PIPETTE: Efficient Fine-Grained Reads for SSDs

Shuhan Bai, *Member, IEEE*, Hu Wan, *Member, IEEE*, Yun Huang, *Member, IEEE*, Xuan Sun, *Member, IEEE*, Fei Wu, *Member, IEEE*, Changsheng Xie, *Member, IEEE*, Hung-Chih Hsieh, Tei-Wei Kuo, *Fellow, IEEE*, and Chun Jason Xue, *Senior Member, IEEE*

***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 upon the traditional storage system rely on the paging mechanism to migrate pages to the host DRAM, tending to suffer from these fine-grained read operations in terms of I/O traffic as well as performance. Motivated by this challenge, an efficient fine-grained read framework, Pipette, is proposed in this article as an extension to the traditional I/O framework. With adaptive design for caching, merging, and scheduling, Pipette explores locality and acceleration for fine-grained read requests to establish an efficient byte-granular read path upon the dedicated byte-addressable interface. When the Pipette prototype on an SSD runs popular workloads, we measured throughput gains by up to 50% and 54% with traffic reduction in the range of 41.3× and 56.5×.**

***Index Terms*—File system, fine-grained reads, solid-state drive.**

I. INTRODUCTION

FINE-GRAINED reads, each a few hundred bytes or less, are prevalent in many large-scale big data applications, such as recommendation system [1], [2], social network [3], and search engine [4]. For example, recommendation system handles sparse input features by looking up embedding vectors from SSDs [5]. The typical size of an embedding vector is 128 B [6], while the minimal read granularity of block storage devices is often at 4 kB, which is highly unfit for tiny accesses.

Manuscript received 28 September 2022; revised 26 January 2023 and 14 April 2023; accepted 22 April 2023. Date of publication 15 May 2023; date of current version 22 November 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 61821003, Grant U2001203, Grant 61872413, and Grant 61902137; and in part by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Grant CityU 11217020 and Grant CityU 11218720. This article was recommended by Associate Editor F. Conti. *(Corresponding author: Fei Wu.)*

Shuhan Bai is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: shuhanbai2-c@my.cityu.edu.hk).

Hu Wan, Yun Huang, and Chun Jason Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: hu.wan@my.cityu.edu.hk; yhuang432-c@my.cityu.edu.hk; jasonxue@cityu.edu.hk).

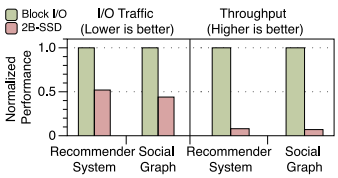
Xuan Sun is with the Department of Computing, Imperial College London, SW7 2BX London, U.K. (e-mail: xuan.sun@imperial.ac.uk).

Fei Wu and Changsheng Xie are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: wufei@mail.hust.edu.cn; cs_xie@hust.edu.cn).

Hung-Chih Hsieh is with the Department of Research and Development, YEESTOR Microelectronics Company Ltd., Shenzhen 518057, China (e-mail: obi.hsieh@yeestor.com).

Tei-Wei Kuo is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan, and also with the Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi, UAE (e-mail: ktw@csie.ntu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2023.3276520



Fine-grained reads are prevalent!

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

Fig. 1. Fine-grained reads dominated applications [1], [7].

Fixed interfaces ease development but lead to severe performance degradation when fine-grained accesses are required. The main reason is that fine-grained reads are incompatible with the paging mechanism and read-ahead strategy that are widely used for bridging block-oriented devices in the traditional storage system. Non-negligible performance overhead is incurred by enforcing accessing and managing data in fixed granularity via the block-based interface. First, page-level migrations triggered by byte-granular requests induce a tremendous amount of extra I/O traffic and 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. Second, the paging mechanism faces pollution and thrashing problems caused by fine-grained reads dominated applications whose working set sizes are significantly larger than the available DRAM capacity. Third, the read-ahead strategy further incurs performance deterioration. The traditional storage system with it consumes an enormous amount of memory handling spatial locality, leaving suboptimal utilization of temporal locality. As described, accessing, prefetching, and managing data in page granularity for fine-grained reads inevitably carries additional transferring and managing costs. Appropriate granularity and mechanisms for data access and management with various characteristics are necessary to avoid performance and data transfer overhead.

To relieve performance overhead for fine-grained accesses upon block I/O interface, the state-of-the-art approaches leverage the byte-accessibility of SSDs [8], [9], [10], [11], [12]. 2B-SSD [10] proposes a dual-interface SSD that allows accessing the device in both byte and block granularity. Fig. 1 shows the normalized I/O traffic and throughput of 2B-SSD compared to block I/O for the recommendation system and social graph. Although 2B-SSD reduces read amplification, it suffers from poor read performance. Three reasons introduce the overhead. First, enabling byte-addressability of SSDs by exposing device internal memory as the extended mappable region to the host requires on-chip persistent memory, increasing management complexity. Second, 2B-SSD requires an address translation for accesses through the memory interface. These operations

are frequently conducted on the critical path and contribute to user-perceived latency. Third, 2B-SSD bypasses the I/O stack and ignores data locality. For temporal locality, it cannot benefit from the fast DRAM access by promoting hot data. Besides, it prevents devices from exploiting internal parallelism since there is no design for I/O scheduling to rethink spatial locality.

This article proposes Pipette, a practical fine-grained read framework concurrent with the traditional block-based framework. It exploits adaptive caching mechanisms and best-effort merging and scheduling strategies for fine-grained requests upon the byte-accessibility of SSDs to provide an efficient fine-grained read path, significantly reducing read amplification while achieving high read performance. Pipette makes use of the physical resources between the SSD and the host DRAM in both byte and block granularity. Thanks to the byte-accessibility provided by the PCIe interconnection, it is feasible to build a byte-addressable interface by establishing direct memory access (DMA) mapping to make programs access data across the SSD and the host DRAM seamlessly with negligible overhead.

Accessing the SSD for every read request is slower than accessing the host DRAM. Pipette treats the host-assigned memory as a fine-grained read cache with adaptive caching mechanisms to determine which fine-grained data should be included or reclaimed. This orchestrated cache dynamically and compactly adapts to different data sizes and maintains efficient memory utilization. Applications can transparently exploit advantages of both the byte-addressable SSD and main memory. We support promoting frequently accessed data to the host DRAM for fast access, benefiting from temporal locality. Note that the promotion granularity is byte. Such a flexible granularity can reduce read amplification and I/O traffic by minimizing unnecessary transferring overhead while keeping the host DRAM from pollution and thrashing problems. Besides, a dynamic allocation strategy is designed to balance the memory usage of the page cache and the fine-grained read cache, which is aided by an adaptive reassignment strategy to improve performance for different access patterns.

Furthermore, to improve system responsiveness, Pipette regroups and schedules requests with best effort based on workload access patterns and available processing resources. Its design is based on two ideas. First, we observed that logical neighboring pages are likely to be independently accessed in parallel inside the SSD. Pipette reorders and groups requests with related logical addresses into large batches to exploit device internal parallelism and workloads' spatial locality. Second, Pipette minimizes I/O waiting time by elegant scheduling of batched requests. A stall-free submission mechanism is adopted to ensure request execution without device pausing, maximizing the utilization of device processing capacity. These ideas help reshape I/O patterns to be friendly to workloads' spatial locality and device internal resources.

We implement the Pipette prototype on a real SSD development platform and evaluate it under various synthetic and real workloads. Our contributions are summarized as follows.

1) We present Pipette, a fine-grained read framework, as an extension to the traditional I/O framework, which achieves good read performance with notably lower read amplification and I/O traffic.
2) 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.
3) We introduce a best-effort fine-grained I/O merging and scheduling scheme that combines read requests that access logically neighboring data and schedules batched requests with minimum I/O waiting time, profiting from the spatial locality, device internal resources and concurrency.
4) We provide evaluations of Pipette to show the throughput gain by up to 50% and 54% for recommendation system and social graph with significant traffic reduction.

The original implementation of Pipette was presented in a previous conference paper [12], which only handled small synchronous reads issued with one thread. Parallel reads, which occur when the process issues multiple requests under concurrency or multithread, are not supported. This article significantly extends our preliminary work in several aspects.

