2018

Bridging the Gap between Application and Solid-State-Drives

Jian Zhou
University of Central Florida

BRIDGING THE GAP BETWEEN APPLICATION AND SOLID-STATE-DRIVES

by

JIAN ZHOU
B.S. Wuhan University of Science and Technology, 2008
Ph.D. Huazhong University of Science and Technology, 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2018

Major Professor: Jun Wang

ABSTRACT

Data storage is one of the important and often critical parts of the computing system in terms of performance, cost, reliability, and energy. Numerous new memory technologies, such as NAND flash, phase change memory (PCM), magnetic RAM (STT-RAM) and Memristor, have emerged recently. Many of them have already entered the production system. Traditional storage optimization and caching algorithms are far from optimal because storage I/Os do not show simple locality. To provide optimal storage we need accurate predictions of I/O behavior. However, the workloads are increasingly dynamic and diverse, making the long and short time I/O prediction challenge. Because of the evolution of the storage technologies and the increasing diversity of workloads, the storage software is becoming more and more complex. For example, Flash Translation Layer (FTL) is added for NAND-flash based Solid State Disks (NAND-SSDs). However, it introduces overhead such as address translation delay and garbage collection costs. There are many recent studies aim to address the overhead. Unfortunately, there is no one-size-fits-all solution due to the variety of workloads. Despite rapidly evolving in storage technologies, the increasing heterogeneity and diversity in machines and workloads coupled with the continued data explosion exacerbate the gap between computing and storage speeds.

In this dissertation, we improve the data storage performance from both top-down and bottom-up approach. **[top-down]** First, we will investigate exposing the storage level parallelism so that applications can avoid I/O contentions and workloads skew when scheduling the jobs. Second, we will study how architecture aware task scheduling can improve the performance of the application when PCM based NVRAM are equipped. **[bottom-up]** Third, we will develop an I/O correlation aware flash translation layer for NAND-flash based Solid State Disks. Fourth, we will build a DRAM-based correlation aware FTL emulator and study the performance in various filesystems.

This dissertation is dedicated to my wife Yaping Wang

and my child Abby Zhou.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my adviser Dr. Jun Wang, for his continued guidance, strong support, and valuable advising throughout my Ph.D. journey. Without his guidance and persistent help this dissertation would not have been possible.

I would like to thank my committee members, Dr. Mingjie Lin, Dr. Deliang, Dr. Fan Rickard Fredrik Ewetz, and Dr. GuoJun Qi, for serving on my dissertation committee and providing valuable guidance and suggestions.

This project is supported in part by the US National Science Foundation Grant CCF-1337244, 1527249 and 1717388, and US Army/DURIP program W911NF-17-1-0208.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: APPROXSSD: DATA LAYOUT AWARE SAMPLING ON AN ARRAY OF SSDS

Execution of analytic frameworks on sample data sets is the current trend in response to increasing data size and demand for real-time analysis. Additionally, high-performance, energy-efficient Solid-State Drives (SSD) arrays are the primary storage subsystem for parallel data analysis systems. To exploit the benefits of SSD arrays when executing sample data set analytics, several key areas must be considered. First, due to logical to physical address translation, random data choice in data sampling jobs can cause unbalanced workloads among SSDs in the array. Second, after the data choice, existing task schedulers in data analysis frameworks can introduce non-negligible resource contentions resulting from the suboptimal Input/Output (I/O). The performance of SSDs is unpredictable because of their varying maintenance costs at runtime, which renders them hard to be managed by the scheduler. With the trend towards sample set data analytics and the use of SSDs, it is increasingly important to ensure balanced workloads and minimize resource contentions. Without addressing these areas, sample-set data analytics on SSDs will continue to suffer from performance inefficiencies.

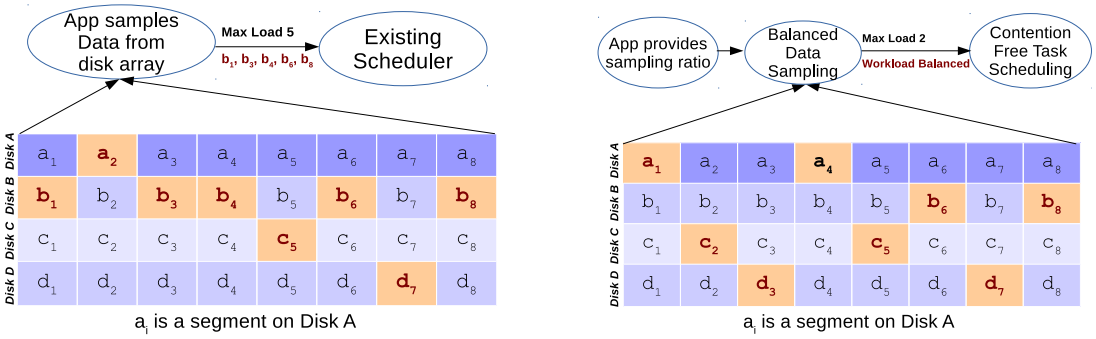
In this paper, we propose ApproxSSD to perform on-disk layout-aware data sampling on SSD arrays. This proposed framework leverages data selection and task scheduling to improve the performance of many applications. ApproxSSD decouples I/O from the computation in task execution. This avoids potential I/O contentions and suboptimal workload balances. We have developed an open-source prototype system of ApproxSSD in Scala at Github. Our evaluation shows that ApproxSSD can achieve up to 2.7 times speed up at 10% sampling ratio under WordCount workloads when compared to Spark, while simultaneously maintaining high output accuracy.

Introduction

As the number of CPU cores continues to increase [1, 2], applications have started to embrace many-core by spawning multiple concurrent jobs. Meanwhile, even when ideal task-level parallelism is achieved, storage systems become the major stumbling block to the scalability of many I/O-intensive applications [3]. Many recent works [4, 5, 6] propose to perform big data analytics and machine learning algorithms on powerful servers equipped with arrays of NAND Flash-based Solid State Drives (SSDs) [7]. SSDs in these scaled-up servers play a key role in reducing the I/O overhead. This is because today's big data programs, especially iterative algorithms can easily exhaust server's memory space. The memory capacity demands are typically handled by using disk space, i.e., swap regions. This results in frequent data partitions migration between disk and memory. Popular in-memory systems, such as Spark [8], are hindered by this approach. Also, data sampling techniques are widely used to shrink the data size and enable in-memory processing. For most sampling applications, uniform data selection [9, 10, 11] is used. In some other cases, such as sparse input datasets [12, 13, 14, 15], conditional sampling is required for higher accuracy. Most research focuses on enhancing the output accuracy or the estimation of the error bar. However, there is another important research area that has not be thoroughly covered, which is motivating the focus of this chapter. It is the sub-optimal use of SSD arrays due to inefficient data sampling techniques that are not aware of the low-level data layout.

The first problem in approximate data analytics on SSD arrays is the workload balance issue. In uniform sampling, the assumption is that the amount of data sampled from each disk is equal [9, 10, 11]. Counter-intuitively, this ideal case rarely happens [16, 17]. Suppose we randomly sample eight data partitions from eight disks, the theoretical probability of having each disk serves one data partition is as low as one in twenty-two. Such pattern persists even if the number of sampled data partitions increases. Figure 1.1a shows an example that disk b contains the most selected

dataset. A key design challenge to optimize task execution time is to balance the data sampling across all SSDs to avoid throttling delays from an SSD with a large amount of input data. To avoid this type of bottleneck, as illustrated in Figure 1.1b, we expose the data layout information to the data sampling engine and then perform a balanced sampling strategy on among the SSDs.

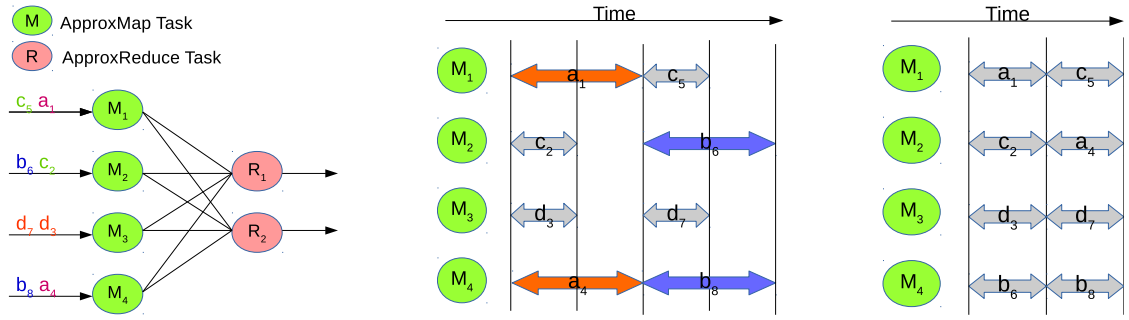


(a) Application selects a subset randomly out of the entire data set. The sampled data are concentrated on few disks.

(b) Given multiple choices of a subset, balanced data sampling selects the one which accesses the same amount of data from each SSD to enforce load balance.

Figure 1.1: Workload balance issue in performing data sampling on SSD arrays. The total amount of sampled data remains the same between current random selection and our proposed scheme. Orange indicates the sampled data.

Secondly, the random access of input data and the blind task scheduler results in severe I/O contention between SSDs. As reported, the performance of random read accesses on SSDs is worse than that of other access patterns and operations, including random write accesses [18]. While sequential accesses can be easily striped over multiple storage media in a round-robin fashion to exploit their parallelism, random read requests cause severe resource conflicts [19]. Figure 1.2a shows the input task orders when we submit the sampled dataset to mapping threads in a round robin way. The ApproxMap and ApproxReduce tasks are variantions of the MapReduce tasks with the added ability to perform data sampling and task dropping operations. In this illustration I/O contention is occurring first on disk a and then on disk b if the map tasks are executed in the default order. This is occurring even though the sampled data set is balanced among SSDs (Figure 1.2b).



(a) The sampled segments are submitted to ApproxMap tasks in a round-robin way so that each task processes the same number of segments.

(b) Contentions happen while multiple tasks request segments on the same disk (contentions happen first on disk a then disk b).

(c) Reordering segments read based on the data layout to remove contention.

Figure 1.2: I/O contention issue on SSD array. The data partitions are sampled as Figure 1.1. The ApproxMap and ApproxReduce tasks are variantions of MapReduce tasks with the added ability to performing sampling and task dropping operations.

Our idea is to develop an on-disk data layout aware scheduler to avoid the I/O contentions so that we can get results as in Figure 1.2c. The way to tackle I/O contention in storage layer is to reorder I/O requests [20, 19], or use decoupled I/O threads to improve the prformance [21]. However, the performance of SSDs at runtime varies significantly and is impossible to forecast by the task scheduler, also the data to disk mapping is hidden by the storage layer. As a result, the proper task execution order cannot be determined. Additionally, reordering tasks will not produce an acceptable solution if tasks are skewed to a few disks. To solve this, we propose to exclusively use I/O threads to load data from each disk, where two threads will not compete for one disk, and one slow disk will not affect others.

Emerging storage systems and devices, such as Ceph[22], copy-on-write filesystems [23] and Open-channel SSDs[24], try to avoid the usage of a fixed RAID controller, but instead, expose the storage-level parallelism to the application layer. However, existing big data frameworks do not take advantage of storage level parallelism in the task planning and scheduling. To improve the

I/O efficiency, we developed ApproxSSD: a novel parallel data processing framework with an integrated *active storage driver*. Upon the job submission, ApproxSSD constructs a dependency tree of computing and I/O tasks. The corresponding I/O tasks are then dispatched to the storage driver for processing. The data sampling job is launched and computed by the storage driver where data layout information is available. Therefore, data partitions can be sampled in a balanced manner while meeting the sampling ratio requirement. The I/O scheduler then pro-actively determines the data loading sequence to fully exploit the bandwidth of each disk. Finally, we process fetched data in parallel. Benefiting from multiple decoupled components, our design is expected to unleash the full potential of SSD arrays and the power of multi-core CPUs. In doing this, we have made the following contributions:

- We proposed an active data storage engine which leverages the flexibility of partition processing order to fully unleash the parallelism in SSD arrays.
- We proposed a data-layout aware sampling scheme to balance the workloads among multiple SSDs by using non-uniform segment sampling.
- We have prototyped ApproxSSD in Scala, which is open sourced at https://github.com/janzhou/approxssd. The results indicate up to 2.7 times speedups compared to the state-of-the-art data analysis engine.

Motivation and Analysis

Workloads Balance in Data Sampling

The sampling-based approximate query has been widely used in modern data-intensive analytic applications. Sampling obtains a much smaller data set with the similar data structure and features.

As a sampled dataset can typically fit into the main memory, the technique reduces the latency and resource usage. The basic technique that is used to perform data sampling is choosing the data randomly from the entire dataset. Unfortunately, random data access results in imbalanced workloads across SSDs in an array. The overall execution time of data sampling is determined by the SSD with the maximum load.

Figure 2.4 shows the workload imbalance issue in 4 real world applications. The details of the applications are described in the evaluation section. The measurements were obtained using the standard Linux iostat command to capture the amount of data read from the SSD during application execution. The balance percentage is calculated by dividing the calculated average amount of data served by the array of SSDs by the maximum amount of data served by a single SSD. For each workload, we get the average value and deviation from 20 executions. In four real-world applications, the work load balance is on average 60%, with deviations as high as 20%, as shown in Figure 3. This is due to the fact that data sampling is performed by the approximate applications or computing frameworks [10, 25], while the physical data layout is masked by the storage software such as filesystems and RAID controller. Motivated by this, we propose to expose the internal parallelism inside the storage architecture to the data sampling framework. The data layout aware sampling can effectively balance the maximum workload of each SSD in a storage array.

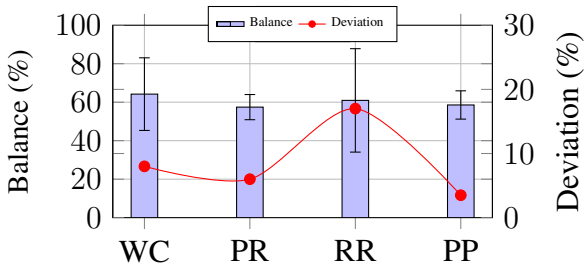
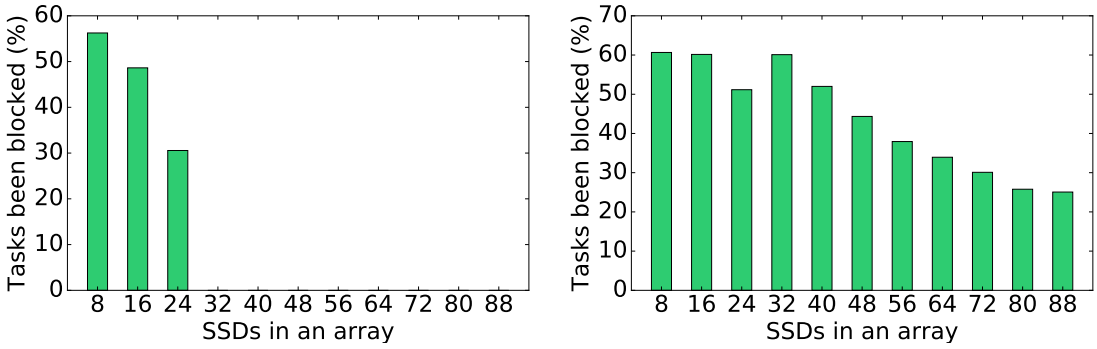


Figure 1.3: Workload imbalance issue. The higher values are optimal, with 100% indicating the load is fully balanced. Using 1% sampling ratio. (WC: Word Count; PR: Page Rank; RR: Request Rate; PP: Project Popularity)

Contention Analysis

To get a better understanding about the I/O contention, we mimic data sampling job performance with an ideal "what-if" simulator. Our simulator is unconstrained and (i) randomly selects data from all the SSDs in the input phase and (ii) submit data analysis tasks such that the workload is distributed evenly across all the threads. We quantitatively measure the contention by counting the number of tasks that have been blocked due to I/O contention. If two tasks contend for one physical disk, we count one of the tasks as blocked. We ignore the internal parallelism of SSDs, such as the channel-level parallelism, in the computational analysis. However, this simplification does not affect the conclusion that random data sampling can potentially increase the I/O contention. Figure 1.4 shows the percentage of tasks that have been blocked compared to the total number of disks in the array. The sampling ratio is the percentage of data been selected from the origional data. We can find that the maximum percentage of tasks been blocked due to I/O contention is around 60%. It also shows that as the number of disks increases the chances of having a contention decreases. However, the percentage of contention I/O tends to increase as we decrease the sampling ratio. This is because when we use a higher sampling ratio, more data will be selected, thus making it easier to parallelize the I/O requests. For example, when processing the whole data (100% sampling ratio), we can perform sequential I/Os which can be easily striped to disks. This is why the result shows no contention when using 100% sampling ratio, 32 I/O threads and 32 disks. When we reduce the sampling ratio, it will result in more random I/O and thus more contentions.



(a) Sampling Ratio 100%.

(b) Sampling Ratio 10%.

Figure 1.4: Computational analysis on the percentage of blocked tasks due to I/O contentions under different sampling ratios. 32 I/O threads are used.

