page cache. Upon a cache miss, memory pages are allocated for the missing file blocks while conducting read-ahead. The file system will retrieve pages' logical block addresses (LBAs) and issue a block request to the block layer. The block layer merges associated read requests and adds them into request queues for scheduling. Then, the request is dispatched to the device driver to generate an I/O command. The storage device receives the I/O command and serves the request. The requested and read-ahead pages will be read into the page cache. The traditional I/O framework performs well in handling data spatial locality for coarse-grained reads due to its read-ahead and paging mechanism. However, fine-grained reads scattered around lead to a huge amount of unnecessary data pollutes memory, causing sub-optimal utilization of the page cache.

2.2 Byte-Addressable Interface for SSDs

NVMe protocol defines Controller Memory Buffer (CMB), a general-purpose region of controller memory exposed as part of a PCIe Base Address Register (BAR) for SSDs, to notify the host to add the extended memory mappable region at the system reset stage. CPU can directly access SSDs with Memory-Mapped I/O (MMIO) by performing memory-like accesses redirected to the CMB. This interconnect scheme handles fine-grained reads in the following fashion: SSD controller reads pages from flash chips to the CMB, invokes page fault or sets DMA mapping, and then transfers fine-grained data to the host through MMIO or DMA [3]. Since there is no caching supported between the host and SSDs, the above solution for realizing byte-granularity accesses cannot cache hot data in memory, missing chances of exploiting the performance benefit of the host-side DRAM.

3 PIPETTE DESIGN

Pipette is a fine-grained read framework that extends the traditional I/O framework for fine-grained reads. Pipette optimizes fine-grained reads to maximize read performance while minimizing read amplification. Figure 2 illustrates the overall system architecture of Pipette. To minimize read amplification, Pipette exploits byte-accessibility of modern SSDs to provide a byte-granular read path (§3.1). To reap the benefit from the data temporal locality and the faster access to the main memory, Pipette provides a fine-grained read cache, which caches fine-grained data promoted from the SSD to the host DRAM via byte-addressable I/O interconnection (§3.2).

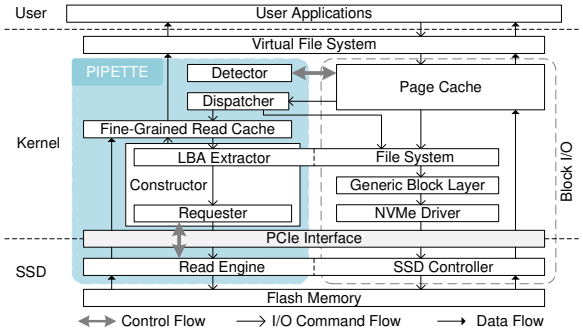


Figure 2: Pipette Architecture.

3.1 Fine-Grained Read Path

Pipette introduces a fine-grained read path atop the byte-addressable interface of SSDs while keeping the traditional block I/O path unchanged. The host can use both the block-based interface and the byte-addressable interface to read.

3.1.1 HMB-based Byte-Addressable Interface. Unlike existing works that leverage CMB to enable a byte-addressable interface, Pipette leverages HMB, a portion of host memory that host provides to device controller for exclusive use. The device establishes DMA mapping that links the controller memory space and the host-assigned memory space during enabling the HMB feature at the initialization stage. Once completed, there is no overhead for data transferring preparations. The host CPU can seamlessly access data in byte granularity across the SSD and the host memory. Fine-grained reads are processed as follows: SSD controller reads NAND pages to the read buffer, and transfers demanded data to the HMB via DMA.

3.1.2 Fine-Grained Read Path Flow. There are five hardware and software components on the fine-grained read path. The detailed design of each component is presented below.

A per-file hash lookup table is created once the user application opens the file that served the fine-grained read for the first time. A fine-grained read request goes through the VFS layer and is first performed by the page cache. If a cache miss occurs, **Fine-Grained Access Detector** is triggered (*see **Detector** in Figure 2*). It verifies the permission to enable byte-granular datapath and maintains all access ranges, so that Pipette can determine which part of each page is demanded. **Read Dispatcher** receives all read requests from the page cache and dispatches them to either the byte-addressable interface or the block I/O interface, mainly based on the data size. *This is the function of **Dispatcher** described in Figure 2.*

A fine-grained read is dispatched to the lookup module of the **Fine-Grained Read Cache**, i.e., the corresponding per-file hash lookup table (*as shown by the bottom left corner in Figure 3*), to