1) We extend Pipette, our proposed collaborative full-stack framework for fine-grained reads, to support an arbitrary number of worker threads, moving the focus toward the parallel architecture and its scalability.
2) We optimize adaptive caching mechanisms with a periodic reset strategy to mitigate the slow perception of access pattern changes by periodically saving and comparing access patterns and resetting associated parameters.
3) We propose a best-effort fine-grained I/O merging scheme to aggregate tiny reads that access adjacent logical blocks. It achieves better system responsiveness since data spatial locality and device-level parallelism can be used to reduce time overhead caused by intradevice data transfer.
4) We present a stall-free submission mechanism for scheduling to pursue nonstop request execution, maximizing the utilization of device processing capacity.
5) We implement and evaluate the Pipette prototype with parallel architecture on a platform offering a larger degree of parallelism. Workloads with more diverse data sizes and heterogeneity are employed under multiple threads.

This article is organized as follows. In Section II, we discuss related works and identify the key factors that make our proposal novel. Section III provides needed theoretical background. Pipette's design and implementation are introduced in Section IV with experimental evaluations in Section V. We present our conclusions in Section VI.

II. Related Work

It is well known that the high-level abstraction of standard block I/O interface limits the utilization of devices and the performance of applications. Academia and industry have long recognized its overhead and attempted to bridge this gap with the flexible storage interface for SSDs [8], [9], [10], [11], [12], [13], [14].

Three pieces of prior works are comparable to ours in that they enable the byte-addressability of PCIe interconnection for data access to remove extra transferring overhead. 2B-SSD [10] exposes persistent memory and allows applications

to access it directly using memory instructions, bypassing the block I/O stack without considering data locality for better performance. Moreover, it employs an address translation unit to enable byte-accessibility by redirecting memory instructions into the persistent memory. In contrast, Pipette provides a portion of host-side memory resources, whose mapping is established with device-side address range at the initialization stage, to cache frequently accessed data and gain benefits from the faster DARM access and temporal locality without address translation overhead. Besides, a module for I/O merging and scheduling is added to take spatial locality and device internal parallelism into account. Pipette can directly access the SSD without paging mechanism as well as achieve good performance in handling data locality. FlatFlash [9] leverages the unified address translation mechanism in [14] to reduce the address translation overhead and introduces a scheme for promoting hot pages to relieve duplicate data movement further. Note that the promotion granularity is page, unnecessary data inside pages still generates a tremendous amount of extra I/O traffic. Pipette organizes the host mapping region into slab structure with multigranularity items to compactly accommodate tiny data of different sizes, fully utilizing precious memory resources and minimizing redundant data transfer. CoinPurse [11] encapsulates direct accesses to the device in the form of system calls to optimize partial updates only, leaving fine-grained reads unexplored. On the other side, Pipette proposes a top-down fine-grained read framework, assisted with adaptive mechanisms for caching, merging and scheduling, as an extension to the traditional I/O stack to enable efficient multigranular reads for SSDs.

As for the initial Pipette [12], it handles synchronous fine-grained reads under a single thread, leaving optimization space to rethink system scalability and parallel reads. Pipette with parallel architecture comprehensively considers spatial locality and device processing resources to improve system responsiveness with best-effort merging and stall-free scheduling.

III. Background

A. Conventional Block-Based Read Path

The current I/O stack for NVMe SSDs requires many operations to service a single read request through multiple independent layers, as shown by the dotted box in Fig. 2. For example, when a few bytes are requested from SSDs, the application issues a buffered read I/O, which first falls into the virtual file system layer to manage the information maintained for entry into the page cache layer. Demanded data will be returned to the user space directly when a page cache hit occurs. Upon a cache miss, memory pages are allocated for storing missing file blocks while conducting read-ahead. The file system retrieves logical block addresses (LBAs) of missing pages and issues a block request to the underlying block layer. In the block layer, associated read requests are inserted into request queues, where I/O merging and scheduling are performed [15]. The request is then dispatched to the device driver layer, which generates the corresponding I/O command and sends it to an NVMe submission queue. The I/O command is received and handled by the storage device. Finally, the host receives an interruption from the SSD when the fine-grained read request is done, and the requested and read-ahead pages are read into the page cache, even if a few bytes are demanded.

The traditional framework performs well in handling spatial locality for coarse-grained reads due to its read-ahead and paging mechanism. However, page cache faces underutilization problem caused by fine-grained reads scattered around, leading to large amounts of redundant data polluting memory. Moreover, the cost of page migration is exacerbated in both I/O traffic and amplification for tiny accesses since the fixed interface enforces transferring data in the unit of block.

B. Byte-Addressable Interface for SSDs

Being able to access devices in multigranularity can eliminate the cost of page migration caused by the block-based interface. With the existing bus/interface standard for SSDs, i.e., PCIe (or NVMe) [16], [17], CPUs are capable of performing memory-like accesses to peripheral devices in byte granularity by utilizing the available base address registers (BARs) in the SSD controller through the memory-mapped I/O (MMIO) [9], [10], [11]. Since NAND flash chips have to be accessed in page granularity, we can leverage the DRAM inside SSDs, which is used to cache user data and store metadata for the flash translation layer (FTL), to service the CPU's memory requests. NVMe protocol defines controller memory buffer (CMB), a general-purpose region of device controller memory exposed as part of a PCIe BAR for SSDs, to notify the host to add this extended mappable region at the system reset stage. Note that the host-assigned address range is only used for address decoding and consumes no physical memory. The device is responsible for linking the address space exposed to the CPU and the memory space in the SSD, i.e., CMB. When MMIO is issued to the specified BAR's memory range, a memory read/write to it will be redirected to the on-device DRAM.

Such a CMB-based byte-addressable interface handles fine-grained reads in the following fashion: SSD controller reads whole pages from flash chips to the CMB, invokes address translation or sets mapping for DMA, and then transfers fine-grained data to the host through MMIO or DMA (an approach that allows certain hardware subsystems to access the main memory independently of the CPU) [10].

Since no caching is supported between the host and SSDs, the above interconnect scheme for realizing byte-granular accesses cannot cache hot data in memory to gain performance benefits. Although high-reusable pages can be cached inside the CMB, on-chip persistent memory is required. Also, the page granularity of data promotion causes device-side pollution or thrashing problems with high management efforts. Besides, the preparation overhead of frequent address translation required before every access cannot be ignored.

IV. Pipette Design

Pipette's architecture is illustrated in Fig. 2. Given an I/O interconnect that allows byte accesses, Pipette superimposes byte-accessibility (Section IV-A) on SSDs to provide an

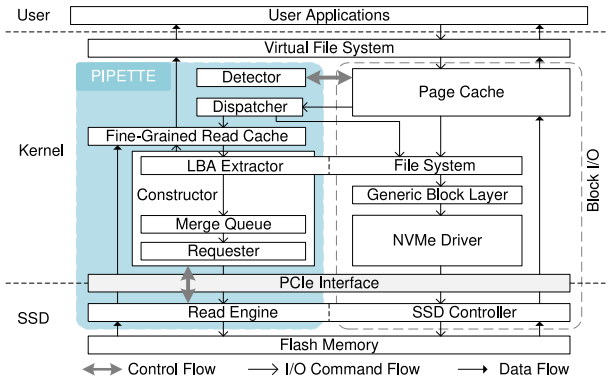


Fig. 2. Pipette architecture.

efficient read path (Section IV-B) with adaptive caching mechanisms (Section IV-C) and best-effort merging and scheduling strategies (Section IV-D).

A. Enabling Byte-Accessibility of SSDs

NVMe standard provides the host memory buffer (HMB) feature for the host to allocate host memory resources requested by the device controller for exclusive use [17]. During system boot time, the BIOS and OS probe PCIe devices, like SSDs, and initialize them to an active state. Upon the PCI enumeration stage after the system reset, the host identifies the information indicated by the controller to create the HMB region within the given limits to avoid incomplete utilization. Afterward, a descriptor list that describes a set of host memory address ranges for vendor-specific use is supplied to the device controller. The end-point device is responsible for establishing DMA mapping that links a portion of internal controller resources and the host-assigned memory space according to the received descriptor list while enabling the HMB feature at the initialization stage. Once completed, there is no overhead for data transferring preparations like address translation.

With PCIe MMIO, all memory requests are allowed to be directly issued to the device. Note that an MMIO read is split into small-size nonposted transactions, which require a bus transaction to respond with a read completion response to indicate the success or failure of the transaction. Taking the round trip delay caused by waiting for a read completion into account, we leverage DMA to offload the expensive memory operations. Moreover, since the dedicated DMA facility provides hardware independence, it can further accelerate memory-like reads by concurrency and the reduction of on-device CPU cycles needed to transfer data. Wherefore, applications may map the HMB region into their user space via `mmap()` system call and then retrieve needed data by accessing the virtual memory address, which will be finally redirected to the corresponding device-side mapping range.