Approxssd Design and Optimizations

Architecture Design

Data analysis frameworks and their corresponding file systems [8, 26] emphasize on high throughput and scalable performance by splitting data into large chunks on servers. However, these servers may includes an array of disks and the task scheduler is unaware of the data distribution among them and assumes I/Os can be easily stripped to disks. Unfortunately, the on-disk data distribution is becoming important when we introduce sampling to reduce the input data size and provide fast approximate results. To guarantee an acceptable approximate error, data sampling is done using random and small-sized data access on disks. Because of this, contemporary analysis frameworks cannot unleash the full potential of SSD array performance. Without the data layout information, applications will sample uneven amounts of data from disks. Currently, the task schedulers have no coordination with the I/O systems to avoid disk contentions. Even though that the analytical frameworks and the underlining storage system work in a parallel manner, the random data access, and the blind task scheduler will cause severe load imbalance and contention issues between SSDs.

We propose ApproxSSD, a decoupled data storage and processing engine designed for big data sampling. As shown in Figure 1.5, **ApproxSSD** mainly consists of an active storage driver and a task scheduler. The active storage driver is a software module that runs on the host system to govern the storage system. We expose the detailed information of the storage system, such as the number of SSDs and the placement of data partitions, to the storage driver. To achieve this, we manage the partition to disk mappings in the active storage driver directly, instead of using a fixed RAID controller or a black-boxed file system. The user then provides the required sampling ratio to the active storage driver. The storage driver then pro-actively loads the data according to the sampling ratio and data layout to maximize the bandwidth of each disk. Finally, the task scheduler launches computing tasks based on the dependencies.

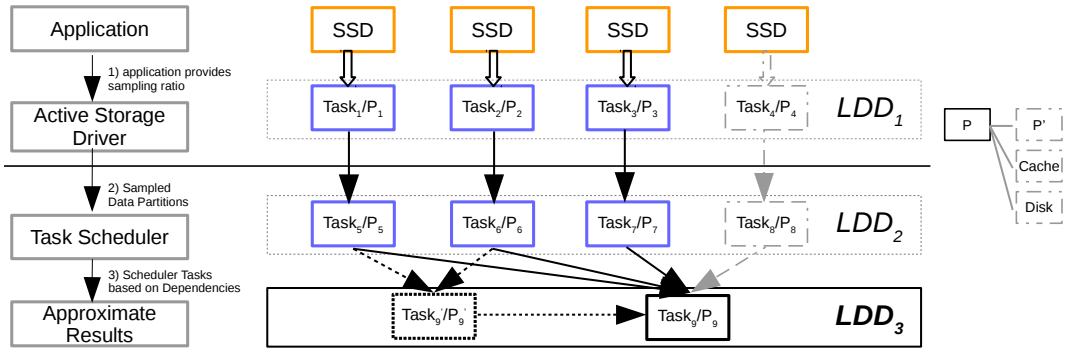


Figure 1.5: ApproxSSD Architecture: Partition and Task Dependency Tree (P_i is a partition).

Data Structure and Programming Interface

ApproxSSD provides a Log-structured Distributed Dataset (LDD) interface to manipulate the dataset. An LDD is constructed by several data partitions. A data partition is the minimum disk I/O unit and is stored in a continuous space on one disk. As shown in Listing 1, LDD supports standard MapReduce data manipulation functions such as map, reduce and join. In addition, we also provide *sample* operation to perform layout aware data sampling.

```
abstract class LDD[T : ClassTag] extends Serializable {
  def partitions : Array[Partition[T]]
  def map[V : ClassTag](v : T => V) : MapPartitionsLDD[V,T]
  def sample(ratio : Double) : MapPartitionsLDD[T,T]
  def samplePartitions(ratio : Double) : LDD[T]
  def filter(v : (T => Boolean)) : MapPartitionsLDD[T,T]
}

abstract class MapLDD[T : ClassTag, U : ClassTag] extends LDD[(T, U)] {
  def reduceByKeyPartitions( f : (U, U) => U) : MapLDD[T,U]
  def reduceByKey(f : (U, U) => U) : ReduceByKeyLDD[T, U]
}
```

Listing 1: Selected Interface Design for ApproxSSD

Characteristics of Partition data structure: The data of an LDD partition can be loaded or computed from multiple sources: the disk, the in-memory cache or the dependent partition. And the data source of a partition is marked by a flag variable. When data is stored on disks, the *disk identity (ID)* and *data offset* are stored within the partition data structure, so there is no additional query when fetching data from disks. The benefit gained by using this type of partitioned data structure is that we can hide the locations of data from the applications and do not need to implicitly load the data before performing data manipulation operations. The procedure of loading the data partition from disk to in-memory cache is defined as *I/O task*. The procedure of executing a program with data from a partition is defined as *computing task*.

Data flow: A data manipulation function transforms one LDD to another. The partition dependent tree is constructed while doing the LDD transformation (e.g. map, reduce and join operations). Upon a job submission, we submit all the partitions whose data are stored on disk in the active storage driver as well as a data sampling ratio. The data loading is automatically done by the active storage driver, and the data source then points to an in-memory cache.

Persistent LDD and Transitional LDD: Intermediate data is defined as transitional LDD and it can be dropped after use. On the other hand, a persistent LDD is the resultant data which needs to

be stored in the disks when the computation is done. When writing persistent LDD data to disks, partitions provide data, but the disk IDs and the offsets of the written data are determined by the storage driver.

Data Sampling: For the data sampling, LDD provides two levels of sampling functions. One is inter-partition sampling. The other is intra-partition sampling. In contrast to the existing approximation techniques in parallel data analysis engines which rely on applications to sample data, ApproxSSD provides high level abstract for the sampling functions so that balanced data selection and contention free task scheduling are enabled.

Task Scheduling and Execution

In this subsection, we discuss the computing and I/O task scheduling. First, we use an *I/O scheduler* in our active storage driver to schedule I/O tasks and a dedicated *task scheduler* to schedule computing tasks. For example, in Figure 1.5, the tasks for partition P_1 to P_4 are simply loading data from disks. The I/O scheduler can utilize the on-disk data layout information to facilitate the scheduling of data loading tasks. The rest of the tasks are scheduled by the task scheduler, where we mainly focus on the dependencies. Second, when a task is executed, a signal is sent to all tasks depending on it. The task scheduler then launches its dependencies accordingly. If one partition depends on multiple children, all its child partitions need to be executed before it can be launched. If a partition is dropped, e.g. P_4, a drop signal is sent to tasks depending on it and its dependency is removed. Third, to improve CPU efficiency, we overlap the execution of tasks on different levels. For example, if Task T_5 and T_6 are finished and the Task T_7 is still waiting for the I/O Task T_3, we dynamically construct a new task $T_9^{'}$, which depends on T_5 and T_6, and submit to the task scheduler. Overlapping the task execution in different levels can reduce the CPU time spend on waiting the I/Os.

There are three major execution steps for a sampling-based approximation job:

- **Job submission.** While submitting an approximate job, the user specifies the sampling ratio for the input data.
- **Input data sampling.** The active storage driver selects the data partitions based on the sampling ratio and loads the data accordingly.
- **Task execution planning.** As the partitions are loaded, the computing tasks are then initiated by the scheduler according to the task dependencies.

The Optimizations for Approxssd

ApproxSSD can perform several on-disk data layout aware optimizations.

Use Data Layout Aware Sampling to Balance the Workloads

In existing data analysis frameworks, the workload imbalance problem cannot be efficiently avoided because 1) there is no easy way to get the on-disk data layout information in existing systems, e.g. HDFS and 2) as discussed in the above section, the chance of having the sampled data evenly distributed is low. However, in ApproxSSD, the data to disk mapping is directly managed by our prototyped active storage driver.

To balance workloads among SSDs, we first group input partitions based on their disk IDs. Then, one sampling process is initiated within each group and an equal number of partitions are sampled. Although this method is simple and straightforward, it can be inefficient when the number of partitions is large, and the sampling ratio is low. When the sampling ratio is low, we choose partitions one by one until the required amount of partitions is satisfied. If the number of partitions

on a certain disk exceeds others by a threshold, then some partitions on that disk are dropped. Thus, the same amount of data is selected from each disk (distinguished by the color orange in Figure 1.1). As a result, the potential hot spots on disks can be avoided. Furthermore, instead of sampling the same amount of data on each disk, the number of sampled partitions can also be weighted by the performance of disks in a heterogeneous system.

Use Decoupled Task Threads to Avoid the Contention

Loading the same amount of data from each SSD does not equal to unleashing the full potential of SSD array performance. There is a chance that multiple independent tasks request data from one disk simultaneously. In order to avoid the array level I/O contention, we developed two thread pools. One is for the computing tasks and the other is for the I/O tasks. The execution of computing tasks requires the I/O tasks finish first. We bind each SSD with a set of exclusive I/O threads. The I/O scheduler will dispatch I/O tasks to SSD's corresponding threads set. Thus, none of the array level contentions happen. After the data loading tasks have been finished, the computing tasks depending on them are submitted to the execution queue.

Writing: The data of the write requests is stored to disks in a Log-structured manner. Because of this, data on disks is immutable and it is safe to spawn any number of read threads for each disk. However, for write threads, a shard write queue and log head is created for each disk to avoid data racing. Contrary to the traditional storage system, target disk and the offsets of the written data are not indicated by the write operations but are determined by the driver. The write requests are dispatched by the write router. Write router collects the write queue information from each disk, then dispatch writing tasks to the SSD with shortest write queue. When the request is finished, the disk ID and the offset of the written data are returned and maintained in the *partition* data structure, so that we don't need to query the mapping when this partition is retrieved again.

To reduce memory overhead, we give write requests higher priority compared to read requests because we can release the memory space occupied the output data only after it has been written to the SSD array. The write requests in each disk's write queue are scheduled by a first in first out (FIFO) scheduler.

Reading: The I/O scheduler does not schedule read requests based on their arrival order. Instead, it creates multiple independent read queues for each disk. Thus, we use multiple decoupled I/O threads to process the read requests in each disk's queue. Because of this, I/O threads from different disks will not compete with each other. All the on-disk data and the location of the data is immutable. As a result, there is no cost in updating the mapping and it is safe to perform multi-threaded data loading without concerning the data racing problems.

Drop the Straggler Disks

Due to the unpredictable performance of SSDs, runtime straggler cannot be avoided even if the even amount of data is sampled from each disk. In order to prevent a straggling SSD, which will slow down the entire application, we always select 10% more data compared to the required samples size from the underlying data and do not wait for the last 10% tasks to finish. For example, if the sampling ratio is 1%, we select a total of 1.1% data from the input data. With fine-grained control over the sampling ratio on multiple disks, the reduction of straggling SSDs during data loading phase leads to further improvements in approximation runtime.

Error-bar Estimation

We adopt the standard multi-stage sampling theory [27, 25] to compute error bounds for approximate applications. The set of supported aggregation functions includes SUM, COUNT, AVG, and

PROPORTION. In particular, we describe the approximation of SUM as an example. Approximation for other aggregation functions shares the similar process. Depending on the dataset and computation, it may be necessary to use bootstrap methods [28, 27, 9] to generate a lower error bar (i.e., more accurate) by introducing additional re-sampling overhead. Assuming we divide the entire datasets into multiple data partitions, we have a total of T items which are divided into N partitions, and each partition has M_i items so that $T = \sum_{i=1}^{N} M_i$. Each unit in partition i has an associated value v_{ij}, and the sum of these values can be obtained by $\tau = \sum_{i=1}^{N} \sum_{j=1}^{M_i} v_{ij}$. To estimate the sum τ, we sample a list of n partitions which are randomly selected based on their inclusion probability π_i. The estimated sum is expressed in equation 3.1:

$$\hat{\tau} = \frac{N}{n} \sum_{i=1}^{n} (\frac{M_i}{m_i} \sum_{j=1}^{m_i} v_{ij}) \pm \epsilon \tag{1.1}$$

where, $\hat{\tau}$ is the estimated τ and the error bound ϵ is defined equation 3.2:

$$\epsilon = t_{n-1,1-\alpha/2} \sqrt{\widehat{V}(\hat{\tau})} \tag{1.2}$$

where, $\widehat{V}(\hat{\tau})$ is defined in equation 1.3:

$$\widehat{V}(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n} \sum_{i=1} nM_i(M-i-m_i)\frac{s_i^2}{m_i} \tag{1.3}$$

where, (s_u^2) is the variance between partitions, and (s_i^2) is the variance within the partition i.

Table 1.1: List of evaluated applications.

Application	Algorithm	Datasets
Word Frequency	Word count	Wikimedia Dumps
Page Rank	Graph processing	Wikimedia Dumps
Request Rate	Average	Wikimedia Pageviews
Popular Pages	Sum	Wikimedia Pageviews
Popular Project	Sum	Wikimedia Pageviews

Experiments

To evaluate ApproxSSD, we used multiple common real world data-analytics applications and algorithms (see Table 1.1 for details). We executed Word Count (WC) and Page Rank (PR) algorithms on 49GB Wikimedia Data Dumps [29]. Popular page (PP) and request rates (RR) are executed on 219GB page access log for Wikimedia Page Views [30].

Experimental Setup

ApproxSSD is built as a standalone prototype system using Scala programming language. It has a direct access to the SSDs. Each SSD is bound to an independent storage driver. The storage driver has one log-structured write head which serves the write request in a first-come, first-served (FCFS) manner. At its initialization, ApproxSSD has its data stored on an external storage server. As a result, the experimental data needs to be moved to the prototyped storage system before execution. During the movement, the write request is submitted to the write router first, then dispatched to the storage driver with the least outstanding requests. Data stored on the disk is immutable to guarantee thread safety so that there is no limit on the number of concurrent read requests. The executor is configured with 8 map threads and 8 reduce threads in total. To conduct our experiments, we used an 8-core Xeon server with 64GB memory and 8 OCZ 120GB SATA SSDs. We employ the RAID

controller and ext4 file system in our experiments to set up a working Spark baseline system for comparison. In ApproxSSD, we bypass the RAID controller to gain direct access to the physical disks. We use a typical 8KB partition size by default in both the baseline and ApproxSSD settings.

We start with studying the impact of ApproxSSD on the performance and the accuracy of the applications. Later, we study and analyze the impact of ApproxSSD on key metrics, such as I/O throughput and I/O contention. For job execution time, the experiment is repeated 20 times. Then we report the average execution time and its deviation. To keep the results consistent, we clear the system buffer before the beginning of each experiment. For accuracy, we calculate estimated error bars and the difference between precise value and approximate value. To measure the I/O contention and its impact, I/O latency, I/O throughput as well as the percentage of the CPU I/O wait time are reported.

Performance Comparison

For our performance evaluation study, we use two publicly accessible datasets, Wikipedia data dump and Wikipedia access log. Specifically, we study 4 applications in this section. First, in the Word Count analysis for the Wikipedia data dump, a key-value pair $< word, 1 >$ is generated in the map phase, and in the reduce phase the count for each word is summed. Second, in the page rank analysis, we extract the $< p, 1 >$ pair from the Wikipedia data dump, where p represents the page that is linked; in the reduce phase, we count the number of links pointing to page p. Third, in the Wikipedia request rate analysis, we calculate the average number of requests from the Wikipedia access log. Similarly, $< time, 1 >$ pair is generated in the map phase. We then group and sum the $< time, value >$ pair based on per hour time unit. Finally, in the page popularity analysis for Wikipedia access log, $< page, 1 >$ pair is produced by the number of page accesses in the map phase and then the total accesses in the reduce phase is obtained by a simple summation.

In the baseline, we use a random boolean generator to randomly select a number of data partitions to be used for analysis jobs. While in ApproxSSD, the same amount of data is sampled from each disk drive. The total amount of sampled data is 10% more than the requested in order to avoid the bottleneck caused by the straggler disks. Then multiple independent data fetchers are launched for each disk drive, and fetched data is submitted to the task scheduler. When the amount of fetched data reaches the requested volume, the fetch process is stopped for the final estimation of results.

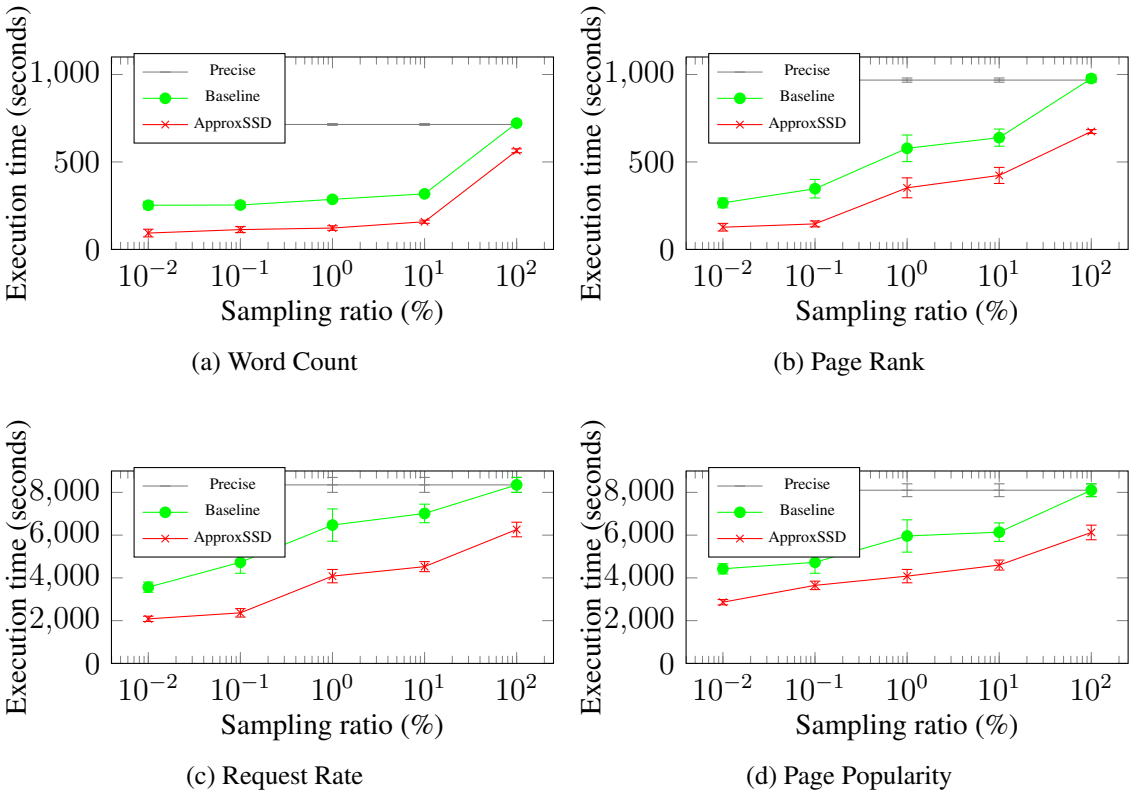


Figure 1.6: Execution time comparison under different sampling ratios.