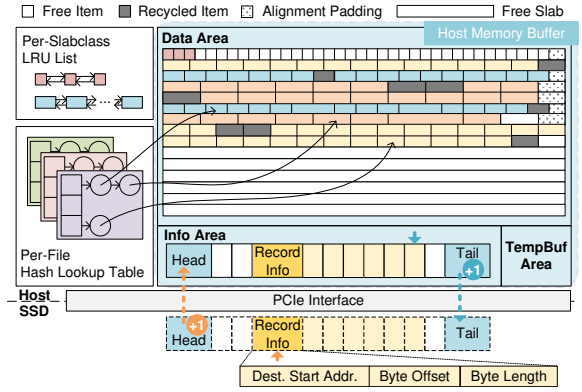


Figure 3: Fine-Grained Read Cache Overview.

retrieve whether the required page range is cached in the host DRAM. Ultimately, the hit data will be returned to the user. As for normal reads, they are handled by the conventional block-based read path for subsequent processing.

When a fine-grained read miss happens, it is sent to the **Fine-Grained Access Constructor**, *which is referred to as* ***Constructor*** *in Figure 2*. The constructor requests **LBA Extractor**, a file system extension for fine-grained reads to bypass the generic block layer, to retrieve LBAs of pages that include needed data, and notifies the **Requester** to submit reconstructed read requests to the SSD.

While the device is reading pages from flash chips, the insertion logic of the **Fine-Grained Read Cache** is switching on. As shown in Figure 3, the fine-grained read cache consists of three partitions: Info Area, Data Area, and TempBuf Area. Info Area is jointly managed by the host and the device to mainly maintain data's destination addresses. Data Area caches byte-granular data in an orchestrated structure that considers various data sizes and balances memory allocation. TempBuf Area is a temporary buffer for low-reuse data to avoid polluting precious memory. The fine-grained read cache allocates space for the small read request according to the recorded access size and inserts a new entry into the corresponding hash table, which also holds assigned data addresses. Afterward, it maintains the destination address in the Info Area (*see host-side step 3a in Figure 4*) and increases the tail's value to assist subsequent read operations. On the other side, after transferring NAND pages through device's internal data path, the SSD digests items in Info Area and increases the head's value, which is read by the host. *This is the function illustrated by the I/O command flow between* ***Constructor*** *and* ***Fine-Grained Read Cache*** *in Figure 2.*

Fine-Grained Read Engine is invoked when the device receives the reconstructed read request. As illustrated in device-side step 3 in Figure 4, the engine processes fine-grained reads in three phases. 1) Load NAND pages to the pre-allocated read buffer. 2) Consume Info Area items to get destination addresses assigned simultaneously as reading full flash pages. 3) Extract all access ranges and transfer them to destination addresses. When the fine-grained read request completes, the engine increases the head's value in the Info Area to indicate the arriving new data and returns the demanded data to the fine-grained read cache (*see data flow between* ***Read Engine*** *and* ***Fine-Grained Read Cache*** *in Figure 2*).

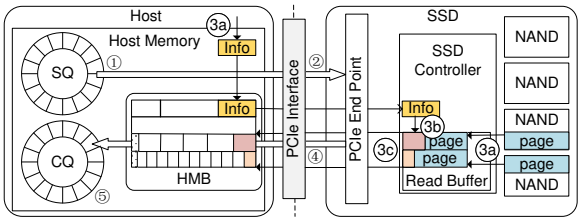


Figure 4: Fine-Grained Read Flow of Read Engine.

3.1.3 Data Consistency Guarantee. Dual caches, i.e., the page cache and the fine-grained read cache, incur possible consistency issues. Write requests first store data in memory pages and then flush corresponding pages to device blocks during persistence. Read requests that occur after the update can obtain the up-to-date data from the page cache, but the proposed fine-grained read cache may still store the old data. When updated pages in the page cache are reclaimed, outdated data from the fine-grained read cache may be returned to fine-grained reads. To guarantee data consistency, Pipette enforces applications to check the fine-grained read cache every time a write operation occurs and delete the found item. In this way, subsequent read requests will either get the updated data from the page cache or the latest data from the flash memory. Moreover, when reads and writes of the same data happen simultaneously, the built-in page lock state can ensure read data's correctness.