After initialization, Pipette enables the host CPU to seamlessly access data in byte granularity across the SSD and the host memory. Upon this proposed HMB-based byte-addressable interface, 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.

Unlike existing works that leverage CMB to enable a byte-addressable interface, Pipette leverages HMB to realize the underlying I/O interconnect with byte accessibility. Compared to CMB, HMB supports cache coherence between the host machine and SSD. With the inherent hardware resources and high-performance CPU on the host side, fine-grained read operations can be accelerated without increasing the complexity of programming and management. The frequently accessed data can be promoted and cached thanks to the mappable resources provided by the host (Fine-Grained Read Cache in Fig. 2), achieving good improvement by exploiting temporal locality. For better performance and memory utilization, we organize fine-grained read cache in multigranularity assisted with several adaptive caching mechanisms (see Section IV-C).

B. 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, as demonstrated in Fig. 2. The details of the host-device interaction under the Pipette framework are shown in Fig. 3.

1) 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.

Once the application opens a file that serves fine-grained reads for the first time, indicated by a particular file flag `O_FINE_GRAINED`, a per-file hash lookup table (as shown by the bottom left corner in Fig. 4) is created to record basic information (location, size, reference count, etc.) and management information (for maintaining LRU lists and the hash table) about fine-grained read requests toward that file.

A fine-grained read request goes through the virtual file system layer and is first performed by the page cache while access counts of it and the fine-grained read cache are recorded. Upon a page cache hit, Pipette increases the page cache's hit count and calculates its hit ratio, and then the requested data will be read from the hit page. If a cache miss occurs, **Fine-Grained Access Detector** is triggered (see **Detector** in Fig. 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 Fig. 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, to retrieve whether the required page range is cached in the host DRAM. Pipette first updates the reuse ratio of the fine-grained data and the reference threshold for filtering cold data. Upon a tiny read hit, the hit ratio of the fine-grained read cache is updated and the status of the hit data is checked. The hit data will be returned to the user if it is not outdated. Instead, when a write operation has rewritten the corresponding raw data, the up-to-date data will be retrieved from the flash memory through the byte-granular datapath (see

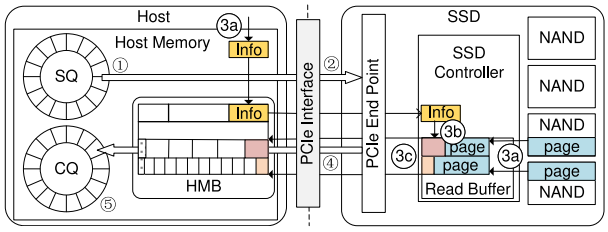


Fig. 3. Fine-grained read flow of read engine.

Section IV-B2). 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 Fig. 2. The constructor requests **LBA Extractor**, a file system extension for fine-grained reads that bypasses the generic block layer, to retrieve LBAs of pages that include needed data and notifies the **Merge Queue** to receive and manage fine-grained reads. According to the status of the submission queues (see **SQ** in Fig. 3), the merge queue either conducts I/O merging or informs the **Requester** to submit the reconstructed read request directly to the SSD. Suppose unprocessed requests are waiting in the submission queues. In that case, i.e., busy status, the newly arriving fine-grained read is stalled inside the merge queue and merged with other in-queue requests that access adjacent data within the device. Once the device completes a request, the earliest queued merged request is sent to the device for processing. The merged read request notifies the device that a batch of fine-grained reads has been issued. Still, it merely sends the start LBA of associated continuous pages, the number of pages and the number of fine-grained reads to allow the device to read several whole pages in parallel first. Instead, when all of the submission queues are empty and remain in idle status, the upcoming fine-grained read transforms itself into a request and is sent directly to the device by the requester. The reconstructed read request informs the device that a fine-grained read is arriving, but only the LBA of the related page is sent to get the device ready to read the entire page first.

While the device is reading pages from flash chips, the insertion logic of the **Fine-Grained Read Cache** is switching on or, when multithreaded, the read flow of other fine-grained reads is in progress. We continue with the former to introduce the following process. The fine-grained read cache allocates space for the small read according to the recorded access size and inserts a new entry into the corresponding hash table to contain associated information about data extraction and transfer. This is the function illustrated by the I/O command flow between **Constructor** and **Fine-Grained Read Cache** in Fig. 2. Afterward, Pipette maintains these related parameters in an Info item (see host-side step 3a in Fig. 3) and notifies the device to assist it in performing subsequent operations.

Fine-Grained Read Engine is invoked when the device receives the reconstructed read request or the merged read request (steps 1 and 2 in Fig. 3). Although the HMB-based byte-addressable interface supports data access in byte granularity, NAND flash chips have to be accessed in page. A portion of available DRAM inside the SSD is leveraged as a read buffer to provide the necessary bridge between the byte-accessible interconnect and flash chips. Further, Pipette treats this read buffer as an SSD-Cache that holds frequently accessed pages for better performance. As illustrated in device-side step 3 in Fig. 3, the engine processes fine-grained reads in three phases: 1) load pages to the preallocated read buffer (step 3a); 2) consume Info items to get host-assigned destination addresses, which are allocated synchronously when full flash pages are read on the device side, and other information about data extraction (step 3b); and 3) extract all access ranges (step 3c). When the fine-grained read request completes, the engine returns demanded data to the fine-grained read cache (step 4) with a read completion (step 5) to notice the host of the arriving new data (see data flow between **Read Engine** and **Fine-Grained Read Cache** in Fig. 2).

2) 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 them 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 set an updated tag to the found item. Once a hit item in the fine-grained read cache is marked as updated, the requested data is retrieved from the flash chips again. In this way, subsequent read requests will either get the updated data from the page cache or the latest data from the flash memory without affecting the promotion of the relevant data. Moreover, when reads and writes of the same data happen simultaneously, the built-in page lock state can ensure the correctness of the read data. For an n-byte write operation, we denote the length of link lists led by hash heads corresponding to its n offsets as $m_1, \ldots, m_n$, and the worst- and best-case time complexity for ensuring data consistency is $O(\sum_{i=1}^{n} m_i)$ and $O(n)$.

C. 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 (Section IV-C1) is introduced to facilitate effective caching for tiny reads in byte granularity with several caching mechanisms (Sections IV-C2–IV-C4). We present the procedure of reading data upon dual interfaces with the adaptive caching scheme in Algorithm 1.

1) Cache Organization and Management: The data layout of the fine-grained read cache is illustrated in Fig. 4. It consists of three partitions, TempBuf Area, Info Area, and Data Area, and their detailed design is shown below.

TempBuf Area is a temporary buffer for low-reuse data to avoid polluting precious memory. Fine-grained data accessed for the first time or the first few times will be temporarily held

Algorithm 1 Read Data Upon Dual Interfaces With Adaptive Caching Mechanisms

Variables:

$NormalAccessCnt, AccessCnt, ReuseAccessCnt, PageCacheHitCnt, CacheHitCnt, ReuseCnt \leftarrow 0$
$PageCacheHitRatio, HitRatio, ReuseRatio \leftarrow 0, LwRatio \leftarrow 0.25, HiRatio \leftarrow 0.75$
$ResetCnt \leftarrow 10k, EpochReuseCnt, EpochReuseCnt\_old, EpochReuseRatio, EpochReuseRatio\_old \leftarrow 0$
$CurrThreshold \leftarrow 2, MaxThreshold \leftarrow 5, MinThreshold \leftarrow 2, refCnt \leftarrow 0$