Figure 1.6 shows the total execution time under different approximation strategies. X-axis is the percentage of the data being sampled and is in log scale. Y-axis is the total execution time in seconds. Each data point in the figure represents the average value over 20 executions as well as the minimum and maximum values. The gray line represents the time spent on precise execu-

tion in baseline system. The green line represents the approximation performance of the baseline system. While the red line represents the approximation performance of ApproxSSD with all the optimization strategies enabled.

Figure 1.6a and Figure 1.6b show the total execution time of Word Count (WC) and Page Rank (PR) respectively when different sampling ratios are applied to the Wikipedia data dump dataset. The execution time of both algorithms decreases with sampling ratio. ApproxSSD is 2.7 and 2.09 times faster than the baseline Spark solution when the sampling ratio is 0.01% (Divide the execution time of the baseline by that of the ApproxSSD). As we reduce the sampling ratio from 100% to 0.01%, the total execution time of PR decreases more significantly than that of WC after 10% sampling ratio. This is because when the input data size is reduced, PR algorithm can achieve better data locality and thus lower memory usage overhead. The extra performance gain when doing approximate query is due to the balanced data choices among the SSDs and the runtime straggler dropping. Even when the sampling ratio is 100% which means the entire dataset is processed without sampling, ApproxSSD is 1.28 and 1.45 times faster than the baseline solution. This is because the reduced I/O stack and parallel data read threads can shorten the data load time.

Similarly, the total Request Rate and Page Popularity analysis time for Wikipedia access log dataset with different sampling ratios are shown in Figure 1.6c and Figure 1.6d respectively. ApproxSSD is 1.70 and 1.55 times faster than the baseline when the sampling ratio is set to 0.01%. ApproxSSD can also outperform the baseline by 1.33 and 1.55 in terms of execution time when analyzing the entire dataset.

Error-bar Estimation

In this section, we study the estimation error bar in ApproxSSD. ApproxSSD balances the workloads by performing segmented sampling in each disk. Some researchers, such as BlinkDB [31]

split the data based on the data dimensions and then perform segmented sampling in each dimension to improve the approximate accuracy. There may exists bias in the sampled data when the content of the file is not equally likely to be selected. However, both the random sampling and the data-layout aware sampling developed in current ApproxSSD does not consider the data dimensions or how the content is distributed in the files. As any partition still has equal chance of being selected in our segmented sampling, ApproxSSD does not introduce bias and should have the same level of sampling accuracy as the random sampling. Future versions of ApproxSSD may focus on improving the sampling accuracy.

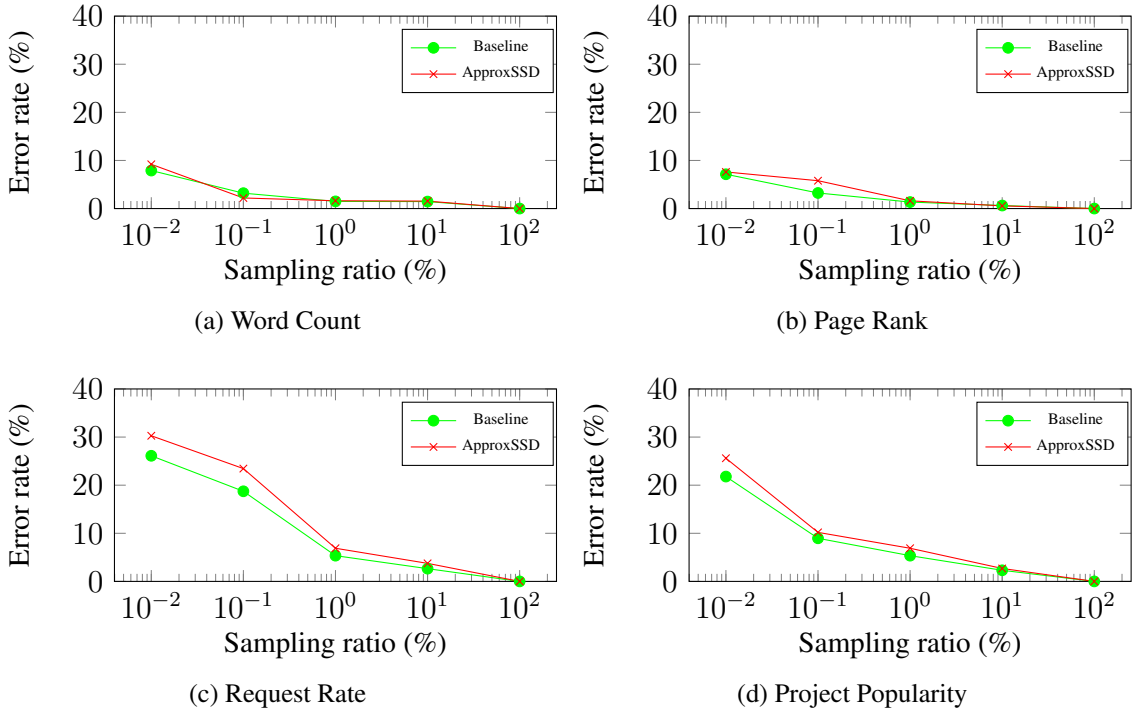


Figure 1.7: Error rate under different sampling ratios. The data point shows the average error rate for the top 1000 items.

Particularly, Figure 1.7 shows the error bar analysis for different applications. In the Word Count analysis, we select the top 1000 most frequently used words and use equation to calculate the

error-bar. We use the error ratio to indicate the differences. The error ratio is calculated by divide the error-bar by the actual experimental value. The graphs show the average error rate. In the page rank analysis, the top 1000 most linked pages are selected. The error rate of the total link times for the selected pages is shown in the graph. In the Wikipedia Request Rate analysis, the approximate number of requests in a time unit (an hour) is compared with the precise one. Then the average error rate is plotted in a time unit (a week). Finally, we conduct the similar comparison in the Popular Page analysis for the top 1000 most accessed Wikipedia pages.

The error rate of ApproxSSD, as shown in the figures, is very close to that of the baseline in every application. It is observed that Word Count and Page Rank analysis generally have lower error rate compared to Request Rate and Page Popularity analysis. This is because the content-based popularity analysis is more stable than the access pattern based popularity analysis.

As shown in Figure 1.7a and Figure 1.7b, the error rate of Word Count and Page Rank are barely affected by the sampling ratio. As shown in Figure 1.7c and Figure 1.7d, the Request Rate and Popular Page analysis have larger error rates which increase significantly when the sampling ratio is reduced to 0.01%. We can see that ApproxSSD has little impact on the overall sampling accuracy. And ApproxSSD can also adopt the technology used in BlinkDB to further improve the accuracy by considering data dimensions when save to disks.

Impact of Design Components

In this subsection, we study the overhead of ApproxSSD in terms of 1) workload balance; 2) I/O contention 3) and task drop. To study the performance of I/O system, we collect the I/O latency and I/O per second (IOPS) in our active storage driver. We also developed a round-robin I/O scheduler for comparison.

Workload Balance

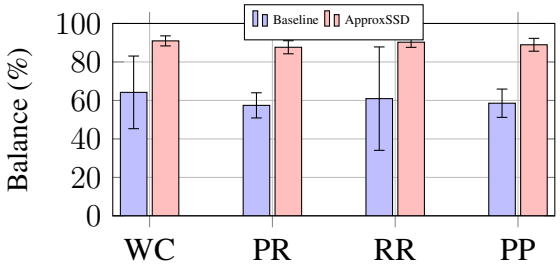


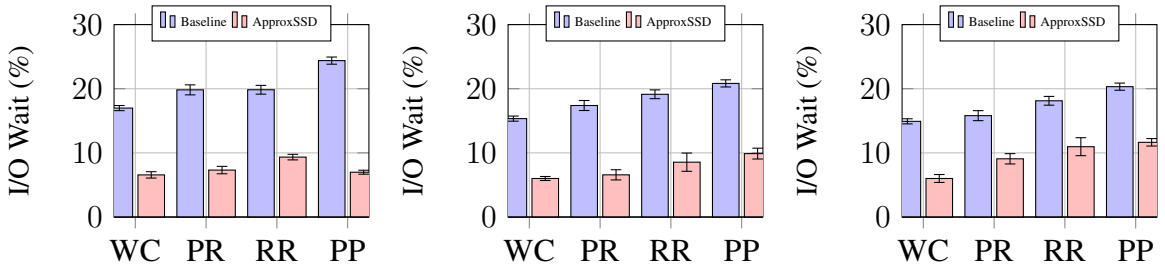
Figure 1.8: Workload balance analysis. The ratio of average load to maximum load, when using 1% sampling ratio.

To study the workload balance issue of ApproxSSD, we capture the maximum amount of data served by a single SSD. We use the standard iostat command in Linux to capture the amount of data read been from SSDs during the execution. Then the average value is calculated from 20 executions, and the max, average, and min values are plotted for each data point. We measure the workload load balance among disks using the ratio of the average amount data been served by disks and the maximum amount of data being served by a single disk, as shown in Figure 1.8. We can observe that the maximum load of the baseline is around 1.48 times larger than that of ApproxSSD, with a very high deviation. On the contrary, the SSD workloads under ApproxSSD are well balanced and very stable with a deviation less than 10%.

Contention & I/O Performance

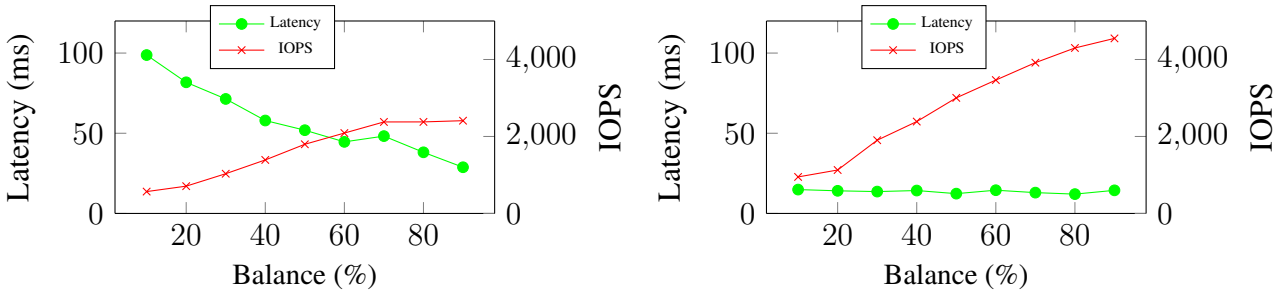
Loading the same amount of data from each SSD does not necessarily mean that we have exploited the I/O bandwidth. If tasks on the same disk are scheduled concurrently, the corresponding SSD will lead to an I/O contention, which results in a waste of CPU cycles. As shown in Figure 1.9, the percentage of I/O wait time is used to indicate the CPU and disk bandwidth utilization ratio. The

results come from the average of 20 executions. We can observe that the baseline wastes 3 times more CPU circles on the I/O wait compared to ApproxSSD.



(a) Sampling Ratio 0.1% (b) Sampling Ratio 1% (c) Sampling Ratio 10%

Figure 1.9: Percentage of CPU I/O Wait time when using different sampling ratios.



(a) Round-robin Scheduling (Baseline) (b) ApproxSSD Decoupled

Figure 1.10: I/O per second (IOPS) and latency under different scheduling strategy.

Figure 1.10 shows the IOPS and latency when using different I/O scheduling strategies. We spawn the same amount of the I/O threads in the baseline and ApproxSSD designs. The I/O threads are equipped with an unbounded I/O queue, and we submit all the I/O requests of the sampled data to these threads while submitting the jobs. In the baseline approach, the I/O scheduler sends requests to I/O threads in a round-robin way regardless of their data location. In ApproxSSD, we bind the I/O threads to disks, each thread only handles requests to the corresponding disk. The results come from the average of 20 executions, We can see clearly that there is a correlation between load

balance and contention. As the load balance increases, the IOPS increases under both designs. While the I/O latency of the baseline design reduces and the latency of ApproxSSD consistently stays very low. Figure 1.11 shows a sample of latency over time, which indicates that ApproxSSD has more stable latency compared to the baseline design.

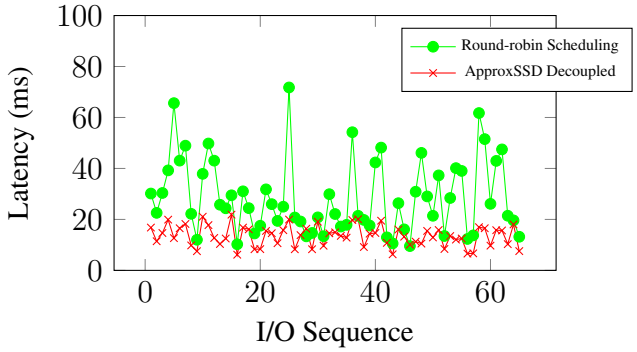
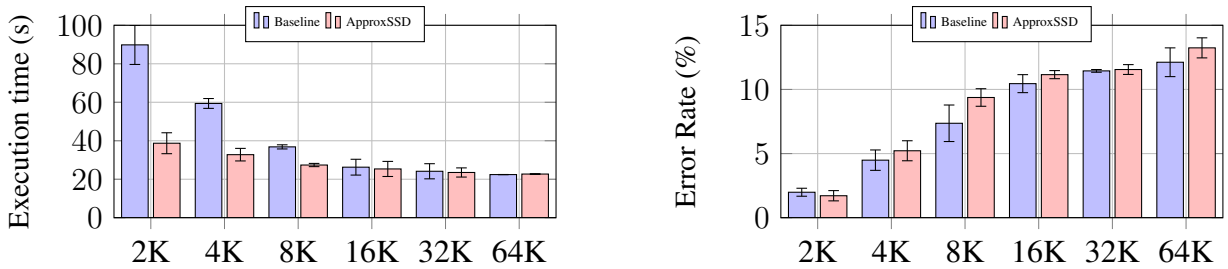


Figure 1.11: Stability of I/O latency.



(a) Execution time under different partition size.

(b) Error rate under different partition size.

Figure 1.12: Impact of partition size.

As we discussed, the size of sampling unit is one of the causes of I/O inefficiency. Figure 1.12 compares the total execution time and the approximate error rate under different partition size. The results come from the average of 20 executions. Results show that there are no significant differences in the execution time when using partition size larger than 64KB. As the partition size decrease to 2KB, ApproxSSD turned out to be 2x times faster than the baseline solution while

providing comparable approximate accuracy. It is worth noticing that as the partitions size goes beyond 64KB, the disk I/O will then become more sequential thus result in higher I/O throughput. However, the approximate error rate will be larger, and make the result useless for applications.

Performance Breakdown

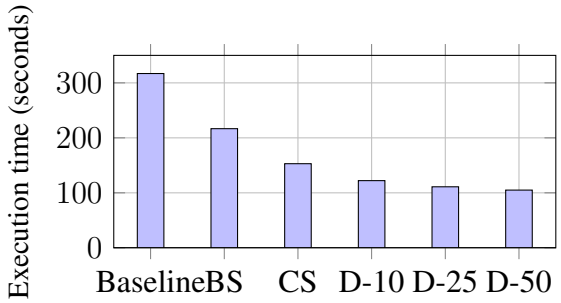


Figure 1.13: Performance Breakdown

In this experiment, we show the performance breakdown of each design components. We develop flag to enable/disable balanced sampling (BS), contention free scheduling (CS) and different task dropping ratios including 10% (D-10) 20% (D-20) and 50% (D-50). In the baseline experiments, we use random sampling. In the balanced sampling we only balance the workloads among SSDs. In the contention free scheduling we balance the workloads as well as use decoupled I/O threads for each disk. In task dropping experiments we further add different task dropping ratio to reduce the impact of unstable SSD performance. We use a fixed 10% sampling ratio and WordCount workload in each experiment. We run each experiment 20 times to get the average. Figure 1.13 shows the total execution time when different flags is enabled. All the percentage of the execution time reduction used below are compared to that of the baseline. The balanced sampling along can reduce the 37.5% execution time on average. The contention free threads can further reduce 14.2% execution time on average. In the case of using 50% dropping ratio, we sample 15% of the original

data, then drop 50% of the sampled data during the execution, so that we still process 10% of the original data. As the results show, the execution time can drop by 9.8% when using the dropping ratio of 10%. Then it drops another 5.2% if further increase the dropping ratio to 50%. However, as the task dropping ratio increases, the data been processed could be more skewed to the SSDs with better performance.