3.2 Fine-Grained Read Cache

Frequently accessed data shall be promoted to the host-side shared region for better performance. A fine-grained read cache assisted by hash lookup tables and LRU lists is introduced to facilitate effective caching for tiny reads with byte granularity.

3.2.1 Cache Organization. The data layout of the Data Area inside the fine-grained read cache is illustrated in Figure 3. It organizes memory into uniformly sized slabs, each of which contains predivided items with the same capacity. Slabs are classified into different classes according to the contained items' capacity, which can be set flexibly. Pipette stores data in the smallest possible slab class that can completely accommodate data when no entry is found in the lookup table. Each slab class records the start offset of the following free item and the number of available items remaining in the last allocated slab, and maintains an LRU list of items holding data. One free slab will be requested if no free space remains in the corresponding slab class. Once Pipette can no longer allocate free memory from the shared memory region, two techniques are adopted according to the dynamic allocation strategy (see § 3.2.4 for details): 1) Evict the least recently used item within that slab class, increase the slab class's eviction count, and record this recycled item's start offset into the slab class's cleanup array. 2) Randomly pick an additional slab class with more than one slab, move one of its slabs' data to the allocated memory out of the fine-grained read cache, and record the two offsets before and after migration.

3.2.2 Adaptive Caching Mechanism. Pipette employs an adaptive caching mechanism to fully use the fine-grained read cache capacity. It monitors access patterns and dynamically adjusts a threshold, which is the metric to decide whether the data item should be

cached into the fine-grained read cache, according to the reusability of workloads. The item's reference count is compared against the current threshold on every access, preventing cold data from occupying precious mapping memory resources.

The traditional paging mechanism promotes every accessed page to the host DRAM, polluting the page cache by pages accessed infrequently. We initialize a default threshold to filter data with low reuse. To detect and dynamically adapt to different data reuse patterns, Pipette maintains an access counter and a reuse counter to record the total number of byte-granular accesses and the number of repeated fine-grained accesses, respectively. The reusability of applications is measured by the ratio of the reuse count to the access count. A higher ratio indicates that more data is accessed repeatedly. Pipette decreases the threshold when there is high data reuse to allow data promotion frequently and increases the threshold in the case of low data reuse. We set minimum and maximum ratios to dynamically tune the threshold. If the current ratio is less than the minimum, then the threshold is increased so that the data is cached infrequently in the low data-reuse situation, and vice versa.

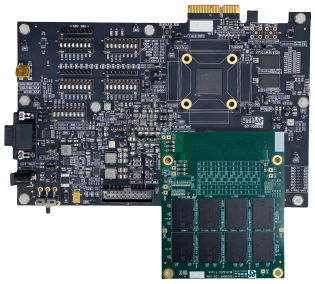
3.2.3 Adaptive Reassignment Strategy. When workloads change, the initially allocated memory to each slab class may no longer be appropriate for subsequent applications. To balance read performance and cache hit ratio of different request sizes, an adaptive reassignment strategy assists Pipette in adapting to different workloads. It periodically re-balances slab allocation to match the current workload without hindering applications execution. A maintenance thread is used to monitor the use of slabs in each slab class by checking their eviction counts. It selects the slab class whose eviction counts unchanged in stages and picks one of its slabs for reassignment. Another re-balance thread is maintained to respond to the reassignment requested by the maintenance thread. It allocates a slab-size spare memory and moves the victim slab to this region. The recycled slab will finally be returned to the free-slab pool of the fine-grained read cache for serving subsequent read requests.

3.2.4 Dynamic Allocation Strategy. The dynamic allocation strategy adaptively makes the memory usage trade-off between two caches, the page cache and the fine-grained read cache, by comparing their hit ratio. Pipette maintains respective hit counters and access counters for these two caches. The hit ratio can be calculated based on these counters, that is, the ratio of the hit count to the access count. The higher the cache hit rate, the more memory space should be assigned to the corresponding cache. Specifically, if the hit ratio of the fine-grained read cache is less than that of the page cache, the page cache dominates, then the least recently used item in the fine-grained read cache should be evicted when the shared memory has no spare space (solution 1 in §3.2.1). If the fine-grained read cache has a hit ratio greater than or equal to that of the page cache, then the solution of migrating slab (solution 2 in §3.2.1) is preferred when the shared memory is exhausted.