if fine-grained read **then**
  $AccessCnt++,\ ReuseAccessCnt++$
else
  $NormalAccessCnt++$
end if
if page cache hit **then**
  if fine-grained read **then**
    $ReuseCnt++$
  end if
  $PageCacheHitCnt++,\ PageCacheHitRatio \leftarrow PageCacheHitCnt/(NormalAccessCnt + AccessCnt)$
  return data from the page cache
end if
if fine-grained read && per-file hashtable hit **then**
  $refCnt++,\ ReuseCnt++,\ ReuseRatio \leftarrow ReuseCnt/ReuseAccessCnt$
  /* Adaptive Caching Mechanism */
  if $ReuseAccessCnt != 0$ && $ReuseAccessCnt \ \%\ ResetCnt = 0$ **then**
    $EpochReuseCnt\_old \leftarrow EpochReuseCnt,\ EpochReuseCnt \leftarrow ReuseCnt$
    $EpochReuseRatio\_old \leftarrow EpochReuseRatio$
    $EpochReuseRatio \leftarrow (EpochReuseCnt - EpochReuseCnt\_old)/ResetCnt$
    if $|\ EpochReuseRatio - EpochReuseRatio\_old\ | \geq 0.5$ **then**
      $ReuseRatio \leftarrow EpochReuseRatio,\ ReuseAccessCnt \leftarrow ResetCnt$
      $ReuseCnt \leftarrow EpochReuseCnt - EpochReuseCnt\_old$
    end if
  end if
  if $ReuseRatio < LwRatio$ && $CurrThreshold < MaxThreshold$ **then**
    $CurrThreshold++$
  else if $ReuseRatio > HiRatio$ && $CurrThreshold > MinThreshold$ **then**
    $CurrThreshold--$
  end if
  if fine-grained cache hit **then**
    $CacheHitCnt++,\ HitRatio \leftarrow CacheHitCnt/AccessCnt$
    return data from the fine-grained cache
    /* Adaptive Caching Mechanism */
  else if $refCnt < CurrThreshold$ **then**
    store data in TempBuf
    return data from TempBuf
  else if $refCnt \geq CurrThreshold$ **then**
    /* Dynamic Allocation Strategy */
    if fine-grained cache is full **then**
      if $HitRatio < PageCacheHitRatio$ **then**
        remove least recently used item
      else
        reassign
      end if
    end if
    store data in the fine-grained cache
    return data from the fine-grained cache
  end if
end if

inside the TempBuf Area by Pipette as infrequent accesses, preventing the host DRAM from thrashing and pollution problems for data-intensive applications. A current position is collected and managed to synchronize the allocation of the temporary destination address in the case of multiple threads.

Info Area is jointly managed by the host and device to maintain the recorded information (see **Record Info** in Fig. 4) of read requests. Each Record Info delivers sufficient information to handle a fine-grained read request or a merged read request representing a batch of tiny reads located in neighboring pages. A composite record corresponds to a read request of data segment(s) resident in (consecutive) LBA(s), so the number of accessed pages (see **LBA number**) and the number of tiny reads related to the read request (see **Record number**) are

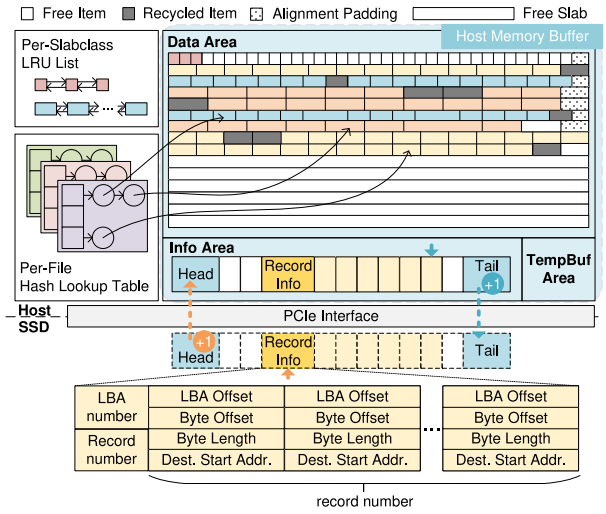


Fig. 4. Fine-grained read cache overview.

contained in the Record Info. Besides, metadata of individual fine-grained reads (i.e., records) that make up the read request are held, including the page index (see **LBA Offset**), read range (see **Byte Offset** and **Byte Length**), and destination address (see **Dest. Start Addr.**). To manage the consumption and generation of Record Infos, the Info Area is organized as a circular queue of composite records with head and tail pointers controlled by the device and the host, respectively. On the one hand, the host increases the tail's value and appends a Record Info (see the blue arrow in Fig. 4) whenever a read request is issued to the device. On the other hand, the device raises the head's value and consumes a Record Info (see the orange arrow in Fig. 4) at the end of a fine-grained read procedure.

Data Area caches byte-granular data in an orchestrated structure that considers various data sizes. Inspired by the Linux buddy system and kernel slab management [18], it organizes memory into uniformly sized slabs, and each slab contains predivided items with the same capacity. Slabs are classified into different classes according to the capacity of contained items, which can be set flexibly. Pipette stores promoted data in the smallest possible slab class that can accommodate the data size completely. An LRU list is maintained for each slab class to arrange items holding data in access order. If no free space remains in the corresponding slab class, one free slab will be requested. Once Pipette can no longer allocate free memory from the shared region, two techniques are adopted according to the dynamic allocation strategy (see Section IV-C4 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 and 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 this slab migration.

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 (lines 33–48 in Algorithm 1).

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 and dynamically adjust it to adapt to various access patterns, so that data can be promoted frequently when there is high data reuse and infrequently in the case of low data reuse. To detect and adapt to different data reuse patterns, Pipette maintains two access counters and a reuse counter to record the total number of byte-granular accesses to the fine-grained read cache 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 high reusability that more data is accessed frequently. We set minimum and maximum ratios to dynamically tune the threshold based on real-time reusability. If the current ratio falls below the minimum, Pipette increases the threshold so that the data is cached less frequently in the low data-reuse situation. On the contrary, a high current ratio larger than the maximum decreases the threshold to allow data promotion frequently. We also specify minimum and maximum thresholds to prevent too much or too fewer data from being cached in the fine-grained read cache (lines 25–29).

Pipette resets the fine-grained reuse counter and access counter (the one maintained to observe reusability fluctuations) when data reuse patterns change dramatically to preserve the most accurate real-time reusability (lines 16–24). This periodic reset strategy periodically records the reuse count and ratio within epochs and compares adjacent epochs' reusability. Upon significant differences, the fine-grained access count, the reuse count and ratio are reset to those of the current phase.

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 the performance of different request sizes and access patterns, an adaptive reassignment strategy assists Pipette in adapting to different workloads. It periodically rebalances slab allocation to match the current workload without hindering application 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 count is unchanged in stages and picks one of its slabs for reassignment. Another rebalance thread is maintained to respond to the reassignment requested by the maintenance thread. It allocates a slab-size spare memory out of the fine-grained read cache and moves the victim slab's data to this region. The recycled slab will finally be returned to the free-slab pool for serving subsequent read requests.

4) Dynamic Allocation Strategy: The dynamic allocation strategy adaptively makes the memory usage tradeoff between two caches, the page cache and the fine-grained read cache,

by comparing their hit ratio. Pipette initializes and maintains respective hit counters (lines 10 and 31) and access counters (lines 1–5) for these two caches during the program execution. The hit ratio can be calculated based on these counters, that is, the ratio of the hit count to the access count. As shown in Algorithm 1 (lines 38–45), the higher the cache hit rate, 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 Section IV-C1). Suppose the fine-grained read cache has a hit ratio greater than or equal to that of the page cache. In that case, the solution of migrating slab (solution 2 in Section IV-C1) is preferred when the shared memory is exhausted.

Algorithm 1 systematically describes the read path with caching mechanisms in the Pipette framework. Buffered read system calls (e.g., `read()` and `pread()` with `O_FINE_GRAINED`) fall into the VFS layer and entry into the page cache. According to the retrieved data size, they are classified into fine-grained and normal reads, and their respective access counters are updated (lines 1–5). Take a fine-grained read as an example. Upon a page cache hit (lines 6–12), the reuse counter is first increased to record tiny data's reusability, and the hit counter for the page cache is raised to calculate its hit ratio. Ultimately, demanded data is read from the hit page (line 11). As for a page cache miss, Pipette determines whether it accesses a file range repeatedly. If so, the reference count of the corresponding range is recorded, and fine-grained data's reuse counter and ratio are calculated (lines 13 and 14). Given the potential for dramatic reusability fluctuations, the adaptive caching mechanism periodically records the reuse count and ratio during epochs and compares neighboring epochs' reusability. Whenever a fluctuation occurs, relative counters and ratios are reset to preserve the most accurate real-time reuse pattern (lines 15–24). Pipette then updates the current reference threshold based on this real-time reusability (lines 25–29). A reuse ratio lower than the minimum increases the threshold to keep data promotion infrequently and vice versa. Next, the fine-grained read cache is checked for the required data. Upon a tiny read hit, the hit count and ratio of the fine-grained read cache is updated and the hit data is returned from the host DRAM directly (lines 30–32). Instead, the adaptive caching mechanism compares the file range's reference count to the current reference threshold. If the former is smaller, missing data is read into the TempBuf Area from the device and then transmitted to the user (lines 33–36). If the latter is smaller, Pipette tries to allocate space inside the Data Area for the request. The dynamic allocation strategy is adopted when the fine-grained read cache is full (lines 38–45). It compares hit ratios of dual caches and reclaims appropriate amount of mappable memory. The device transfers required data to the Data Area, which is then read from the fine-grained read cache to the user space. For the caching operation, we denote the length of link lists led by the corresponding hash heads as n, and the worst- and best-case time complexity for ensuring data consistency is $O(n)$ and $O(1)$.