Discussion & Related Work

In this section, we discuss the related work and the requirements, challenges and solutions for effectively deploying ApproxSSD in real systems.

Approximative Computing

Data-intensive computing frameworks usually handle large amounts of data and require fast data retrieval to support applications, such as intelligent machine learning [32], graph processing [33, 34, 35, 36]. To cope with the explosive growth in data size and facilitate near-real-time analysis, many works have been proposed to reduce the input data volume, e.g., data transformation and sampling. The data transformation technique utilizes a compressed format to represent the original data, for example, the random projection is used to reduce the dimensionality of a matrix [37], and multi-probe locality sensitive hashing [38] is used in similarity search [39]. Data sampling, on the other hand, selects only a portion of the input data for processing and typically has low overhead [11, 13, 12, 15]. Due to its simplicity, data sampling has promoted the deployment of approximation in big data applications.

Scaled-up SSD Servers

Many recent works [4, 5, 6] propose efficient frameworks that can run big data analytics and machine learning algorithms on a single powerful server equipped with an array of SSDs. The direct benefit of running analytics or machine learning jobs on a single server is the removal of the heavy communication cost in distributed frameworks. Indeed, it has been shown in [40] that the majority of real-world analytic jobs process less than 100 GB of input. However, to process the entire dataset, applications have to repeatedly switch the data between memory and external disk array [41].

Without any moving parts, SSDs are expected to provide fast random read accesses and low I/O latency [42]. However, counter-intuitively, contemporary SSDs deliver unstable performance due to their distinct electrical characteristics and complex firmware design [7, 43]. 0.6% of the time, an SSD is more than 2x slower than its peer drives in the same array [44, 45]. Note that the worst-case latency of fully-utilized SSDs is much worse than that of HDDs due to garbage collection invocations in SSDs [18]. It is also reported that the random read performance can be even worse than random write performance [18]. Even when the workload is sequential read, the SSD performance can degrade with NAND flash cell aging and the increasing number of I/O requests [18]. Moreover, the concurrent read and write will degrade the performance further [20]. Additionally, SSD garbage collection can increase latency by a factor of 100 [46]. “Fast” and “slow” pages exist within an SSD [47], thereby causing uneven read/write latencies. ECC correction, read disturb, and read retry are also factors of performance instability [48]. In the task dropping design, additional data is sampled from SSDs which results in extra read requests to the SSDs. The read requests have little impact on the SSD lifespan compared to the other operations on the SSD [49]. Thus, it does not noticeably increase the cost of using the SSD array.

Local Storage System

It is not uncommon for storage systems to expose the data layout information in the pursuit of better performance in data analysis jobs [50, 51, 52, 53, 54]. There is also some recent work to expose the internal parallelism of SSDs in filesystem or block I/O [55, 24, 21]. Our work further exploit the storage level parallelism in the application layer. However, contemporary storage systems introduce many advanced techniques such as data compression and data deduplication. Thus, capturing the data layout information at the application level becomes even more complex and may change at runtime. Fortunately, the implementation of ApproxSSD only requires information that relates to the internal parallelism of the SSD arrays, such as data-to-device mappings. Accordingly, it is sufficient to support ApproxSSD through adding a mapping query function in existing storage systems.

To optimize the mapping query performance, the metadata in ApproxSSD is cached in memory. When partitioning 1TB data with the partition size of 8KB, we approximately need 1GB of memory for the metadata. If we implement data sampling, the size of the metadata to be cached will be further reduced. To the best of our knowledge, it is not common for a single node to store hundreds of Terabytes. Therefore, we can manage storing metadata in memory to both reduce the write traffic and improve the read performance. However, if we do need to process hundreds of Terabytes in a single node, we can first split the data to several Terabytes then process the files separately. The metadata is generated in the write threads in background. Only the initial portion of metadata is loaded from disk not the entire metadata set. Thus, the metadata in ApproxSSD does not introduce significant overhead.

Storage Virtualization

Virtualization is another widely adopted technology to consolidate servers. It brings many other potential optimizations, e.g. system wide data compression and deduplication, dynamic resource and performance allocation according to the workload. However, the sharing of storage system among the co-located virtualization environments incurs performance interference between each other [56, 57]. Without exposing the storage level parallelism to ApproxSSD, there is not enough opportunity for performance optimizations. In fact, this is not limited to ApproxSSD; state-of-the-art I/O protocols, e.g., NVM Express (NVMe), provide multiple submission queues that are visible to the SSD controller, thus without assuring such parallelism is exposed directly to the guest VM, it is challenging for the guest VM to optimize for SSD performance. To enable the performance gains from ApproxSSD in the presence of storage virtualization, we need to deploy a set of techniques. First, allowing the VM to query the host/hypervisor to directly interact with the block device, to obtain the actual device mappings (only those related to it). Second, any changes in the actual mappings are frequently propagated to the VM through an interrupt or special device driver (similar to Memory Ballooning driver). Third, using independent I/O threads to consolidate the requests from the hypervisor. Finally, providing a LDD-based storage virtualization interface for applications. Accordingly, ApproxSSD that runs on the hypervisor can get the data layout and leverage it effectively to achieve data-layout aware task scheduling.

Heterogeneous System & Data Skew

The write scheme of ApproxSSD is depending on the SSD status (e.g. if it is performing garbage collection). In a homogeneous system, we expect that the data will be eventually evenly placed among SSDs. On the other side, in a heterogeneous system, where SSDs from multiple vendors are equipped, a long-term execution of ApproxSSD may result in a skewed data placement. However,

the unbalanced data writing will not hurt the overall execution performance as ApproxSSD tends to write more data to more powerful SSDs. In the meanwhile, by introducing skewed data sampling, which takes different SSD performance into consideration, ApproxSSD would further improve the sampling performance.

Distributed Data Processing

Shared-nothing software architecture, such as Hadoop [26] and Spark [8], are becoming the de-facto standard for big data processing. ApproxSSD is a good compliment to the Hadoop ecosystem. As the Hadoop Distributed File System (HDFS) is mainly optimized for large sequential I/O, loading even 8KB data from a chunk would need to read and parse the whole chunk which is typically 64MB. In the meantime, reducing the chunk size would increases the size of the metadata and result in too much overhead in the NameNode. As a result, one of the most feasible ways to improve performance is to combine the function of ApproxSSD in the DataNode to support random sampling and leverage the I/O parallelism in SSD Array. We could then redesign the task executor to avoid I/O contentions.

Meanwhile, with the current implementation, the following options are available. First, ApproxSSD can potentially be incorporated into existing architecture as a local executor. Second, we can develop a data layout task grouping strategy that combines multiple co-located tasks into one. Third, we can launch a remote data loading task through ApproxSSD, and perform asynchronous data transfer between nodes, if remote data accessing is unavoidable.

CHAPTER 2: ARCHSAMPLER: ARCHITECTURE-AWARE MEMORY SAMPLING LIBRARY FOR IN-MEMORY APPLICATIONS

With the explosive rate of data growth, the limited scalability of the DRAM technology defies the performance potentials for in-memory applications. Fortunately, emerging non-volatile memory (NVM) technologies, such as Phase-Change Memory (PCM) and Memristor, are promising candidates for replacing DRAM. Emerging NVMs are very dense, hence promise large capacities. Additionally, NVMs are non-volatile, thus enable persistent applications and byte-addressable files. Both, density and persistency, are key enablers for in-memory applications. On the other side, emerging NVMs are slower than DRAM, thus optimizing for locality and avoiding contentions are key aspects to unlock the NVM performance.

In this paper, we study the impact of memory contentions and architecture-oblivious implementations on the performance of sampling based in-memory approximation. Sampling has become an imperative technique used to accelerate big data processing, especially in today's emerging in-memory computing. However, we observe multiple of times slow-down for naive and default implementations of in-memory data sampling. Accordingly, we propose *ArchSampler*, an architecture-aware sampling library. The main idea is to exploits free choice of data samples to dynamically select which bank as a host to serve memory requests. Hence, ArchSampler enables efficient and high performing sampling through employing its knowledge of the NVM architectural details to maximize data locality and avoiding inter-thread contentions. Our evaluation shows that ArchSampler can achieve up to **1.62** speed up (*1.20* on average) for different in-memory applications.

Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as Phase-Change Memory (PCM)[58, 59] and Memristor[60], are promising candidates for building future memory systems[58, 61]. Compared to DRAM, emerging NVMs promise high densities, near-zero idle power and persistent applications [62, 63, 58, 59]. While orders of magnitude faster, similar to flash-based Solid-State Drives (SSDs), emerging NVMs enable persistent data storage, hence allow hosting filesystems and persistent data applications. Accordingly, applications that process a large amount of persistent files, such as in-memory database systems and big data applications, are expected to benefit heavily from the deployment of emerging NVMs. Given the high density of emerging NVMs, it is possible to completely host the filesystem, hence all the files, instead of having them frequently swapped in and out between the memory and a much slower storage device, e.g., SSDs.

In many cases, instead of completely processing all data, it is sufficient to do sampling to infer key characteristics about the data. Sampling has been proven to be an efficient yet accurate way to solve real world problems[27, 9, 64]. For instance, in a recent work [64], the authors find that sampling only 10% of the original data set can be done with less than 5% accuracy loss for major data analytics applications. We expect sampling applications to become more common with emerging NVMs, given the huge amount of data NVMs can host. With SSDs, sampling is typically done through obtaining samples from a huge file, copy them to main memory (e.g., DRAM), and finally process them.The sampling process involves accessing slow SSD drives and copying the data to DRAM. A process that can waste tens of microseconds for each sample.

Future systems with NVM-based main memories can directly host huge files, hence enable in-place sampling and processing of files' data. Recent Linux implementations of filesystems started to support Direct-Access for Files (DAX) to facilitate direct accesses to NVM-hosted filesystems [65]. Figure 2.1 depicts sampling and processing data in future NVM-based main memory systems.

As mentioned earlier, given the huge amount of data NVMs can store, processing all the data of the files can be replaced with fast NVM sampling. On the other side, since emerging NVMs have small latencies, the contention of accesses on the memory can incur significant overheads. The contentions could result from unbalanced accesses to the NVM memory banks and row buffer conflicts. Compared to DRAM, this overhead is much more significant; the actual NVM access latency incurred due to row buffer conflicts is multiple of times higher than DRAM, however, row buffer hits are as fast as DRAM. Moreover, the data sampling on NVM will generate more random memory request which in turn reduces row buffer hits. Our evaluation shows that the performance overhead of internal NVM contentions can reach up to 39%. Fortunately, we observe that sampling applications can exploit their *inherent data choice liberty* to more efficiently utilize the internal architecture of emerging NVM technologies.

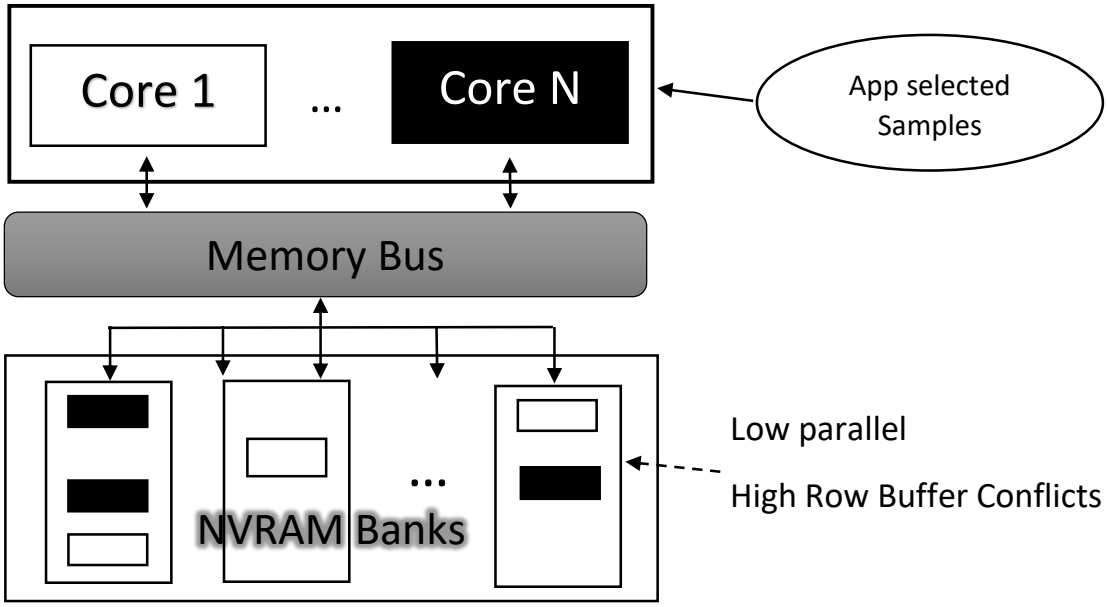


Figure 2.1: Data sampling on systems with NVM-based main memories.

In this chapter, we develop *ArchSampler*, a software framework library that enables architecture-aware data sampling. We discuss the design, challenges and potentials for this type of abstractions.

ArchSampler primarily achieves two main objectives. First, reducing the unbalanced load on different NVM banks. Second, maximizing the row buffer locality on each bank by reducing bank conflicts. Unfortunately, there is a lack of work that aims for architecture-awareness in data sampling applications. To the best of our knowledge, this is the first work to propose architecture-aware sampling framework for emerging NVM technologies. We strongly believe that this work serves as a first step towards hardware-awareness in big data applications.

To evaluate ArchSampler, we use the Structural Simulation Toolkit (SST)[66], a widely-used detailed architectural simulator. We run several data sampling algorithms derived from real world applications. Our results show that ArchSampler can improve the performance of the default implementations by up to *1.62* (*1.20* on average).

Motivation and Analysis

In this section, we motivate our bank-aware data approximation approach by showing how applications sampling on NVMs will cause more interference than other applications, and how leveraging the ”inherent data choices” of data sampling in bank selection can ameliorate this problem.

To enable scale-out data analysis, the data analysis frameworks usually slice the data into multiple splits and use shared-nothing threads to process that data in parallel [67].Complex data analytics jobs or queries are then translated into multiple iterations of the map, reduce and join operations. In addition, data sampling techniques are nowadays widely adopted to shrink the input data and generate fast and approximate results. Because of this, improved processing performance of sampling plays a key role in responsive analytics jobs.

NVM-Based memories are logically organized as groups of banks within a single or multiple ranks. NVM banks can service memory requests in parallel. Bank-level Parallelism (BLP) is used

to mask the latency of accessing memory through servicing memory requests in parallel. However, suboptimal access patterns can result in contention and unbalanced loads among banks. Therefore, the performance of NVMs is sensitive to the data access pattern.

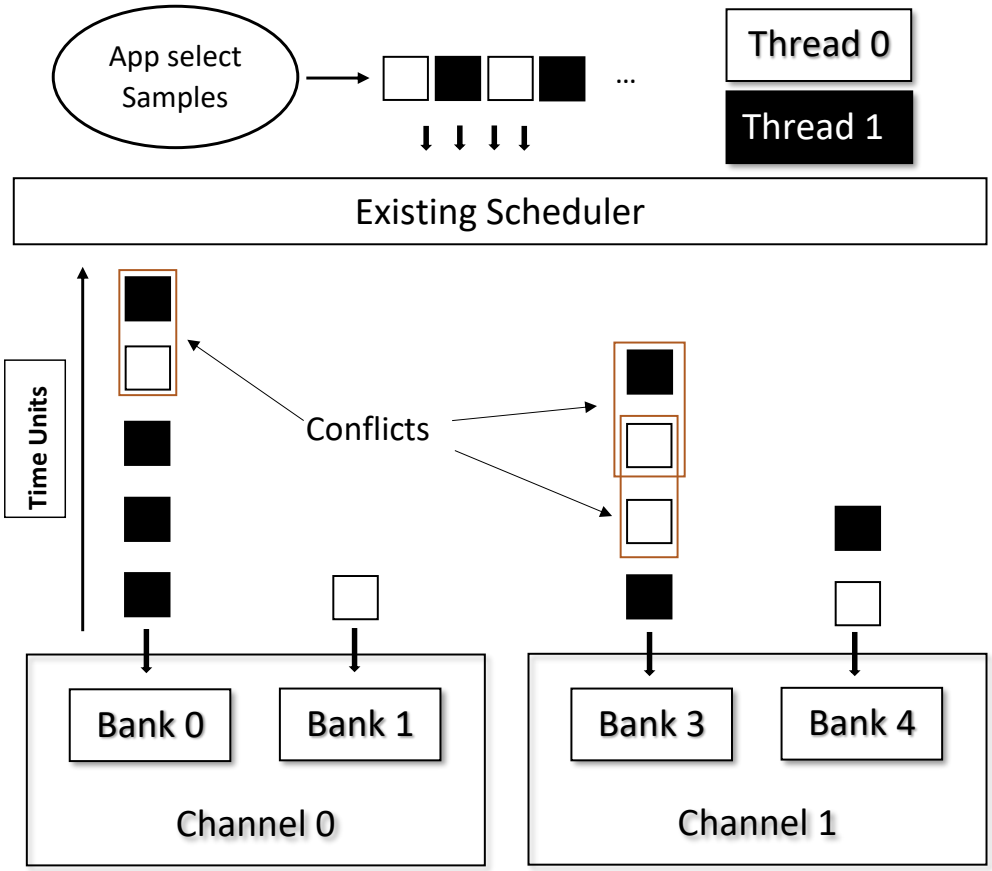


Figure 2.2: Conventional Random Sampling.