4 EVALUATION

4.1 Experimental Setup

We implement host-side components of Pipette as a kernel module based on Ext4 file system in the Linux kernel v5.4.0. A new file open flag, `O_FINE_GRAINED`, is introduced to use the proposed read path selectively. The implementation supports fine-grained reads by calling POSIX `read()` system call. We also modify the VFS and file system layer to support the interactions between the host and the device. Our code is available at https://github.com/MIoTLab/Pipette.



Item	Description
Host Interface	PCIe Gen.3 x 4
Protocol	NVMe 1.2
Module Capacity	477GB
SSD Architecture	8 channels 8 ways 2 cores
Storage Medium	SLC NAND flash MLC NAND flash TLC NAND flash
Mapping Region	64MB
Max DDR size	4GB

Figure 5: Pipette Hardware Prototype and Specification.

We develop a prototype based on the YS9203 NVMe SSD development platform [13]. The details are listed in Figure 5. We implement Pipette by modifying the firmware. The HMB mapping region is remapped to the host-side fine-grained read cache through DMA mapping to achieve a byte-addressable interface. We also extend the NVMe command set to support fine-grained reads.

We compare Pipette with conventional block I/O and the state-of-the-art fine-grained read approach, 2B-SSD [3], which supports two read modes through MMIO and DMA (denoted as 2B-SSD MMIO and 2B-SSD DMA, respectively). We also compare Pipette with and without fine-grained read cache (denoted as Pipette w/o cache). We report the throughput normalized to block I/O and the I/O traffic on read operations under synthetic workloads and real applications.

4.2 Results under Synthetic Workloads

Five synthetic workloads are constructed by varying the large and small read ratio, as listed in Table 1. We perform 2.5 million read requests using synthetic workloads. File offset of each request is chosen according to the uniform and zipfian distribution. When using a uniform distribution, the file offset is chosen uniformly at random. When using a zipfian distribution, some offsets will be extremely popular (the head of the distribution) while others will be unpopular (the tail), indicating a better reuse pattern.

Under read requests with uniform distribution, Pipette achieves a higher throughput as the ratio of small reads grows, as shown in Figure 6. For pure fine-grained read workload E, it gains 31.2× performance benefit compared to the block I/O. Moreover, for pure large read workload A, Pipette causes negligible overhead. Pipette achieves a significant throughput enhancement mainly because of its caching mechanism. The fine-grained read cache adaptively holds data with high reuse, preventing DRAM from polluting by data with low reuse. Moreover, it could store tiny data compactly,

Table 1: Synthetic Workloads[1, 2].

Workload	A	B	C	D	E
Large/small read ratio	100/0	90/10	50/50	10/90	0/100

[1] Default small read size: 128B, large read size: 4096B.
[2] Request distribution: uniform random and zipfian (α=0.8).

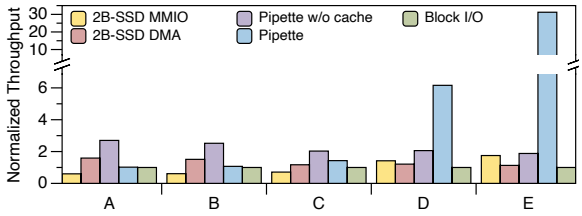


Figure 6: Normalized Throughput of Synthetic Workloads with Uniform Distribution.

providing more memory for subsequent highly-reused data. On the other hand, page cache promotes every accessed data in page granularity, resulting in a frequent data movement in the poor data-reuse situation, as is the case with uniform random distribution. Further, the read-ahead strategy within the block I/O intensifies the access to the device. Pipette w/o cache and 2B-SSD, which we collectively refer to as no-cache approaches, slightly improve the performance because the prefetching overhead is not introduced. Besides, 2B-SSD MMIO suffers a performance degradation when the percentage of large read increases. This is because MMIO is a non-posted transaction that requires waiting for a read completion and is split into small-size (up to 8 bytes on current x86 CPUs) read transactions to guarantee atomicity [3].

Table 2 shows the I/O traffic for workloads with uniform random distribution. Pipette drastically reduces the I/O traffic due to the dedicated fine-grained read cache and the temporal locality of workloads. With more large reads issue, more block I/Os involved, its read-ahead strategy and page-level migration result in an increment of data transfer, thus gradually increasing Pipette's I/O traffic. Since the location distribution, instead of size distribution, determines which pages are read, block I/O has the same I/O traffic in all five workloads. No-cache approaches only transfer needed data by leveraging byte accessibility; their I/O traffic is identical and lower than the block I/O when small reads dominate. For pure small read workload E, I/O traffic of the block I/O is 9.4× larger than that of no-cache approaches due to the poor spatial locality of synthetic workloads with uniform random distribution.