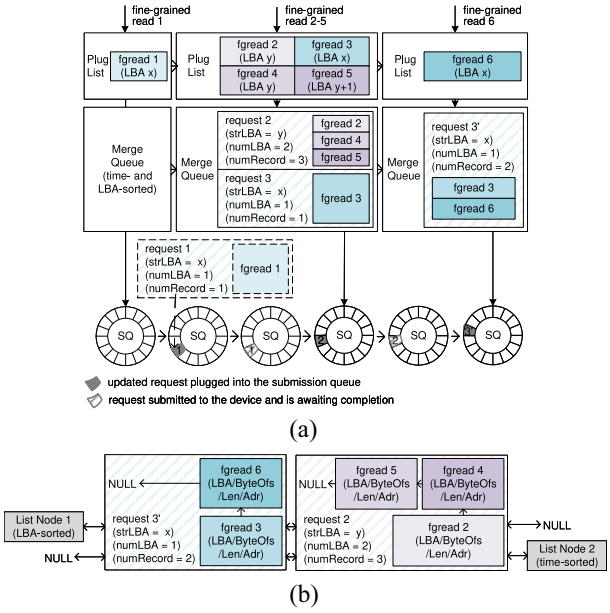


Fig. 5. (a) Example for the best-effort fine-grained I/O merging and scheduling flow. (b) LBA- and time-sorted merge queue structure for this example (supposed that LBA x is smaller than LBA y).

D. Best-Effort Fine-Grained I/O Merging and Scheduling

A hierarchical architecture gives flash arrays a high degree of internal parallelism. A best-effort fine-grained I/O merging and scheduling scheme is performed before the request submission to utilize workload spatial locality and intradevice resources. Tiny reads whose requested data falls into logically continuous pages are combined into one request, facilitating efficient fine-grained reads while taking full advantage of device processing capability for performance optimizations.

Since logical blocks are striped over different SSD chips in multiple distinct flash memory packages, data accesses are performed independently in parallel. Though this is thanks to the FTL's mapping algorithm and has little relevance to whether those LBAs are sequential, parallel writes when writing to a file allow sequential LBA reads for that file to leverage device parallelism to some extent accordingly. Accurate and comprehensive utilization of SSD internal parallelism requires caching FTL's mapping table in the host-side mappable region or device-side IO merging. However, this inevitably adds complexity to the programmability and management of byte-addressable SSDs, and it remains unclear whether system software and applications can benefit more in this case. Thus, we simply merge fine-grained reads that access data in consecutive LBAs to provide best-effort exploitation of device internal parallelism. Instead of submission queues that hold outstanding requests waiting to be processed by the device, we introduce a merge queue to manage I/O merging and rebuild merged requests before fine-grained reads are issued to the submission queues. After merge operations, merged requests are sorted in both LBA and chronological order on the host side [see Fig. 5(b)]. By overlapping the page movement time consumed by the fine-grained reads contained in each merged request, the cost for device internal datapath can be extremely decreased, achieving higher performance. For merged requests

that cannot be transferred from the flash chips concurrently, the device automatically reads out the required pages in batches.

As illustrated in Fig. 5, an arriving fine-grained read first stalls in the plug list to determine whether the submission queues and the merge queue are empty. If they are both empty, this fine-grained read is converted to a read request and is directly issued to the device, bypassing the merge queue. Instead, the earliest queued merged request inside the chronologically ordered merge queue is sent to the device if merely the submission queues are empty, and I/O merging is carried out by dispatching the fine-grained read to the merge queue. Furthermore, as long as one of the submission queues is not empty, i.e., the device is busy processing the previous request, the coming fine-grained read is flushed into the merge queue (whether empty or not) for merging. Pipette traverses the merge queue to conduct the best-effort I/O merging. It either combines the fine-grained read into one of the in-queue merged requests based on the logical block accessed or independently transforms the fine-grained read into a read request and inserts it in the appropriate location in the merge queue. For the latter, to ensure that read requests are submitted to the device strictly in the order of issue, it is also necessary to synchronize the insertion of the newly created read request to the end of the chronologically organized merge queue.

Whenever the host driver is notified that the device processor is idle by receiving a request completion, the earliest read request populated into the chronologically organized merge queue is submitted to the device if the queue is not empty. Additionally, the corresponding read request in the merge queue, arranged by LBA and chronological order, respectively, is removed to prevent subsequent fine-grained reads from IO merging with it. Such a stall-free submission mechanism ensures that application requests are processed without pausing on the device side while keeping IO merging as comprehensive as possible by constantly updating and flushing unsent requests. Through multithreading and pipeline, the host will also perform other read requests or subsequent procedures of the corresponding read request in the process of device-side execution to achieve better performance.

We use an example to demonstrate the best-effort fine-grained I/O merging and scheduling scheme. As shown in Fig. 5(a), fine-grained read 1 is issued when both submission queues and the merge queue are empty. It bypasses the merge queue and is directly submitted to the device as reconstructed request 1, which contains only one small read of its own. During the device execution, fine-grained reads numbered 2 to 5 are coming. Due to the nonempty submission queues, these four tiny reads are plugged into the merge queue to conduct I/O merging. Depending on whether the logical blocks they access are contiguous or not, merged requests 2 and 3 are created and inserted into the merge queue in LBA and chronological order, respectively. When the completion of request 1 is returned, merged request 2 is sent down to the device for subsequent processing. Meanwhile, fine-grained read 6 falls into the merge queue since both the submission queues and the merge queue are not empty. It is combined with request 3, and an updated merged request 3' is generated. In this case, the structure of the merge queue in both LBA and chronological order is illustrated in Fig. 5(b). According to the aforementioned states of the queues, requests and processing situations when each fine-grained read arrives, two merged requests are residing in the merge queue: one is request 2, including three records numbered 2, 4, and 5, that accesses two continuous LBAs starting at LBA y; and another one is request 3', containing two records numbered 3 and 6, that accesses single LBA x. Assume that LBA x is smaller than LBA y, the LBA-sorted merge queue is organized with List Node 1 on the left as the head of the linked list, and request 3' comes first. However, since fine-grained read 2 contained in request 2 is the earliest to be issued, the time-sorted merge queue is arranged with List Node 2 on the right as the head of the linked list, and request 2 is at the front. Note that request 2 will be removed from the LBA- and time-sorted merge queue after it is completed on the device side.

E. Implementation

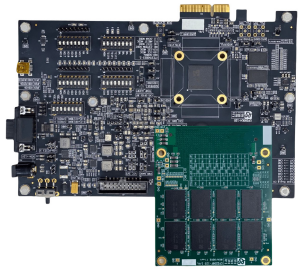
We implement host-side components of Pipette as a kernel module based on Ext4 file system in the Linux kernel v5.4.0. Users can flexibly configure the fine-grained read cache according to specific application characteristics. The relevant `settings` object that consumes 32 bytes maintains passed-in parameters like the cache size, slab size, maximum/minimum item size and growth factor for item sizes. For effective management inside this slab-structure cache, the `slabclass` object records four types of information for each slab class.

1) Basic information includes the contained items' capacity, the number of slabs and items per slab, and slab location.
2) Information of available capacity, i.e., the start offset of the following free item and the number of available items remaining in the last allocated slab.
3) Information of reclamation, such as eviction count, the number of recycled items, and their locations.
4) Information of reassignment like the number of reassigned slabs and offsets before and after reassignment.

Each `slabclass` object consumes 80 bytes to retain the aforementioned information, whose storage overhead is trivial.

A new file open flag, `O_FINE_GRAINED`, is introduced to use the proposed fine-grained read path selectively. The implementation supports fine-grained reads by calling POSIX `read()` system call. We modify the VFS layer and file system layer to support productive caching management. To assist adaptive caching mechanisms, several counters recording access, reuse and hit counts of dual caches, ratios indicating reusability, cache efficiency and hit rates in period, and thresholds are maintained (as shown by the Variables in Algorithm 1), which consumes 68 bytes. Besides, a group of per-file hash lookup tables, whose table sizes are adaptive to their corresponding file sizes, is managed to keep three types of associated information of fine-grained data.