Data analytics applications retrieve and process results from large sets of data. Thus, the overall load imbalance issue between banks is usually not significant in these applications. This relies on the fact that larger datasets can be more easily stripped out to banks evenly. However, as shown in Figure 2.2, in the case of sampling, it is more likely that the workloads data will be skewed among the banks because of the randomness in memory requests. Moreover, when doing parallel data processing, multiple threads may compete for the same bank, resulting in performance degradation.

Figure 2.3 shows a conceptual example that application submit all the data to ArchSampler. The ArchSampler then leverage the multiple choices in sampling and the architecture information to balance the workloads in each bank and avoid thread stall. In the following, we will analysis the problem of conventional random sampling when NVMs are involved.

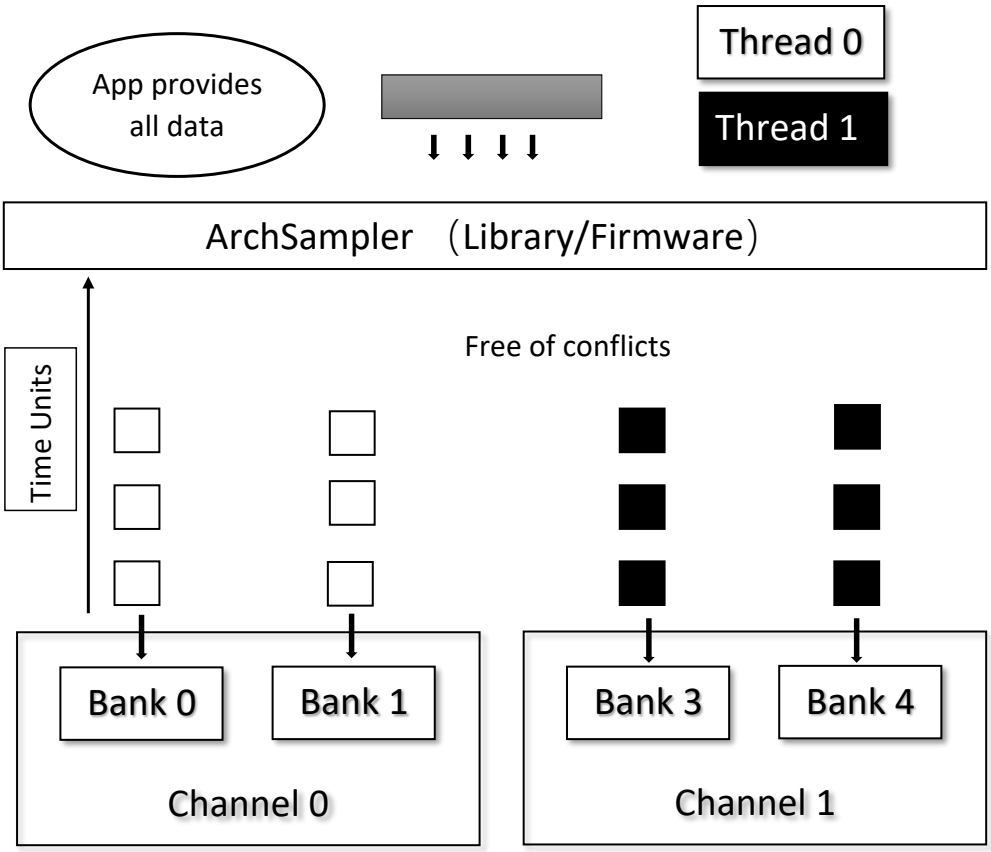


Figure 2.3: Conceptual example showing benefits of Bank-aware Sampling.

Load Balancing: In the scope of our paper, load balancing can be defined as the ability to evenly distribute concurrent memory requests among memory banks. The amount of load imbalance depends heavily on the type of application:

Content-independent applications: In these applications, the access to data usually has nothing to do with the content. The MIN, COUNT, AVG, SUM, PERCENTILES, and MAX are the most

popular functions [9]. These functions traverse all the selected data to deliver an aggregated result. Thus, the size of the input data plays a key role in determining the workloads. In Figure 2.4, we perform random data sampling while fixing the maximum number of rows sampled from one bank and the average number of rows need to be sampled. The results show that, as the sampling ratio goes smaller, the load imbalance issue will become more severe. Unfortunately, it is common for sampling based data analysis to use a sampling ratio smaller than 2% [25, 68].

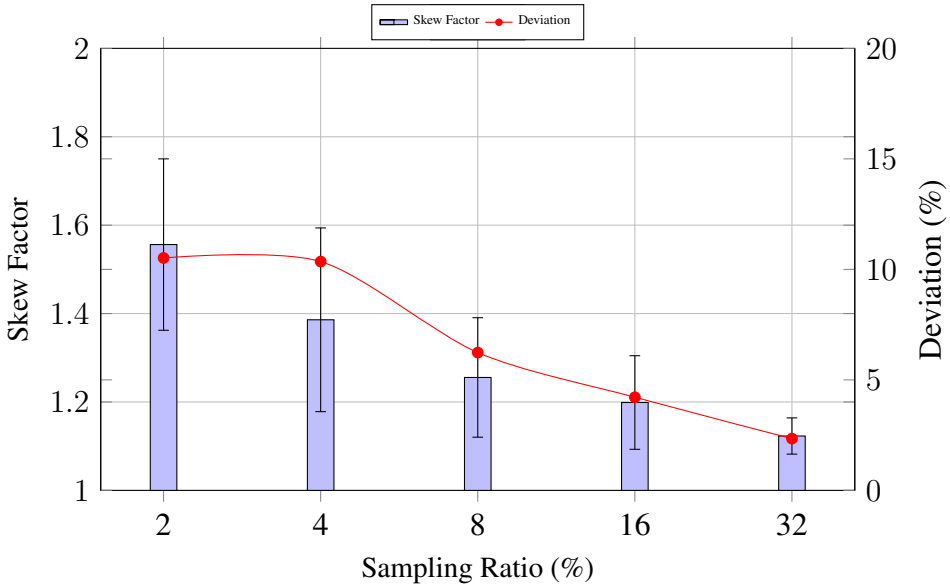
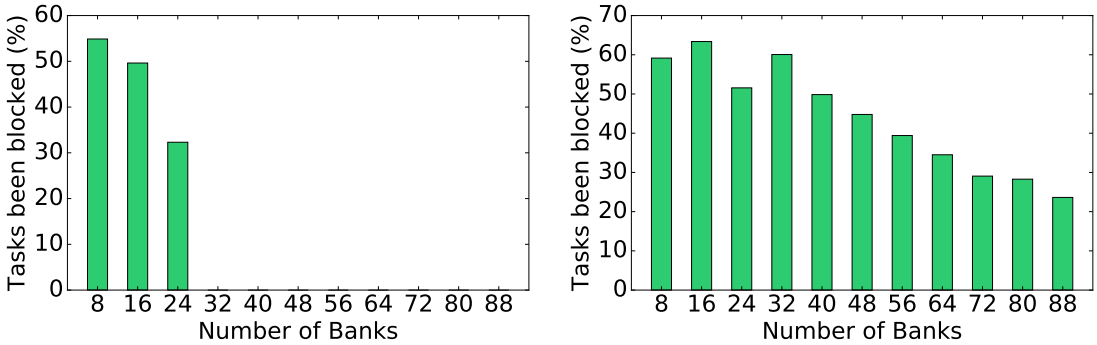


Figure 2.4: Workload imbalance issue (Skew factor = max/avg NVM rows requested from banks).

Content-dependent applications: In these applications, such as sorting and graph-processing, the access to data are greatly determined by the content to be processed. As a result, even processing the same size of the input data may requires a different amount of time. Among these, perfectly balancing the load is usually impractical. Data sampling technique is used to get a fast estimation of the content distribution, e.g. sampling sorting [69] and graph sampling [70]. Then, a relatively more balanced data partition strategy can be performed.

Contention: In order to leverage multi-core CPUs, shared-nothing software architectures are

widely used, with the expectation that no contention or data racing will occur among the threads. However, when multiple threads compete on accessing data on similar banks, banks' row buffers become less efficient and many requests are serialized due to bank conflicts. Accordingly, the overall performance can be significantly degraded when contentions occur frequently. In Figure 2.5, we did the computational analysis for the possibility of contention issue. The results show that, when sampling ratio is lower than 10%, up to 60% of the memory request will be blocked.



(a) Sampling Ratio 100%. (b) Sampling Ratio 10%

Figure 2.5: Percentage of tasks been blocked due to bank-level I/O contentions.

In summary, our key strategy in this work is to perform ① load balanced data sampling where samples are picked from specific banks ② contention-free task scheduling where each thread accesses only a specific set of banks.

Design and Optimizations

In this section, we discuss our design of ArchSampler along with the challenges, requirements and the potential optimizations.

Load-balanced Sampling

In order to leverage the parallelism among banks, applications need to distribute the requests to the memory as evenly as possible, especially for the more time-consuming write requests. In this way, each bank is equally busy as others. Therefore, we need to create a bank-aware data selection and memory allocation scheme that balances the load distribution and achieves efficient resource usage.

As explained above, balancing the load among banks is highly dependent on the type of application. Accordingly, ArchSampler implements key sampling functionalities that can be used for various types of applications. In the following, we describe how each of such key sampling functionalities is designed and implemented for approximative query.

To balance the load of a simple approximative query, an equal amount of data from each bank should be selected. The sampling-based approximation is flexible on which data should be selected as samples. While doing uniform random sampling is the most obvious way to do the approximation, there is no guarantee that the load will be evenly distributed across banks. To optimally balance the load, we restrict the amount of data to be selected from each bank. At first glance, such biased sampling is expected to affect the accuracy of approximation; however, we find that ArchSampler's ability to freely choose data within banks can achieve sufficient coverage and selection diversity. In fact, in most of our experiments, ArchSampler provides similar accuracy to uniform sampling, and even slightly better accuracy in some cases due to the increased diversity of samples.

To enable balanced sampling, we do the following. ① We split the input data into partitions, each equal to the size of the memory row. ② We group these partitions according to their bank IDs. ③ A sampling function is initiated for each bank and an equal number of partitions is then selected

randomly. As a result, the potential hot spots on banks can be avoided. In a heterogeneous system, i.e., a system with multiple memory technologies, the number of sampled partitions can also be weighted by the performance of banks.

Contention-free Threading

Shared-nothing architecture is widely used in data parallel analysis frameworks. The parallel threads execute tasks that are assigned to them without any locking requirements to avoid data races. The communication between threads is explicitly done at the synchronization step, typically wrapped by a reduce or a join functions. Thus, each data partition is accessed by no more than one thread. However, in contemporary design, applications have no way to get the bank information of the allocated memory. The data to be accessed by the threads will span multiple banks. Because of this, multiple threads may have contention if they request data on the same bank at the same time.

To mitigate the contention issue, we restrict the partitions assigned to one thread to span only one bank. In case we have T threads and N banks (both T and N are in power of 2). The strategy is straightforward if T is equal to N. However, in case T is less than N, each thread is assigned data from N/T banks. In contrast, if T is larger than N, there is no way to completely avoid the bank contentions between threads; however, we minimize the contention by restricting the maximum number of threads map to each bank to only T/N threads. Given that modern DIMMs come typically with 32 banks, while modern processor sockets have only 8-16 cores, the latest case is rare. In the case of multi-socket systems, it is typical to attach DIMMs to each processor socket, thus ArchSampler can assign the threads of each socket to the banks of the local (close) memory.

Bootstrap Error Estimation

The results of sampling-based approximation are affected by noise and sampling errors. To address this issue, the least efficient way would be through collecting more samples. However, this can hinder the responsiveness of the analysis tasks and is not always practical. To assess the quality of the estimation and give error bars with confidence, we need to determine how the results are distributed. Bootstrap resampling is one of the most common methods to draw the empirical distribution for data to be sampled by using the data itself [71]. Because of its simplicity and effectiveness, it can be used in almost any type of data and application [72, 9].

To get the approximation and the error bar with x% confidence interval, the process of bootstrap resampling is generally as follows. First, resample the data set several times to obtain different approximate results. Second, trim $\frac{1-x}{2}\%$ of the approximate results from the lower and upper ends. For example, when sampling data 10 times to obtain approximate results with 80% confidence, we trim the most and least significant results. Finally, after excluding the results from the previous step, we find the summary of the statistics by calculating the mean, minimum and maximum values.

Implementation: Putting It All Together

To better understand how ArchSampler can be used by applications, Figure 2.6 depicts the flow of an application using ArchSampler. ArchSampler framework takes an architecture specification file as an input. The ArchSampler library can be provided with the specifications file path. The architectural specifications can include information such as: the number of banks, the number of ranks, row buffer size and the number of NUMA domains. Note that such architecture specification files are common in state-of-the-art frameworks and libraries. For instance, the Message Passing Interface (MPI) library maintains a system configuration file that is used to optimize the processes'

assignment to cores. Typically, the memory configurations are recognized at the time of boot up. For instance, in the BIOS setup wizard, the user can specify the memory mapping, interleaving and read the different memory controller timing parameters. However, another approach will be through micro-benchmarks to quickly identify the configurations.

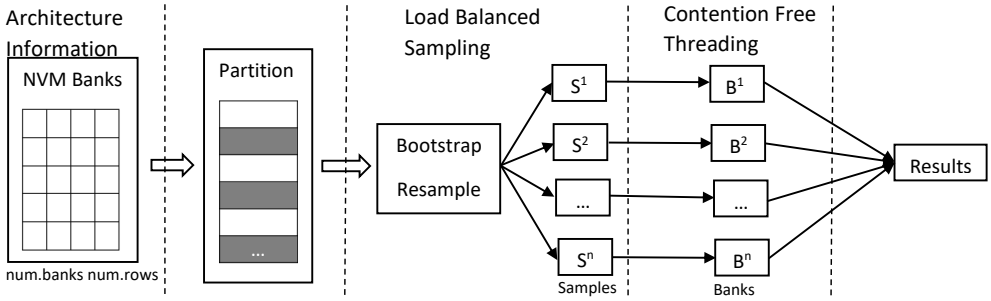


Figure 2.6: Overview of ArchSampler framework.

The memory specification file is used by ArchSampler to identify the data partitioning scheme that achieves load balancing and minimizes threads contention. An application can directly call the ArchSampler framework to do sampling functionalities, such as: approximative query and graph sampling. ArchSampler is a multi-threaded library that abstracts the complexity of bank-aware load-balanced and contention-free sampling from the programmer. Modern data analytics applications can be ported to use ArchSampler to do efficient sampling.

Applications can repeatedly call ArchSampler to re-sample the dataset in case the original dataset is being frequently updated, or repeatedly do re-sampling until an acceptable error is achieved.

Evaluation

In this section, we first introduce our experimental methodology, including the workloads and the simulator assumptions. We then present the evaluation and analysis.

Methodology

To evaluate our design, we use the Structural Simulation Toolkit (SST) [66], a widely-used detailed architectural simulator. SST provides detailed timing models for the memory system and other major architectural components (processors, caches, memory). Most importantly, SST has an integrated detailed model for modern NVM-Based DIMMs, Messier [73], that we use as our main memory. In our evaluation, we use SST's Ariel component for emulating an x86 processor. We model a three-level cache hierarchy with 32KB L1, 256KB L2 and a 2MB shared L3 cache. For the main memory, we model a 16GB PCM-based DIMM (Messier component). Messier models in detail the asymmetric PCM read/write latencies, row buffers, write buffers' threshold-based flushing, and power-constrained writes to PCM banks. We use a typical 8KB row buffer size. We vary the number of banks for most of the experiments. However, we use 32 banks by default. For the PCM read latency, similar to recent studies [74, 75], we use 150ns as our default latency. For PCM write latency, similar to [76], we use 500ns write latency of PCM cells. Table 2.1 shows the detailed configuration of the simulator. Our choice of using SST allows us to run simulations with reasonable speed while modeling all important aspects with sufficient details. Accordingly, we could ran all of our benchmarks from start until completion.

To mimic real-world sampling applications, we developed standalone multi-threaded microbenchmarks. Our implemented microbenchmarks resemble real-word workloads, such as: word-count, calculate the average rating and finding the elements with highest average rating on Amazon reviews and rating datasets[77, 78]. We also study two synthetic read and write workloads. A complete list of workloads and the corresponding algorithm and dataset are shown in Table 2.2.

At the beginning of each application, a large memory space is allocated (using malloc) and used to mimic a memory-mapped region from a directly-accessible file, i.e., DAX-based file. In future NVM-based systems, files' data can be accessed directly through the memory bus through a simple

`mmap` call at the beginning of the application. The only difference between the behavior of malloc (what we use) and DAX-based mmap (what NVM-based files use) is the behavior of the initial page faults of the pages; DAX faults will be handled partially by the filesystem layer. To avoid the impact of such variance, we exclude the page initialization stage from the execution time through starting simulation after initializing the malloc'ed region.

Table 2.1: Simulated system configuration.