Figure 7 plots the measured throughput of workloads with zipfian distribution. For workload A which large reads dominate, Pipette achieves throughput comparable to that of the block I/O, meaning that the proposed fine-grained read framework will not degrade the performance of the traditional block I/O. As the small read ratio increases, Pipette shows increased improvement of throughput, which is from 1.1× to 1.4×. Because workloads with zipfian

Table 2: I/O Traffic of Synthetic Workloads with Uniform Distribution (MB).

	A	B	C	D	E
Block I/O	2973.6	2973.6	2973.6	2973.6	2973.6
2B-SSD MMIO	9765.6	8819.6	5035.4	1251.2	305.2
2B-SSD DMA	9765.6	8819.6	5035.4	1251.2	305.2
Pipette w/o cache	9765.6	8819.6	5035.4	1251.2	305.2
Pipette	2973.6	2678.4	1479.7	313.45	79.8

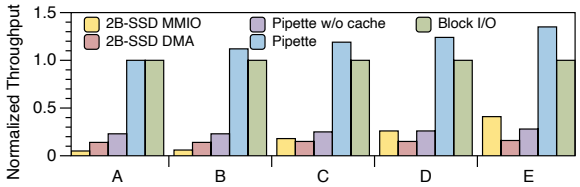


Figure 7: Normalized Throughput of Synthetic Workloads with Zipfian Distribution.

distribution preserve certain levels of spatial locality, page cache gains benefits from its advantages. Thus, Pipette has a smaller optimization space. No-cache approaches have throughput similar to the previous experimental result: in the case of more small reads, 2B-SSD MMIO has the best performance of the three due to its fast speed of reading tiny data, and 2B-SSD DMA gets the worst performance because of its software overhead; in the case of more large reads, 2B-SSD MMIO turns to the worst performance.

Under read requests with zipfian distribution, block I/O transfers 2.5× amount of data in the case of pure small read workload E, as shown in Table 3. Compared to workloads with uniform random distribution, block I/O has less I/O traffic increment since the data reuse pattern from zipfian distribution and the read-ahead strategy increase its chance of getting data directly from the page cache.

Under read-intensive workload E with uniform distribution, we now compare read latencies of Pipette with block I/O and other fine-grained read approaches and report average read latencies of varying request sizes from 8 bytes to 4KB in Figure 8. Except for 2B-SSD MMIO, each approach performs stably for different read sizes. Pipette achieves a read latency of about 2µs thanks to the fine-grained caching mechanism. It shows 33.8× shorter latencies than block I/O since transparently exploiting both the byte accessibility and the fast DRAM access. Due to the hardware limitation that cannot synchronously read data from parallel channels, block I/O

Table 3: I/O Traffic of Synthetic Workloads with Zipfian Distribution (MB).

	A	B	C	D	E
Block I/O	748.3	748.3	748.3	748.3	748.3
2B-SSD MMIO	9765.6	8819.6	5035.4	1251.2	305.2
2B-SSD DMA	9765.6	8819.6	5035.4	1251.2	305.2
Pipette w/o cache	9765.6	8819.6	5035.4	1251.2	305.2
Pipette	748.3	684.2	399.9	107.0	33.3

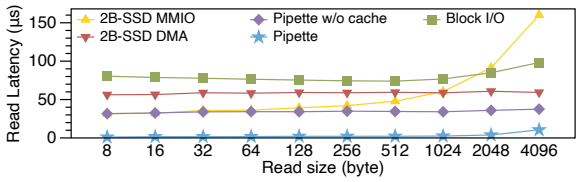


Figure 8: Read Latency of Workload E of Varying Size with Uniform Distribution.

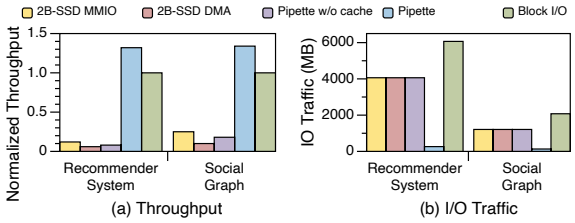


Figure 9: Real-world Applications Results.

suffers 14.56-38.89µs slower than 2B-SSD DMA. As for 2B-SSD DMA, the time-consuming per-access DMA mapping introduces significant latency, making it 21.79-25.06µs slower than Pipette w/o cache. Besides, the latency of 2B-SSD MMIO increases in proportion to request sizes. It consumes much more time than Pipette w/o cache and 2B-SSD DMA at a read request size of approximately 32 bytes and 1KB, respectively.