1) Basic information like the file offset, data length, reference count, and location inside fine-grained read cache.



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

Fig. 6. Pipette hardware prototype and specification.

2) Management information for maintaining LRU lists, hash tables, slabs, and waitqueues.
3) Flag bits for status, such as cached or uncached, updated or not updated, and reassigned or not reassigned.

The per-file hash lookup table has entries with 112 bytes each, and 24 of those bytes are used for the LRU list's management.

To support the interactions between the host and device, a lightweight `ebio` object is introduced to bypass the expensive block layer, avoiding unnecessary memory cost of fine-grained access. Each `ebio` object consumes 136 bytes to store its:

1) Essential information like LBA, the start offset inside the LBA, data length, destination address relative to the fine-grained read cache and status.
2) Information of I/O merging, such as the number of LBAs and fine-grained reads of merged requests, and tiny reads' LBA offset in the merged request's consecutive LBAs.
3) Management information for maintaining the sequence of fine-grained reads that make up a merged request and managing merge queue in LBA and chronological order.

Since Pipette no longer uses the `bio`, `bio_vec`, `request`, `iod`, and `prp_list` objects allocated in the block layer of the traditional storage system, the storage overhead for holding the information of fine-grained reads is insignificant. As for statically allocated memory, Pipette replaces the per-core `request` pool, which requires 412-kB memory, with the per-core `ebio` of size 136 kB.

We develop a prototype based on the YS9203 NVMe SSD development platform [19]. The details are listed in Fig. 6. We implement Pipette by modifying firmware. We convert a real SSD to a byte-addressable SSD and emulate a portion of available internal DRAM as the SSD-Cache. The device-side address range corresponding to the HMB 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. Pipette uses the introduced command upon the byte-accessibility of modern SSDs to build a fine-grained read framework while providing direct access to any data in multigranularity.

For multiple threads, we maintain a thread pool in the user program to execute concurrent tasks orderly without overhead for repeatedly creating and destroying threads. Moreover, thread affinity is leveraged to eliminate the impact of thread migration caused by the operating system's scheduling. Through `pread()`, the host is capable of guaranteeing the atomicity of fine-grained reads under concurrent conditions. We also use waitqueues and spinlocks in the Linux kernel to synchronize access to management resources, ensuring correctness. On the device side, we further modify the firmware to support internal parallelism in its vendor read path.

TABLE I
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).

V. EVALUATION

A. Experimental Setup

We use a server machine with an 8-core Intel Core i7-9700K CPU running at 3.60 GHz with 64 GB of DRAM for our experiments. For evaluations, we utilize both synthetic workloads and real-world data-intensive applications to quantify performance benefits of Pipette in different aspects. Under these popular workloads, we report the throughput normalized to the block I/O and the I/O traffic on read operations.

We compare Pipette (donated as *Pipette*) with conventional block I/O (denoted as *Block I/O*) and the state-of-the-art fine-grained approach, 2B-SSD [10], which supports two read modes through MMIO and DMA (denoted as 2B-SSD MMIO and 2B-SSD DMA, respectively, and collectively referred to as 2B-SSD). We also compare Pipette with *Pipette w/o merging*, a solution without scheduling and merging scheme, and *Pipette w/o c&m*, another solution without both the fine-grained read cache and the merging scheme. Note that 2B-SSD and Pipette w/o c&m have no caching supported, we refer to them collectively as *no-host-cache approaches*.

B. Results Under Synthetic Workloads

Five synthetic workloads are constructed by varying the large and small read ratio, as listed in Table I. We perform 2.5 million read requests using synthetic workloads. The 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 spatial reuse pattern. We conduct experiments with one thread. Except for Block I/O and 2B-SSD, we only compare Pipette with Pipette w/o c&m since the merging scheme makes no sense under a single thread.

Under read requests with uniform distribution, Pipette achieves a higher throughput as the ratio of small reads grows, as shown in Fig. 7. For pure fine-grained read workload E, it gains 31.2× performance benefit compared to 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 mechanisms and workloads' temporal locality. The fine-grained read cache

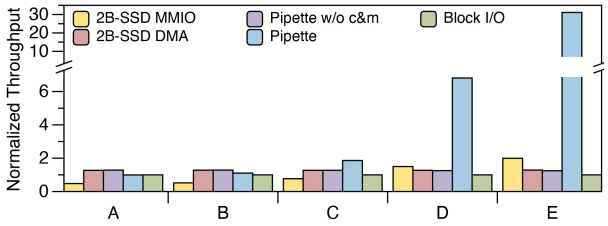


Fig. 7. Normalized throughput of synthetic workloads with uniform distribution.

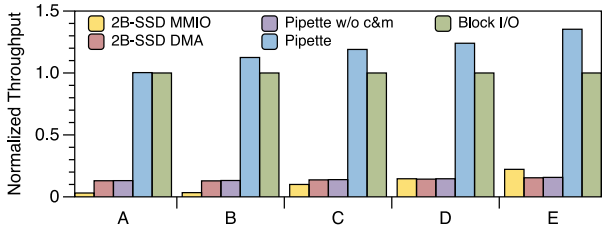


Fig. 8. Normalized throughput of synthetic workloads with Zipfian distribution.

TABLE II
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
No-Host-Cache	9765.6	8819.6	5035.4	1251.2	305.2
Pipette	2973.6	2678.4	1479.7	313.45	79.8

TABLE III
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
No-Host-Cache	9765.6	8819.6	5035.4	1251.2	305.2
Pipette	748.3	684.2	399.9	107.0	33.3

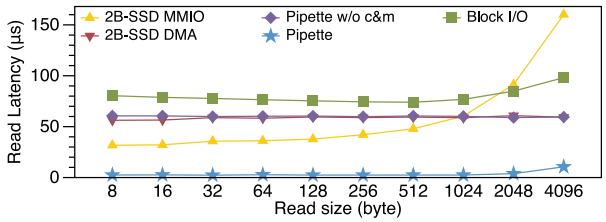


Fig. 9. Read latency of workload E of varying size with uniform distribution.

adaptively holds data with high reuse, preventing DRAM from polluting by low-reusable data. Moreover, it could store tiny data compactly, providing more memory for subsequent highly reused data. On the contrary, page cache promotes every accessed data in page granularity with prefetching, inducing pollution/thrashing problems and frequent page movement in the poor spatial data-reuse situation, as is the case with uniform distribution. No-host-cache approaches slightly improve the performance by avoiding prefetching overhead. Besides, 2B-SSD MMIO suffers a performance degradation when the percentage of large reads increases. This is because MMIO is a nonposted transaction that requires waiting for a completion and is split into small-size (up to 8 bytes on current ×86 CPUs) read transactions to guarantee atomicity [10].

Table II shows the I/O traffic for workloads with uniform distribution. Pipette drastically reduces the I/O traffic by up to 37.3× due to flexible accesses and adaptive caching to consider temporal locality. More block I/Os involved as large reads increases, data transfer increment caused by the read-ahead strategy and page-level migrations gradually raises Pipette's I/O traffic. Since prefetching depends on the location distribution, instead of size distribution, Block I/O has similar I/O traffic in all workloads. No-host-cache approaches only transfer needed data using byte accessibility; their I/O traffic is identical and lower than Block I/O when small reads dominate.

Fig. 8 plots the measured throughput of workloads with zipfian distribution. It shows a similar trend as the previous experimental results but with a slight improvement in throughput when small reads dominated, which is from 1.1× to 1.4×. Because workloads with zipfian distribution preserve certain levels of spatial locality, page cache gains benefits from its advantages. Thus, Pipette has a smaller optimization space. Results of no-host-cache approaches are also similar. In the case of more small reads, 2B-SSD DMA or Pipette w/o c&m gets the worst performance because of their software overhead, i.e., overhead for DMA setting or long I/O stack, respectively. As for 2B-SSD MMIO, it has the best performance when small reads dominate due to its fast speed of reading tiny data but turns to the worst performance in the case of more large reads.

Under read requests with zipfian distribution, Block I/O transfers 22.5× amount of data in the case of pure small read workload E, as shown in Table III. Compared to workloads with uniform distribution, Block I/O has less I/O traffic increment since the data reuse pattern with better spatial locality from zipfian distribution and the read-ahead strategy increase its chance of getting data directly from the page cache.