Parameter and Configuration	
Processor	8 cores
Core	2GHz, 3 issue/cycle 16 max. outstanding memory requests
Clock	2GHz
I/D L1 cache	32KiB, 4 cycles latency
L2 cache	256KiB, 6 cycles latency
L3 cache	2MB 12 cycles latency
Memory Size	16GB
Number of banks	4, 8, 16, 32
Read latency (t_{RCD})	50ns, 150ns, 250ns, 350ns, 450ns (100, 300, 500, 700, 900 cycles)
Row buffer hit (t_{CL})	15ns
Write latency	500 ns (1000 cycles)
Scheduler	FR-FCFS prioritizing row buffer hits

The addresses allocated by malloc are virtual addresses as seen by the program. As our algorithms

work in close co-ordination with the banks' information, we use the virtual addresses returned by malloc requests to produce the banks' information; since we have 64B cachelines and the smallest page size is 4KB, the bits used for indexing maps (bits 6 to 10) falls within the page offset that is similar to the physical address. Note that this is the case for page-aligned allocations typical in filesystems' files. Note that our assumption of the sufficiency of virtual address holds upon two key requirements: page-aligned files and having a number of banks that can be indexed through the remaining bits of the page offset, i.e., up to 64 banks. If any of those conditions are not met, ArchSampler needs to be exposed explicitly to the virtual-to-physical mappings to efficiently allow bank-aware placement of samples. Such information can be furnished to ArchSampler through a system call; however, given the current trends of using 16-32 memory banks, we do not expect having more than 64 banks. Furthermore, given the trend of using huge pages (2MB or 1GB) with NVM systems, we expect a negligible overhead to retrieve such mappings.

Table 2.2: List of evaluated applications.

Application	**Algorithm**	**Datasets**
Word Count	Sum	Review text
Average Rating	Average	Review rating
Synthetic read	Synthetic	NONE
Synthetic write	Synthetic	NONE

The main memory (PCM) is logically divided into rows, with each row is equal to the size of the row buffer (typically 8KB). These rows are mapped to the banks to exploit row buffer locality for the accesses of open row/page. ArchSampler logically splits the allocated memory into rows, and later assigns each row a unique number, starting from 0 to $(memorysize/rowsize) - 1$. The rows are mapped to banks in a round-robin fashion. Thus, in order to get the corresponding bank from a row number, we use $row\%numberofbanks$. Before the test algorithm of the benchmark

application is triggered, ArchSampler is called to spawn threads in which each thread is assigned a set of rows to work on. We have implemented different ways in which these rows are assigned to threads:

Bank-aware contention-free sampling (ArchSampler): In this assignment, each threads is assigned a set of rows that belong to the same bank. For example, if we have a 4 banks, 4 threads and 16 rows scenario, thread-0 will be assigned rows 0, 4, 8 and 12, whereas thread-1 will be assigned rows 1, 5, 9 and 13 and so on. Meanwhile, thread-2 will be assigned the rows 2, 6, 10 and 14, while thread-3 will be assigned the rows 3, 7, 11 and 15. Therefore, due to this arrangement, no thread conflicts with other threads on requesting data from the same bank, i.e., thread-0 generates requests pertain to bank-0 only, thread-1 to bank-1, thread-2 to bank-2 and so on. Thus, achieving true parallelism. We consider this as the best case.

Load-balanced sampling: In this scheme, each sampling task is assigned an equal number of rows from each bank, hence balancing the loads across memory banks. Although this scheme can accomplish a fairly equal distribution of rows across threads, the sampling tasks from different threads can compete on the same bank resulting severe contentions.

Random sampling: The rows are selected and assigned to the threads in a random fashion without considering banks information and maintaining equal bank distributions. We consider this as the average case scenario and its performance can vary widely.

Bank-aware contended sampling (Synthetic Worst): In contrast with the other schemes, this scheme is synthetically designed to generate the worst case scenario. This scheme has all the threads competing to access the same bank at the same time, thus resulting in a heavy contention.

Once the threads are assigned the rows work on, we choose a sample size to work on and the benchmark algorithm is executed. All of our experiments are conducted on a 6-core Xeon server

with 64GB memory. For the simulated PCM memory, the size configured to be 16GB and the number of banks is varied from 4 to 32.

Analysis and Discussion

To quantify the performance improvements ArchSampler can achieve, we start with analyzing the content-independent applications, such as estimating the WordCount and Average Rating for the amazon review dataset. We then show the results for the synthetic read and write workloads and study the load imbalance issue. Finally, we study the content-dependent applications, such as graph processing.

Approximate Accuracy

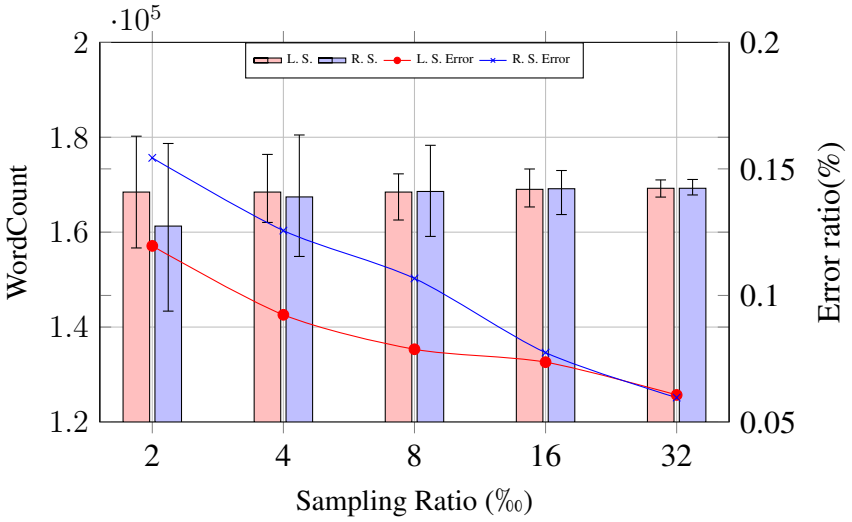


Figure 2.7: Results of WordCount under different sampling ratio. (L. S.: Load-balanced sampling with estimated error bound, R. S.: random sampling with estimated error bound, L. S. Error: error rate of load-balanced sampling, R. S. Error: error rate of random sampling.)

In this subsection, we focus on investigating the impact of bank-aware sampling on the accuracy

of the used approximative sampling techniques. In the baseline approach, we use uniform random sample strategy where the data is randomly selected regardless of its position. In contrast, the bank-aware sampling explicitly selects equal amount of data from each bank. We study two real word applications, WordCount and average rating. In the WordCount application, we count the appearance of a given word in the amazon review text file, whereas in the Average Rating application, we calculate the average rating of the amazon movie rating dataset. At first glance, we would expect biased bank-aware sampling to negatively impact the accuracy of estimated results. However, in Figure 2.7 and Figure 2.8, the results show that the bank-aware sampling achieves approximative results that are within a smaller error bound. Specifically, the error rate from the result of bank-aware sampling to the accurate result is $33 - 43\%$ smaller compared to the random sampling when the sampling ratio is 2%. This is due to ArchSampler's segmented sampling increased diversity of samples [79].

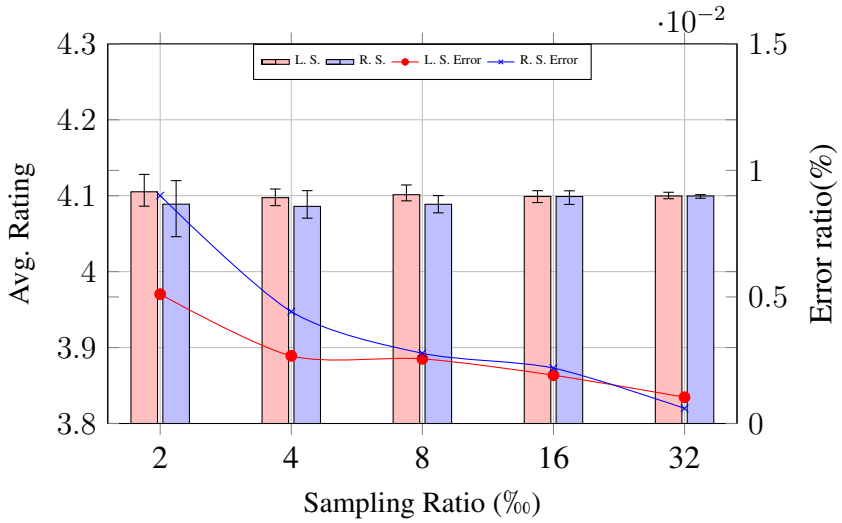


Figure 2.8: Results of Average Rating under different sampling ratio.

Performance of Approximate Query

To quantify the potentials of ArchSampler, we study its impact on the performance of approximate queries. Approximate query includes a variety of applications, such as SUM, COUNT, MAX, MIN, AVERAGE. In this part, we select 2 typical applications: WordCount for unstructured amazon review text file and getting Average Rating for structured amazon rating dataset. To investigate the impact of the number of banks on ArchSampler, we also vary the number of banks from 4 to 32, increasing by multiples of 2. Moreover, since different applications and datasets can tolerate varying levels of error, we vary the sampling ratio from 2‰ to 32‰, increasing by multiples of 2.

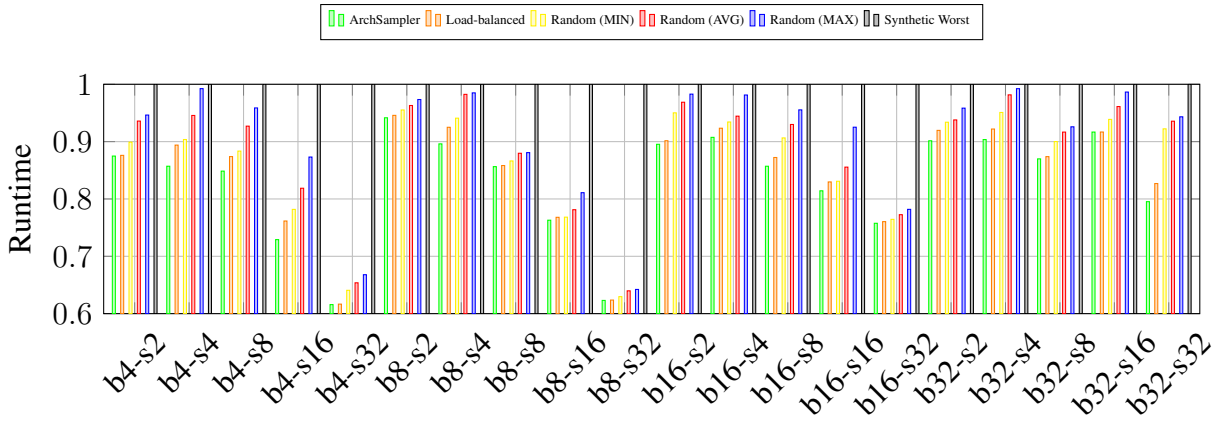


Figure 2.9: Normalized Runtime performance of WordCount. Note the performance is normalized to the synthetic worst case. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent m‰ sampling ratio.

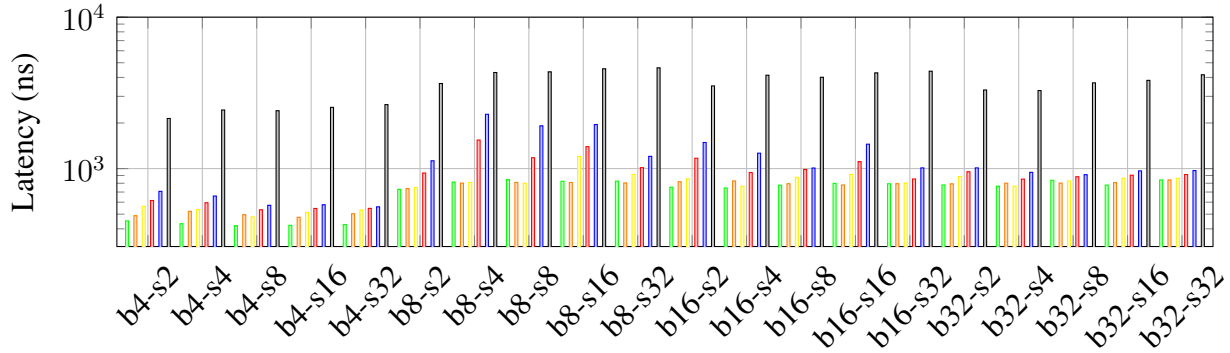


Figure 2.10: Latency of WordCount. Note the Latency is in logarithmic scale. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent m‰ sampling ratio.

In our experiments, we compare the four sampling and task scheduling schemes previously dis-

cussed. In ArchSampler, we use bank-aware sampling and contention-free task scheduling scheme. To study the effect of contention-free threading, we then use random task assignment scheme after doing the bank-aware sampling. In the random sampling, we select the data and schedule the tasks randomly. Because the performance of random sampling has high variance compared to other schemes, we repeat its runs for 10 times and collect the average, minimum and maximum values. In the synthetic worst case study, we managed to make all the requests go to the same bank, so that in theory we get the worst performance. For all of our experiments, we report the execution time and total memory latency as shown in Figures 2.9, 2.10 and 2.11. In order to place the data in one chart, we plot the execution time using normalized data and plot the total memory latency using logarithmic scale.

Figure 2.9 shows the normalized execution time for WordCount. The results show that ArchSampler can reduce the execution time by up to 38.4%, with the potential for an average of 16.9%, compared to the synthetic worst case. When compared to the random sampling, ArchSampler can reduce the execution time by up to 15.1%, with the potential for an average of 6.2%. Figure 2.10 shows the latency summation of the memory accesses for WordCount. The results show that ArchSampler can reduce the total memory latency by up to 97.4%, with the potential for an average of 76.5%, compared to the synthetic worst case. Compared to the random sampling, ArchSampler can reduce the memory latency by up to 60.5%, with the potential for an average of 40.7%.

For Average Rating, we also study the impact of ArchSampler on the execution time and memory latency. Figure 2.11 shows the normalized execution time for the different sampling schemes. The results show that ArchSampler can reduce the execution time by up to 28.9%, with the potential for an average of 14.1%, comparing to the synthetic worst case. While comparing to the random sampling, ArchSampler can reduce the execution time by up to 13.4%, with the potential for an average of 6.8%. Figure 2.12 shows the sum of the memory latency for Average Rating. The results show that ArchSampler can reduce the total memory latency by up to 83.9%, with the potential

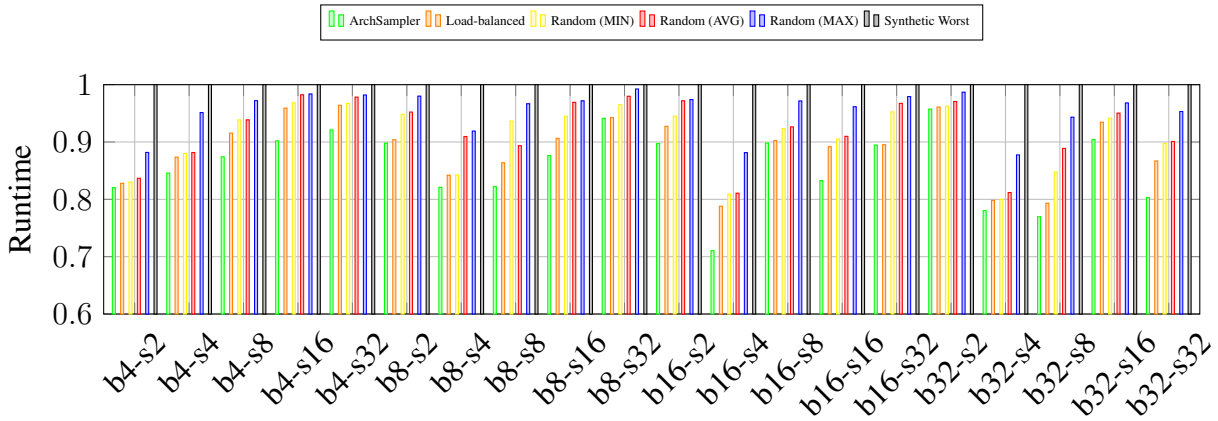


Figure 2.11: Normalized Runtime performance of Avg. Rating. Note the performance is normalized to the synthetic worst case. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

for an average of 80.6%, comparing to the synthetic worst case. While comparing to the random sampling, ArchSampler can reduce the total memory latency by up to 47.2%, with the potential for an average of 22.3%.

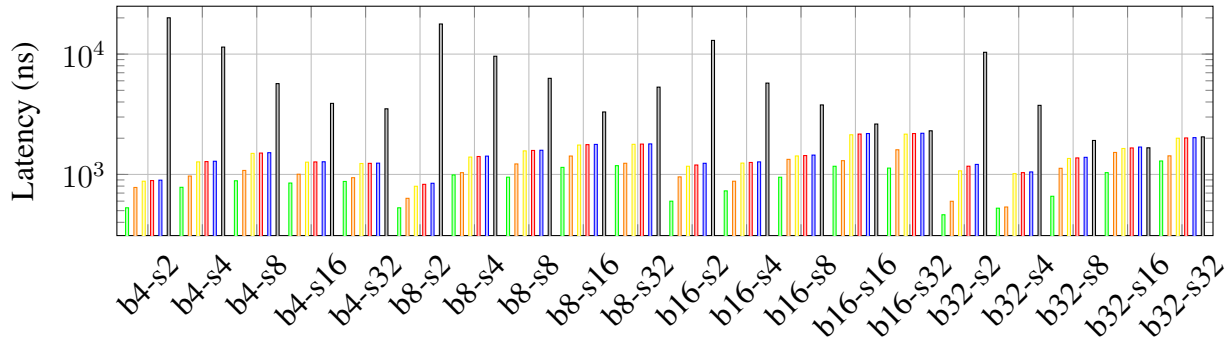


Figure 2.12: Latency of Avg. Rating. Note the Latency is in logarithmic scale. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

Load-balance Issues

To study the load imbalance issue, we have developed and tested synthetic read/write workloads. The synthetic workloads perform read/write operations on all the sampled memory rows. Figure 2.13 shows the execution time of the synthetic workloads. We collect the results by calculating the average performance of 10 complete runs. As we can observe from Figure 2.13, ArchSampler

can achieve up to 1.59 speed up (1.32 on average) for read workloads and up to 1.67 speed up (1.38 on average) for write workloads at various sampling ratios. Figure 2.14 shows the number memory requests served by each bank while using 4% sampling ratio for various synthetic workloads. The results show that ArchSampling can reduce the maximum load of banks by 21.1% to 32.4% compared to random sampling (26.4% on average).