4.3 Results under Real-world Applications

We now demonstrate the benefit of Pipette for typical real-world applications: recommendation system and social graph. We use deep learning-based recommendation model [8] in our experiments. It handles sparse input features by looking up fixed-sized (128B) embeddings from the 4.1GB tables inside the SSD [5, 11]. We use the graph and requests based on LinkBench's default setting [2].

Pipette outperforms the block I/O by 1.3× in throughput for both recommendation system and social graph, as shown in Figure 9(a). It gains performance because lots of random accesses hit the fine-grained read cache. The efficiency of the cache is also shown from the low throughput of the no-cache approaches, that is, 2B-SSD and Pipette w/o cache. In the case of Pipette, fewer pages are moved between the SSD and the host DRAM and only demanded data is transferred and stored. Due to the read-ahead prefetching and the page-level promotion, block I/O transfers a tremendous amount of data and stores redundant parts of pages in the page cache. As shown in Figure 9(b) and Table 4, Pipette reduces data movement and memory usage by orders of magnitude compared to the no-cache and block I/O solutions. No-cache solutions transfer data in byte granularity, avoiding extra I/O traffic caused by unnecessary data, thus having comparable I/O traffic. Further, Pipette has significantly less data movement than block I/O and no-cache solutions because its caching mechanism provides a high hit ratio, 93.5% for recommendation system and 89.09% for social graph as shown in Table 4, enabling more data to be accessed from the cache.

5 CONCLUSION

This paper proposes Pipette framework, a fine-grained read path including adaptive multi-granularity host-side read cache, as an extension to the traditional I/O framework. The proposed framework Pipette could not only offer a tremendous saving in I/O traffic but also provide a significant performance gain for fine-grained reads. A prototype of Pipette is implemented with Ext4 file system on an SSD development platform. Experimental results show that Pipette can improve I/O throughput by 31.6% and 33.5% for real-world recommendation system and social graph, respectively.

Table 4: Page Cache vs. Fine-Grained Read Cache.

	Recommender System		Social Graph	
	Hit Ratio (%)	Memory Usage (MB)	Hit Ratio (%)	Memory Usage (MB)
Block I/O	64.50%	2382	66.50%	1112
Pipette	93.50%	91	89.09%	70

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61821003, No. U2001203, No. 61872413, No. 61902137, and by the Research Grants Council of the Hong Kong Special Administrative Region, China under Grant No. CityU 11217020, No. CityU 11218720.

REFERENCES

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence, RI, USA, 971–985.

[2] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1185–1196.

[3] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE, Los Angeles, California, 425–438.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference*. USENIX Association, San Jose, CA, USA, 49–60.

[5] Criteo. 2014. *Kaggle Display Advertising Challenge Dataset*. Criteo. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[6] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems*, Vol. 1. Systems and Machine Learning Foundation, Palo Alto, CA, USA, 40–52.

[7] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *47th ACM/IEEE Annual International Symposium on Computer Architecture*. IEEE, Valencia, Spain, 982–995.

[8] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture*. IEEE, San Diego, CA, USA, 488–501.

[9] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. Read as Needed: Building WiSER, a Flash-Optimized Search Engine. In *18th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, USA, 59–73.

[10] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. Improving SSD Lifetime with Byte-Addressable Metadata. In *Proceedings of the International Symposium on Memory Systems*. ACM, New York, NY, USA, 374–384.

[11] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. FlashEmbedding: Storing Embedding Tables in SSD for Large-Scale Recommender Systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, New York, NY, USA, 9–16.

[12] Zhe Yang, Youyou Lu, Erci Xu, and Jiwu Shu. 2020. CoinPurse: A Device-Assisted File System with Dual Interfaces. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. IEEE, Virtual Event, USA, Article 100, 6 pages.

[13] YEESTOR. 2021. *Yeestor YS9203 PCIe SSD Memory Controller*. YEESTOR Microelectronics. http://www.yeestor.com/