We now compare average read latencies of varying request sizes from 8 bytes to 4 kB under small-read-intensive workload E with uniform distribution, as reported in Fig. 9. Except for 2B-SSD MMIO, each approach performs stably for different read sizes. Pipette achieves a read latency of 2.31–10.55 μs s thanks to the adaptive caching mechanisms. It shows 9.3× to 33.6× shorter latencies than Block I/O since transparently exploiting both byte accessibility and the fast DRAM access. Due to the page migration and read-ahead strategy, Block I/O suffers 14.56–38.89 μs slower than 2B-SSD DMA, which gains performance from concurrency and fewer in-SSD CPU cycles required to transfer data. As for 2B-SSD DMA and Pipette w/o c&m, their time-consuming software overhead, including per-access DMA mapping and long I/O stack, introduce significant latency, making them comparable latency and 11.51–24.53 μs slower than 2B-SSD MMIO. Besides, the latency of 2B-SSD MMIO increases in proportion to request sizes. It consumes much more time than Pipette w/o c&m and 2B-SSD DMA at a read request size of approximately 1 kB.

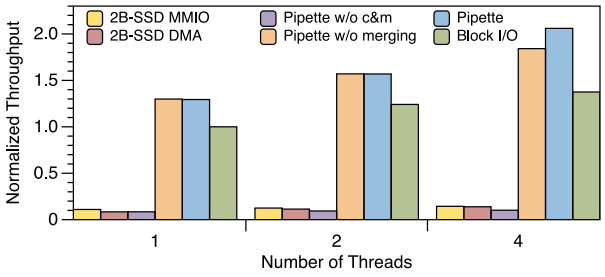


Fig. 10. Normalized throughput of recommendation system.

TABLE IV
RECOMMENDATION SYSTEM: PAGE CACHE VERSUS FINE-GRAINED READ CACHE [1,2]

	Hit Ratio (%)	Memory Usage (MB)	I/O Traffic (MB)
Block I/O	95.16%	2388	7781.06
No-Host-Cache	\	\	4062.50
Pipette w/o merging	95.36%	116	188.39
Pipette	95.36%	116	188.39

[1] The limited host DRAM capacity is set to 2473MB.
[2] Results are measured under one thread.

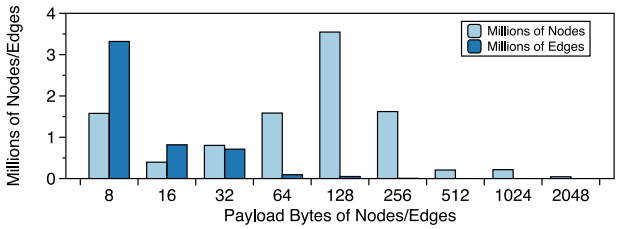


Fig. 11. Payload bytes of social graphs' data in LinkBench.

C. Results Under Real-World Applications

In this section, we evaluate Pipette performance with typical applications: recommendation system and social graph. We limit the host DRAM size for the workloads to simulate a memory-starved environment while ensuring the main memory required by the operating system. For the high learning rate of the adaptive caching mechanisms and their efficiency, we set the counter of `CurrThreshold` to 2, and initialize `MaxThreshold`, `MinThreshold`, and `ResetCnt` to 5, 2, and 10 000, respectively. Users can adjust these parameters flexibly based on workload characteristics in the pursuit of a higher hit ratio, lower memory usage, or higher throughput.

1) Performance Benefit for Recommendation System: We now demonstrate the benefit of Pipette for the recommendation system, a typical high-performance computing (HPC) application broadly utilized in modern online services to provide personalized suggestions based on user interests and preferences. The recommendation models are data- and memory-intensive due to the need to handle categorical input features. In our experiments, multiple threads cooperate to handle these sparse input features by looking up fixed-sized (128 B) embeddings from the 4.1-GB tables inside the SSD [5], [20]. We keep the host DRAM size at 60% of total table size, i.e., 2473 MB, avoiding all data cached upon the block-based interface.

Fig. 10 shows the throughput of the recommendation system with multiple threads, and results are normalized to Block I/O and one thread. Pipette scales the throughput well as the number of threads increases and outperforms Block I/O from 1.26× to 1.50× due to thread-level concurrency, pipeline, and device-level parallelism. Pipette performs faster than Block I/O because many random fine-grained reads are retrieved directly from the fast host DRAM, thanks to the adaptive hot-data promotion. Cache-missed requests are issued to the SSD through the byte-addressable interface, preventing inefficient data movements for low-reusable pages. Moreover, the best-effort merging scheme combines several reads into one batch request that accesses logically neighboring flash pages, providing a high parallelism to alleviate device-side transferring overhead caused by the internal datapath.

Fewer data is transferred between the SSD and the host DRAM upon byte-accessibility. Pipette reduces I/O traffic by 41.3× and 21.6× compared to Block I/O and no-host-cache approaches, respectively, as shown in Table IV. Pipette's adaptive caching mechanisms and refined cache organization make its drastically fewer data movements. Multigranular data is stored compactly to provide more available DRAM for the frequently accessed portion of the application's working set. Besides, the dedicated access scheme for fine-grained reads avoids redundant page migrations and read-ahead strategy, further offering a high hit ratio of about 95.36% and much less memory usage of about 116 MB.

2) Performance Benefit for Social Graph: Beyond the HPC workload, we also evaluate the performance benefit of Pipette for large-scale social networks. Many data powering social networks are represented as social graphs, which includes nodes and edges and are organized into databases. In our experiments, we vary the number of threads from 1 to 4 and use LinkBench [7], an open-source benchmark closely based on production databases that store social graphs for Facebook, to generate a 2-GB graph. The requests are made to access the graph's nodes and edges with a memory limit of 1258 MB (60% of the graph's size). The mean payload per node is 143.3 bytes, while the average payload per edge is much smaller at 15.3 bytes. Fig. 11 shows the overall distributions of nodes' and edges' data sizes. The experiments with social graphs aim to illustrate that Pipette can also flexibly adapt to memory-intensive applications that retrieve objects of various sizes.

Pipette outperforms Block I/O by 1.30-1.54× in throughput, as shown in Fig. 12 (results are normalized to Block I/O and one thread). As we increase the number of threads, the benefit brought by Pipette over Block I/O is increased because the best-effort merging and scheduling scheme can utilize device internal parallelism. The elegant concurrency of device-side whole-page movements and host-side metadata processing operations allows single transferring overhead for SSD's internal datapath to serve multiple fine-grained reads, providing high performance in multithreaded scenarios. As for one thread, Pipette gains benefits because of efficient caching structure and mechanisms, which is also shown from the low throughput of the no-host-cache approaches. In the case of Pipette, only demanded data are moved between the SSD and the host DRAM to store flexibly via the byte-accessible

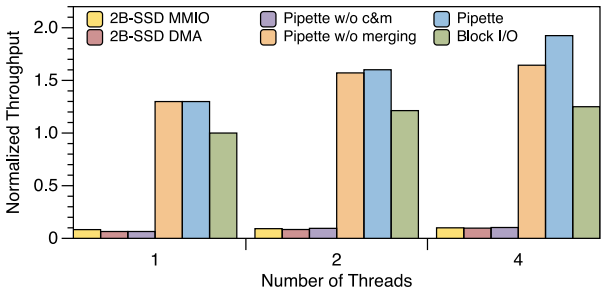


Fig. 12. Normalized throughput of social graph.

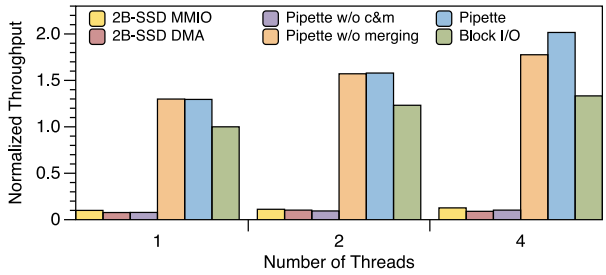


Fig. 13. Normalized throughput of the mixed real-world workload.

TABLE V
SOCIAL GRAPH: PAGE CACHE VERSUS FINE-GRAINED READ CACHE [1,2]

	Hit Ratio (%)	Memory Usage (MB)	I/O Traffic (MB)
Block I/O	95.73%	1191	3255.56
No-Host-Cache	\	\	1439.20
Pipette w/o merging	95.45%	70	57.59
Pipette	95.45%	70	57.59

[1] The limited host DRAM capacity is set to 1258MB.
[2] Results are measured under one thread.