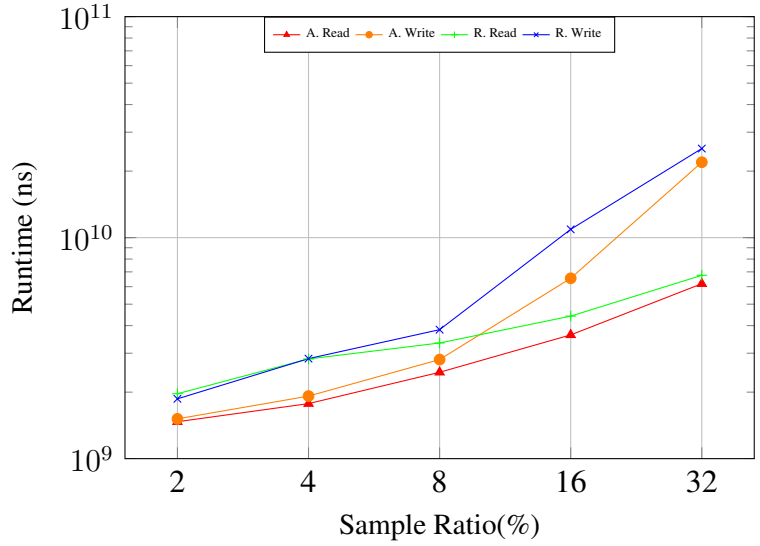


Figure 2.13: Runtime performance for Synthetic Workloads. Note the Y label is in logarithmic scale. A. stands for ArchSampler. R. stands for Random.

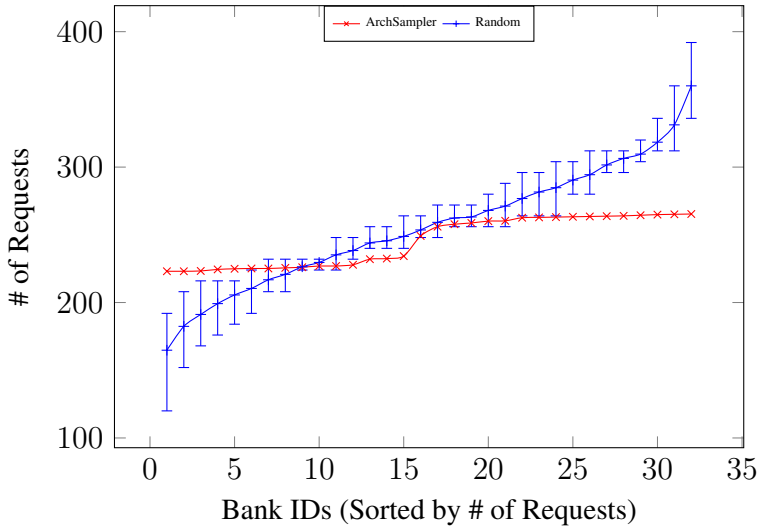


Figure 2.14: Load balance in Synthetic Workloads.

CHAPTER 3: A CORRELATION-AWARE PAGE-LEVEL FTL TO EXPLOIT SEMANTIC LINKS IN WORKLOADS

NAND Flash Solid State Disks (SSDs) are gaining tremendous popularity in today's storage market due to the unique erase-before-write feature of NAND flash, the Flash Translation Layer (FTL) in the SSD redirects the incoming writes to a free physical address and manages a logical to physical address mapping table. However, this induces significant performance degradation to the SSD. One of the main reasons is that current cache management in FTLs mainly focus on temporal or spatial locality. However, because of multiple levels of data buffers in the whole storage architecture, the locality of disk I/O is relatively low. What's more, the increasing capacity of SSD not only leads to mapping tables large in size, but also imposes high pressure on the efficiency of page-level address mapping.

To overcome this limitation, we propose Correlation-Aware Page-level FTL, a.k.a CPFTL, which exploits I/O correlations in workload. First, a correlation-aware mapping table is developed based on the correlation in read operations. Second, we propose a correlation prediction table to support fast mapping entry lookup in correlation-aware mapping table. Third, because the data that has been flushed to the disk by the host buffer has low chances to get reused in a short timeframe, we developed separate read and write caches to improve the cache hit ratio. Finally, we propose a skew-aware dirty entry index to improve the "in-page" locality aware dirty cache update and thus reduce the garbage collection overhead. We developed an emulator and prototype, being open sourced at: https://github.com/janzhou/SSD-Emulator. Our experimental results show that CPFTL can reduce the average response time by 63.4% for read dominant workloads and 32.9% for transaction workloads.

Introduction

NAND Flash Solid State Disks (SSDs) [7] offer several key advantages over traditional mechanical Hard Disk Drives (HDDs), such as low access latency, low power consumption, and excellent shock resistance. As the cost per bit has been decreasing recently, SSDs have been deployed as the primary storage in mobile devices, personal computers, and large scale enterprise systems [80]. However, the idiosyncrasy of NAND flash, such as erase-before-write, slow erase operation, and limited endurance, have hindered the wide-scale adoption of SSD. As a core component in SSDs, the Flash Translation Layer (FTL) maps a virtual address to physical address, thereby hiding the complexities of flash. It maintains a mapping table that translates a logic page number to a physical page number in SSD and redirects new writes to a free page and invalidates the old page.

To accelerate the address translation, onboard RAM is utilized to cache the mapping table. As the capacity of SSDs continues to increase, the mapping table has been growing so rapidly that it cannot be fully accommodated by limited RAM. Therefore, a full mapping table is divided into translation pages. However, frequently migrating pages between RAM and SSD can lead to significant degradation of SSD performance. This issue becomes even more severe for a page-level FTL [43, 81], which requires a larger mapping table in order to achieve a higher random access performance.

Unlike synthetic benchmarks, real world I/O workloads exhibit more random characteristics and are more challenging to achieve higher performances. Fortunately, real world workloads are not truly random, and in fact considerable latent semantic links of data remain in most random I/O workloads. Specifically, in storage systems, if two or more data blocks are accessed together, these data blocks are defined as being correlated [82]. For example, in generating a web page, multiple files and database entries will be accessed, thus their corresponding on-disk data blocks are correlated. Whereas access pattern such as temporal locality and sequentiality depend on work-

loads [83, 84] and therefore can change dynamically, data correlations are governed by data semantics [85], and are more static in nature. Even though data correlations have been exploited in prefetching technologies in storage system [86] and database engines [87], the performance of SSDs are still bottlenecked by FTL mapping table lookup and require significant research.

There are several key challenges that must be addressed in order to exploit data correlation in a page-level FTL both effectively and efficiently: 1) *Limited Resources* - Capabilities of the SSD hardware, such as the computing power of the flash controller and the size of the memory, are typically very limited. Since data correlations mining in SSDs needs to be done in the firmware, this needs to be implemented carefully so that it won't impact other essential tasks, such as address translation. To that end, this work adopts a regional correlation mining method to analyze workloads iteratively. We also designed a clustering-aware correlation mining algorithm, leveraging the clustering of correlations to reduce the overhead. 2) *Out of Page Prefetch* - there is no straightforward way to exploit the data correlation to reduce the number of page read/write operations. For example, we can use the data correlations to prefetch the mapping table or even the data. However, this cannot reduce the total number of page read or write operations on the SSDs. We can rearrange the mapping table based on the data correlations to reduce the number of page reads. However, in doing this, the mapping looks up mechanism will become more overwhelmed. 3) *Unordered Mapping Table* - in contrast to the mapping table in traditional page-level FTL, the correlated mapping table is unordered. To look up a correlated mapping entry is even more challenging. To this end, we designed the correlation prediction table to locate the page at which the correlated mapping entry is stored. 4) *Distinct Read/Write Operation* - write and read operations in SSD have distinct characteristics, which needs to be taken into consideration in the FTL design.

With the consideration of foregoing challenges, we make the following contributions in this chapter:

- As far as we know, this work is among the first to exploit data correlations within FTL. Most contemporary mapping table design in FTLs is either based on spatial or temporal locality. However, the real-world workloads for FTLs usually exhibit very low locality. This is due to a fact that majority of locality has been filtered by higher level main memory, file system buffer and storage cache, etc. By uniquely leveraging semantic links in workloads, the proposed Correlation-aware Page-level FTL (CPFTL) can significantly enhance page mapping efficiency, read and write performance and mitigate the garbage collection overhead.

- By analyzing data access logs and identifying correlated pages, the correlated mapping table, which aims to reduce the number of page fetch during address translation is developed. With the consideration of limited on-disk memory and computing resources, we proposed the correlation prediction table to support fast correlation table look up.

- To alleviate the garbage collection overhead, the skew-aware dirty entry index scheme is developed here, which performs locality aware dirty entry update to reduce the total number of write to flash.

- Using a kernel based emulator and a widely accepted simulator, we conduct comprehensive evaluation on the performance improvement and potential overhead of CPFTL. We evaluate CPFTL against various real world I/O traces. The results show that CPFTL can reduce the average response time by 63.4% for read dominant workloads, and 32.9% for transaction workloads.

Background & Motivation

Performance of SSDs

FTL is a key component in all standard SSDs, which manages the key function of virtual-to-physical address mapping. Performance wise, the I/O cost incurred by FTL is equal to, if not greater than, the cost incurred by other components such as flash. Typically, due to the overhead in address mapping, garbage collection, etc., applications can only utilize half of the maximum bandwidth provided by SSDs as reported in [80].

The I/O throughput of SSDs today under random workloads is two or three times lower than that under sequential workloads [44]. To further demonstrate the impact of I/O patterns on the SSD throughput and latency, we execute several performance tests on Flashsim [88] using the Financial 2 I/O traces [89]. Two state-of-art demand based page-level FTLs, namely DFTL [43] and SFTL [90], are configured into the Flashsim respectively. In particular, the first test, referred to as F2 DFTL, runs the trace with DFTL, which uses mapping entry as the basic cache management unit. The second test (a.k.a. F2 SFTL), is conducted under SFTL settings, which is a derivation of DFTL. In SFTL, the basic cache management is a group of mapping entries (i.e., a mapping page). The third test, referred to as Seq. DFTL, runs the manipulated sequential workload with DFTL. In this test, we convert the random workloads in to sequential workloads by change the requested addresses. The purpose of this test is to study the performance when the workloads are optimally sequential. The fourth test, referred to as Seq. SFTL, runs the manipulated sequential workload with SFTL. The fifth test, referred to as Optimal, runs DFTL with unlimited mapping cache. Figure 1 shows the performance and overhead analysis of DFTL and SFTL under different workloads. As shown in Figure 1a, F2 DFTL doubles the response time compared to the Optimal. Even under the favorable sequential workload, the response time of DFTL is slightly improved at a significant cost

of translation page read and translation block erase overhead (Figure 1b & Figure 1c). Despite the near-optimal performance of SFTL in all fours aspects under sequential workloads, SFTL cannot guarantee such performance in some real world workloads, such as the Financial 2.

Latency Analysis

In the previous subsection, we demonstrated that realist workloads can only achieve up to half of the maximum bandwidth offered by SSDs. Next, we provide a detailed analysis of the internal mechanics of SSDs to get a better understanding.

Additional Page Read Overhead in Address Translation

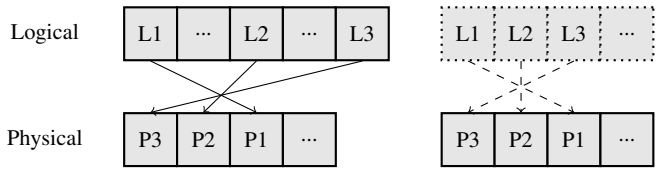
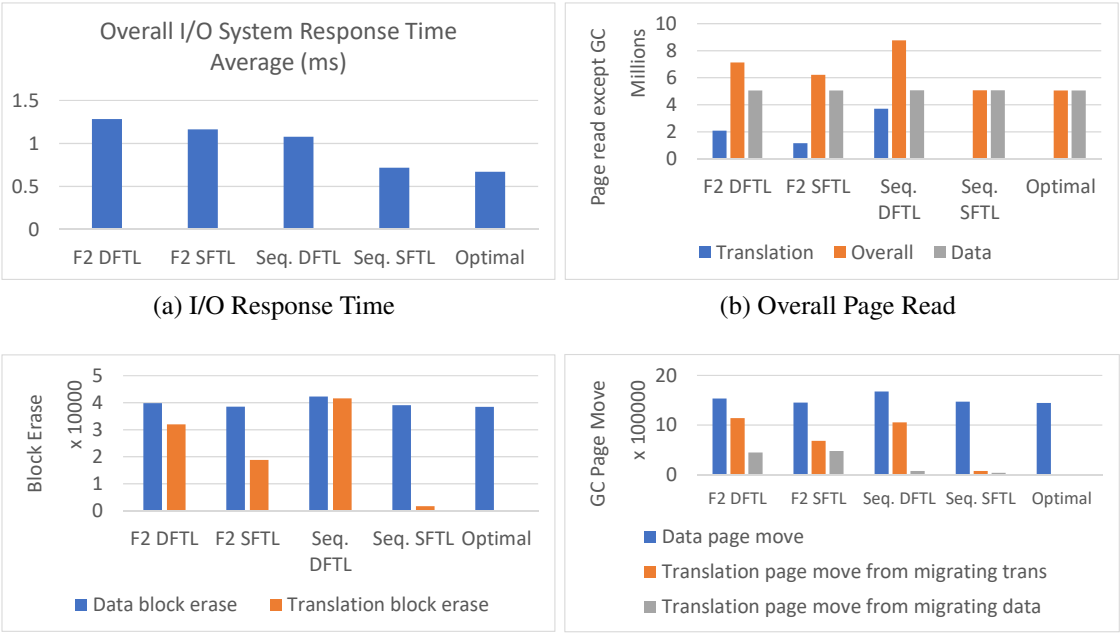


Figure 3.1: Mapping Management in SSD: Correlated Mapping Table is stored separately, causing additional read request to locate the correlated data.

One cause of low application I/O bandwidth in SSDs is the substantial overhead associated with address translation. Under the manipulated sequential workloads, we make the logical addresses always been sequential. As such FTL can take advantage of data locality by loading multiple mapping entries in a single mapping translation page read. In FTL, when two or more logical page number (LPN) are frequently accessed together, then there is *correlation* between these LPNs, and we define these LPNs as *correlated LPNs*. However, in real world applications, correlated LPNs can be dispersed onto non-sequential logical addresses. For example, in an e-commerce database system, purchase orders can be linked with user and item information, and most often

they are stored in separate files, resulting in non-adjacent logical address being accessed. To query information, multiple mapping pages may be loaded, and this results in higher read latency and higher contentions in the internal bandwidth of SSDs.



(a) I/O Response Time

(b) Overall Page Read

(c) Block Erase

(d) Page Move by GC

Figure 3.2: Performance Analysis under different FTL schemes.

Figure 3.1 represents an example of non-sequential workloads. As shown in the solid lined boxes, logical addresses $L1$, $L2$ and $L3$ are correlated and are non-sequential. If these addresses are accessed by applications, the FTL needs to launch 3 translation page reads to locate the data. Many methods have been introduced to improve the address translation performance. In DFTL, the LRU cache can accelerate the read performance of popular data [43]. However, the cache hit ratio is very low for the external I/O workloads, and the miss penalty is very high. One cache miss will double the latency of a read operation. What's more, the added translation page read will result in internal bandwidth contention, which further degrades the performance. Large chunk data access can benefit from sequential mapping prefetch [91] because the mapping entries are often naturally

stored in the same translation page. Figure 3.2b compares the number of translation page reads and the number of data page reads under different FTLs while running Financial Workloads. The results show that the page locality aware prefetch implemented in SFTL can reduce the number of translation page reads. However, SFTL cannot reduce the translation page read unless the locality of the mapping entries can be improved. This is because sequential prefetch cannot reduce the translation page read for non-sequential data. This motivated us to store correlated mapping entries in the same translation page.