TABLE VI
MIXED REAL-WORLD WORKLOAD: PAGE CACHE VERSUS FINE-GRAINED READ CACHE [1,2]

	Hit Ratio (%)	Memory Usage (MB)	I/O Traffic (MB)
Block I/O	95.34%	3584	11036.62
No-Host-Cache	\	\	5501.70
Pipette w/o merging	95.39%	186	245.98
Pipette	95.39%	186	245.98

[1] The limited host DRAM capacity is set to 3748MB.
[2] Results are measured under one thread.

interface. Furthermore, by accessing tiny data without the paging mechanism and read-ahead strategy, the pollution and thrashing problems of the limited DRAM are relieved, providing high DRAM efficiency.

Since fixed-granular management and accesses, Block I/O transfers a tremendous amount of data and stores redundant parts of pages in the page cache. As shown in Table V, Pipette reduces data movement by 56.5× lower than Block I/O and alleviates memory usage by orders of magnitude. No-host-cache solutions transfer data in byte granularity, avoiding extra I/O traffic caused by unnecessary data, thus having comparable I/O traffic slightly fewer than Block I/O. However, Pipette has significantly fewer data movements than others because its caching mechanisms provide a high hit ratio, 95.45%, enabling more data to be accessed from the cache. Further, the compact organization of the fine-grained read cache for the necessary data greatly reduces Pipette memory usage to about 70 MB.

3) Performance Benefit for the Mixed Real-World Workload: We now evaluate Pipette's performance with a mixed real workload to prove that Pipette scales well to datasets of varying sizes and heterogeneity. We concatenate the workloads of both the recommendation system and social graph to simulate their continuous execution with a host DRAM limitation about 60% of the dataset size, that is, 3748 MB.

Fig. 13 shows the throughput of the hybrid workloads with multiple threads. Normalized to Block I/O and one thread, Pipette performs better than Block I/O by 1.28× to 1.51×, which is comparable to the performance gain for the case of a single application. As shown in Table VI, Pipette transfers significantly less data with a considerable hit ratio and much less memory usage, regardless of whether the workload is heterogeneous. For large working sets, performance can be further improved by enlarging the mappable memory region.

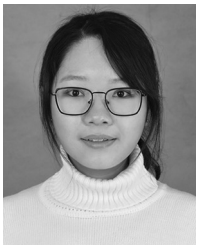
VI. CONCLUSION

This article proposes Pipette, a fine-grained read framework including adaptive host-side multigranular read cache and best-effort merging and scheduling strategies, as an extension to the traditional I/O 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 throughput by up to 50% and 54% for real-world recommendation system and social graph under multiple threads.

REFERENCES

[1] U. Gupta et al., "DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference," in *Proc. 47th ACM/IEEE Annu. Int. Symp. Comput. Archit.*, 2020, pp. 982–995.
[2] U. Gupta et al., "The architectural implications of Facebook's DNN-based personalized recommendation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 488–501.
[3] N. Bronson et al., "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 1–12.
[4] J. He, K. Wu, S. Kannan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Read as needed: Building WiSER, a flash-optimized search engine," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 59–74.
[5] H. Wan, X. Sun, Y. Cui, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "FlashEmbedding: Storing embedding tables in SSD for large-scale recommender systems," in *Proc. 12th ACM SIGOPS Asia–Pacific Workshop Syst.*, New York, NY, USA, 2021, pp. 9–16.
[6] A. Eisenman et al., "Bandana: Using non-volatile memory for storing deep learning models," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 40–52.
[7] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the Facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2013, pp. 1185–1196.
[8] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "Improving SSD lifetime with byte-addressable metadata," in *Proc. Int. Symp. Memory Syst.*, 2017, pp. 374–384.
[9] A. Abulila et al., "FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 971–985.

[10] D.-H. Bae et al., "2B-SSD: The case for dual, byte- and block-addressable solid-state drives," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 425–438.
[11] Z. Yang, Y. Lu, E. Xu, and J. Shu, "CoinPurse: A device-assisted file system with dual interfaces," in *Proc. 57th ACM/EDAC/IEEE Design Autom. Conf.*, 2020, pp. 1–6.
[12] S. Bai et al., "Pipette: Efficient fine-grained reads for SSDs," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, 2022, pp. 385–390.
[13] B. Jacob, "The 2 PetaFLOP, 3 Petabyte, 9 TB/s, 90 kW cabinet: A system architecture for exascale and big data," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 125–128, Jul.–Dec. 2015.
[14] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped SSDS with FlashMap," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 580–591.
[15] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: Introducing multi-queue SSD access on multi-core systems," in *Proc. 6th Int. Syst. Storage Conf.*, 2013, pp. 1–10.
[16] "PCIe 6.0 Specifications." 2022. [Online]. Available: https://pcisig.com/specifications
[17] "NVMe 2.0 Specifications." 2022. [Online]. Available: https://nvmexpress.org/specifications/
[18] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall, 2004.
[19] "YEESTOR: YS9203 PCIE SSD memory controller." YEESTOR Microelectronics. 2021. [Online]. Available: http://www.yeestor.com/
[20] "Kaggle display advertising challenge dataset." Criteo. 2014. [Online]. Available: https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/



Shuhan Bai (Member, IEEE) received the B.E. degree in communication engineering from the Wuhan University of Technology, Wuhan, China, in 2018. She is currently pursuing the Ph.D. degree in computer science with the Huazhong University of Science and Technology, Wuhan, and the City University of Hong Kong, Hong Kong.

Her research interests include computer architecture, high-performance and high-reliability storage systems, near-data processing, and smart storage.

Hu Wan (Member, IEEE) received the B.E. and M.S. degrees in computer science from Capital Normal University, Beijing, China, in 2013 and 2017, respectively, and the Ph.D. degree in computer science from the City University of Hong Kong, Hong Kong, in 2023.

His research interests include memory and storage systems, emerging nonvolatile memory and storage technologies, in-storage computing, and systems for machine learning.

Yun Huang (Member, IEEE) received the B.E. degree in information engineering from the South China University of Technology, Guangzhou, China, in 2020. She is currently pursuing the Ph.D. degree in computer science with the City University of Hong Kong, Hong Kong.

Her research interests include near-data processing and smart storage.

Xuan Sun (Member, IEEE) received the B.S. degree in electronic information science and technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2014, the M.S. degree in computer control and automation from Nanyang Technological University, Singapore, in 2015, and the Ph.D. degree in computer science from the City University of Hong Kong, Hong Kong, in 2021.

She is currently a Postdoctoral Fellow with Imperial College London, London, U.K. Her research interests include hardware acceleration and in-storage computing framework.

Fei Wu (Member, IEEE) received the B.E. and M.E. degrees in electrical automation, control theory and control engineering from Wuhan Industrial University, Wuhan, China, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, in 2005.

She is currently a Professor with the Wuhan National Laboratory for Optoelectronics, HUST. Her research interests include computer architecture, and high-performance and high-reliability storage systems.

Changsheng Xie (Member, IEEE) received the B.E. and M.E. degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1982 and 1988, respectively.

He is currently a Professor with the Wuhan National Laboratory for Optoelectronics, HUST. He is also the Director of Data Storage Systems Laboratory, HUST, and the Key Laboratory of Ministry of Education of China. His research interests include computer architecture, emerging memory and storage, big data storage, and computing systems.

Hung-Chih Hsieh received the B.E. degree in computer science and information engineering from Da-Heh University, Changhua, Taiwan, in 2002.

He is currently working with the Department of Research and Development, YEESTOR Microelectronics Company Ltd., Shenzhen, China. He is the Architect of NVMe storage solutions and is involved in other NVMe-related topics, such as computing storage devices, NVMe-oF, ZNS, and Open-Channel SSDs. .

Tei-Wei Kuo (Fellow, IEEE) received the B.S.E. degree from National Taiwan University (NTU), Taipei, Taiwan, in 1986, and the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 1994.

He is a Distinguished Professor with the Department of Computer Science and Information Engineering, NTU. His research interests include embedded systems, nonvolatile-memory software designs, neuromorphic computing, and real-time systems.

Dr. Kuo is the Vice Chair of ACM SIGAPP and the Chair of ACM SIGBED Award Committee.

Chun Jason Xue (Senior Member, IEEE) received the B.S. degree in computer science and engineering from The University of Texas at Arlington, Arlington, TX, USA, in 1997, and the M.S. and Ph.D. degrees in computer science from The University of Texas at Dallas, Richardson, TX, USA, in 2002 and 2007, respectively.

He is currently a Professor with the Department of Computer Science, The City University of Hong Kong, Hong Kong. His research interests include memory and storage systems.