Garbage Collection Overhead in Translation Blocks

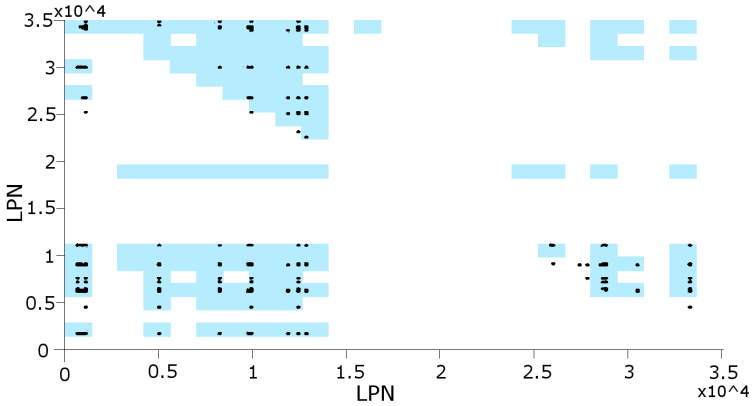
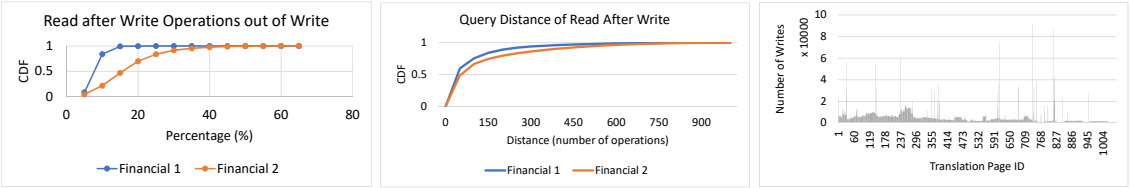


Figure 3.3: Correlation Distribution for Financial 2 workload.

Another major factor that influences the performance of real world applications is garbage collection (GC) overhead. To understand the GC behavior, we analyze the GC overhead under different FTLs. We first run the Financial workload, which consists of about 10^6 page writes. Then, we convert the workload to a sequential workload and rerun the experiments. As shown in Figure 3.2c, while we erase approximately the same amount of data blocks under different experiments, the number of translation block erases changes noticeably. In a typical page-level FTL, the number of data blocks is at least 1024 times larger than the number of translation blocks (1024 mapping

entry per translation page). However, the number of translation block erases is about 1/2 - 3/4 of the number of data block erases in the real application experiments (F2 DFTL and F2 SFTL). Figure 3.2d shows the number of translation pages moved by the GC. This number is slightly larger than the number of data pages moved by the GC in the real application experiments. We can also observe that the majority of the translation page movements are triggered by translation pages migrating themselves. This gives us evidence that the majority of the GC overhead is related to translation pages.



(a) CDF of Dirty Read (b) Distance of Dirty Read (c) Write Distribution

Figure 3.4: Workload Analysis.

We then analyze the root cause of such a large overhead in the Translation Block Garbage Collection as compared with the Data blocks. To efficiently serve the write requests, data are written in a log-structure by FTLs based on the request arrival time in the storage system. Multiple groups of correlated data can be generated concurrently; thus, different data can be mixed in the log stream. The SSDs perform journal writes based on the arrival time of the data. Consequently, blocks contain pages from various correlated content. Thus, upon an erase, live pages from other correlated content in the victim block will be moved to a new block during the GC process. This GC process will in turn mix multiple types of content in a single block. The erase of correlated content can cause unnecessary garbage collection contention. Because one mapping page corresponds to multiple data pages, the garbage collection contention is worse in the mapping blocks.

Addressing Performance Issue in FTL

To design an efficient correlation aware FTL, we consider the ways in which LPNs are correlated.

Improving Data Locality by Introducing Correlated Translation Blocks

Nand Flash based SSDs can perform batch mapping entry look up with a relatively low cost when the entries are stored on the same page. However, real world applications consist mostly of random I/O workloads. While block correlation is well studied in C-miner [82], none of the FTLs take the correlation into consideration. As we can see in Figure 3.3, each data point indicates there are frequently appeared I/O sequence first access the LPN on the x-axis then access the LPN on the y-axis. We can find that the correlation is not evenly distributed: some areas have more correlated LPNs than others. Two types of correlation clustering are expected to be observed. First, related content has implicit clustering on the timestamps. For example, in E-commercial workloads, the correlated order and shipping information are clustered on the timeline. Thus, the physical locations of these data are close to each other [92]. Second, storage software, such as filesystems and databases, trend to aggregate the metadata in a small region. Because of this, the correlated LPNs are often non-adjacent. In journal file systems, such as ext4, the metadata is mainly stored in the inode Table; thus, the correlations are mainly from the inode blocks to data blocks. Copy-on-write file systems, such as btrfs, trend to aggregate metadata into a larger inode block to reduce the number of random I/O requests. Furthermore, researchers propose more aggressive aggregation and batching of metadata updates [93].

Hours or days are often required to mine the frequency subsequence in a workload [82]. Running such a long correlation mining job on the CPUs inside SSDs will inevitably interfere with the performance of the SSDs. Based on the above observation, the correlations are aggregated in only

a few hot spots. To perform quick mining of the most useful correlations we first perform regional correlation mining to find these hot spots. Then, we filter the original I/O traces to only analyze the correlations within a specific region. The analyses of different correlation regions can be isolated; thus, we can divide the analyses into different steps, each step corresponds with a single region. In doing this, we can reduce the interference with the outside workloads.

Improving Cache Utilization by Exploiting Read and Write Disparity

We define the condition in which a write operation has a pending read operation on the same address in a short time frame as *Read after Write Operation*. We define the pending read operation of the dirty entry as a *Dirty Read*. To analyze the dirty reads in workloads, we partitioned the I/O traces, with each partition containing 1024 traces. We then calculated the *dirty-read/write ratios* for each partition by divide the number of write operations by the number of read after write operations. Figure 3.4a shows that the average dirty-read/write ratios of different workloads are very low compared to the LRU Read cache. The ratio of dirty-read/write for Financial traces range from 8% to 17%, and the ratio for WebSearch traces are 0%. The figure also indicates that this is a very stable parameter throughout the execution of the application. Figure 3.4b shows the distance from the write operation to the correlated dirty-read in the I/O queue. 99% of the distances are lower than 500 traces. This means that the dirty page will not likely be accessed if it has not been accessed during the first 500 traces. Based on these observations, we can selectively evict more dirty caches to allow the read caches to stay in place for a longer time.

Reducing GC Overhead by Leveraging Write Skewness

Researchers have long observed write skewness in I/O workloads. A disk-level trace of personal workstations at Hewlett Packard laboratories exhibits a high locality of references in that 90% of

the writes go to 1% of the blocks [94]. Roselli et al. [95] analyzed file system traces collected from four different groups of machines: an instructional laboratory, a set of computers used for research, a single web server, and a set of PCs running Windows NT. They found that files tend to be either read-mostly or write-mostly and the writes show substantial locality. Lee and Moon [96] showed that the update frequency of TPC-C workloads is highly skewed, in that 29% writes go to 1.6% of pages. Figure 3.4c shows the distribution of the write operations. The results show that the write operations are highly skewed to a small number of pages. This is because the upper level file system is intended to aggregate the metadata in a small region, and these small regions have higher update frequencies as compared with the regions that store the data. For example, journaling file systems, such as ext4 file systems, place an inode table and data blocks on separate LBAs. Log-structured file systems also place the frequently updated blocks in the same segmentation [97].

Block Padding Least Recently Used (BPLRU) [98] improves the random write performance by introducing a block aware LRU cache management algorithm in the FTL, which updates the LRU list with consideration of the size of the erasable blocks in order to minimize the number of merge operations. TPFTL [91], which performs a batch update for every dirty entry in a single translation page, has proven to reduce the total number of translation page writes. However, these algorithms ignore the fact that write skewness in the cache management can reduce the chance to absorb more entry updates in the frequently updated pages. The write skewness in translation blocks is more severe than data blocks. Because one translation page points to multiple data pages, updating the corresponding data pages will require that the translation page be updated. As a result, the update frequency of translation blocks is much higher. We propose to keep the dirty entries in frequently updated translation pages longer and index the dirty entries by the page id to perform co-located dirty page updates.

Another problem regarding the write skewness is that pages with different update frequencies can be mixed in an erase block, resulting in unnecessary translation page movement during garbage

collection. Stoica [99] used over provisioning to leverage the write skewness. However, over provisioning cannot prevent the mixture of pages with different update frequencies. Instead, we propose to isolate translation pages with different update frequencies to minimize the garbage collection contention.

The Design of CPFTL

In this section, we show the design details of CPFTL. First, we designed an efficient on-line correlation exploit algorithm. We then store the correlation in an efficient data structure to reduce the access latency of non-sequential correlated I/Os. Second, we developed a dirty entry index in the mapping cache to improve the cache utilization and enable collocated batch updates of dirty entries. Finally, we studied the skewed cache management based on the write skewness of the workloads.

Overview of CPFTL

The design of CPFTL aims to reach the following three critical objectives.

- *Correlation-Aware Prefetch* - By prefetching the correlated mapping entries of the incoming read request, we can improve the cache hit ratio, thus effectively reducing the read latency.
- *Reduce the Number of Translation Page Reads* - By grouping and storing the correlated mapping entries in a single mapping page, we can load these mapping entries in one page read, thus we can reduce the internal bandwidth contention.
- *Reduce the Garbage Collection Contention* - By developing a Skew-Aware Dirty Entry Index, we can perform batch updates under the same mapping page and enable early detection

of Dirty Entries with low hit ratios.

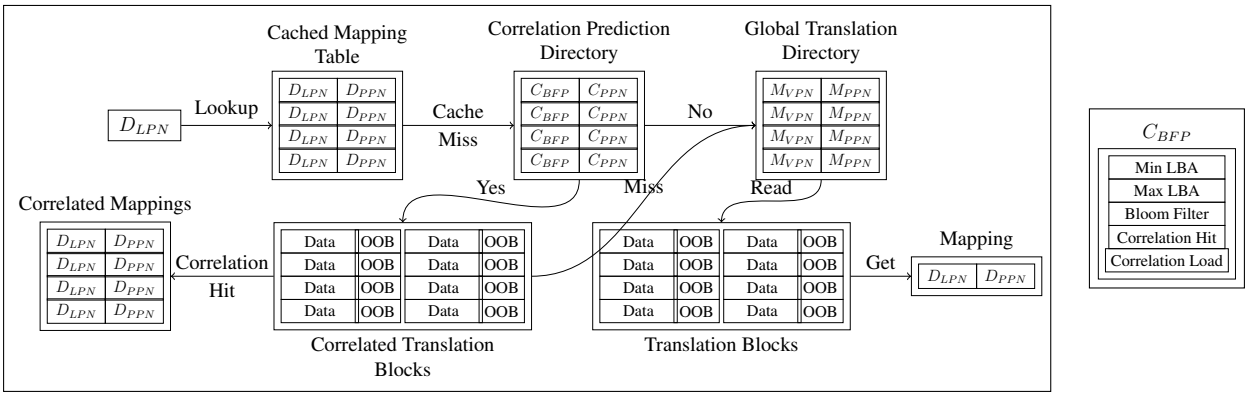


Figure 3.5: Architecture of CPFTL. D_{LPN}: Logical Data Page Number, D_{PPN}: Physical Data Page Number, C_{BFP}: Correlation Bloom Filter Predictor, C_{PPN}: Physical Correlated Translation Page Number, M_{VPN}: Virtual Translation Page Number, M_{PPN}: Physical Translation Page Number.

To achieve these objectives, we propose Correlation-Aware Page-Level FTL, aka CPFTL. CPFTL mines the correlated mapping entries and stores them into one page called the Correlated Translation Page. In doing this, we can get the correlated mapping entries in a single page read operation, thus reducing the total number of page reads and improving the I/O performance. As shown in Figure 3.5, the flash memory is divided into three parts: 1) the data blocks store user data, 2) the page-level mapping blocks store the mapping entries, which are ordered by the LPNs and 3) the correlated mapping blocks store the correlated mapping entries. To translate the LPNs, we first look up the mapping cache. The uncached LPN will then be send to correlation prediction directory. If there is no correlation prediction hit, the regular page table look up processed will be issued. Otherwise, we load the correlated translation page and cache all the correlated mappings. If we do not find any correlated mappings, we fallback to the regular page look up process.

Correlated Translation Page

To improve the "in-page" locality of the correlated mapping entries, we designed a correlated mapping table. As opposed to the conventional page-level mapping table, in which the mapping entry is ordered by LPNs, the correlated mapping table stores the correlated mapping entries together regardless of the LPN. In doing this, when an LPN is accessed, we can prefetch all of the correlated mapping entries to the mapping cache. There are several challenges in designing the correlated mapping table. First, the CPU and memory resources in SSDs are very limited, we need to carefully develop a low overhead mining algorithm. Second, because the correlated mapping table is unordered, the overhead to update and search the correlated mapping table is high. To solve these challenges, we first design a clustering aware correlation mining algorithm to quickly mine the most used sequences. We then design a correlation prediction directory to support fast location of the mapping entries. The main service loop is in Algorithm 1.

Clustering Aware Correlation Mining

One of the major challenge of implementing the Correlation-Aware Page-Level FTL is how can we mine the correlations with limited hardware resources. As discussed in Section 3, majority of the correlations are clustered. Based on this observation, we designed Clustering Aware Correlation Mining (CACM). We split the LBA addresses into fix-sized blocks, aka regions. We use regions and blocks interchangeably in this paper. We then collect the read traces and convert them to regional access sequences. We periodically perform a fast regional correlation lookup to find the most correlated region. In Figure 3.3, 92.9% of the correlations are covered in the regional correlation area (gray area). For example, if LBA a in region A and LBA b in region B are accessed. We record the regional access sequence $\{A, B\}$. If $\{A, B\}$ frequently appears as a regional access sequence, then we label $\{A, B\}$ as a frequently correlated region.

Algorithm 1: Mapping table look up Loop in Correlation Aware FTL

```
Data: Logical Page Number
Result: Physical Page Number
initialization;
while new I/O request do
    if read request then
        if cache hit then
            return PPN;
        else
            while LPN hit in C_BFP do
                load Correlated Translation Page;
                if LPN exist in Correlated Translation Page then
                    cache Correlated Mapping Entries;
                    return PPN;
                end
            end
        end
    end
    else if write request then
        while LPN hit in C_BFP do
            load Correlated Translation Page;
            if LPN exist in Correlated Translation Page then
                remove Correlated Translation Page;
            end
        end
        return new PPN;
    end
end
```

Because the correlation mining time and memory usage is exponentially related to the problem size, we focus on the correlation mining algorithm in the most frequently correlated regions to keep the problem size small. We use a fixed sized non-overlapping cutting window to extract all the 2-fold subsequences from region A and B. For example, if $\{A, B\}$ is a frequently correlated region in the LBA sequence $\{a, b, c\}$, then only $\{a, b\}$ will be extracted and $\{a, c\}$ and $\{b, c\}$ will be disguarded. We first calculate the frequencies, or *support*, of all extracted subsequences. We then store the subsequences with a *support* larger than a predefined threshold T in a compressed

manner. For example, if we get 3 subsequences $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$, we store them in Flash as $\{a, bc\}$ and $\{b, c\}$. The $\{a\}$ and $\{b\}$ in the subsequence are the keys used to identify the sequence. If the FTL receives a request for data $\{a\}$ and encounters a cache miss, we load the sequence $\{a, bc\}$ into memory.

Correlation Prediction

Because the mapping entries in the Correlated Translation Blocks are unordered, the entry search in the unordered mapping table is extremely slow. To solve this problem, we designed a light weight correlation prediction directory. The entries in the Correlation Prediction Directory consist of a Correlation Bloom Filter Predictor (C_{BFP}) and a physical page number of the Correlated Translation Page (C_{PPN}). The C_{BFP} consists of the largest and smallest LPNs in the C_{PPN} as well as the bloom filter constructed by all of the LPNs in the C_{PPN}. When the FTL needs to locate a D_{LPN}, it first checks the C_{BFP} to determine if the D_{LPN} is in the bloom filter. If the D_{LPN} is in the bloom filter, then the Correlated Translation Page is loaded to search the D_{LPN}. If the D_{LPN} is found, then the correlated mappings will be prefetched. Suppose the number of the correlated LPNs is n, the percentage of correlated LPNs in the I/O sequence is x, and the false positive of bloom filter is y. The number of page read can be calculated from in Equation 3.1

$$x + (1 - x)((n + 1)y + n(1 - y)) \tag{3.1}$$

Thus, we can reduce the total number of page reads if y meets the condition in Equation 3.2

$$y < (n - 1)x/(1 - x) \tag{3.2}$$

The element in the bloom filter cannot be removed. However, the correlation inside the correlation translation page could be changed and some correlations will need to be removed. There are several alternative approximative data structures, such as Cuckoo-filter [100] and bloom filter with removal support [101]. These data structure can support the removal of the obsolete correlations with in a page, but they will require additional engineering work. Fortunately, in the experiments, we find that the I/O correlation does not change significantly in the entire trace. Thus, simply removing the entire correlation translation page and its corresponding bloom filter is sufficient. In the following section, we will discuss how we use adaptive prefetching to deal with the obsolete correlations.

Adaptive Prefetching

As the application is running, the prior correlated pages may no longer be correlated because of the changes in the data or the application behavior. Thus, the obsolete correlations must be removed. To solve this problem, we developed two algorithms. First, we remove the Correlated Translation Page when new data is written to one of the correlated pages. Second, we detect the quality of the correlated translation pages by collecting and calculating their confidential ratio. The C_{BFP} has a Correlation Hit and Load counter. Every time we read the correlated mapping page, the Load counter is incremented by 1. If the correlated mapping preloaded to the cache is accessed, then the Hit Counter is incremented by 1. If the Hit/Load ratio is lower than a predefined threshold, then the Correlated Translation will be removed from the dictionary. The threshold can be dynamically calculated from the hit/load ratio of the regular cache. e.g. the LRU cache. The adaptive prefetching works only as an online method to remove the obsolete or low "quality" correlations. However, to reconstruct the correlated mapping table a new correlation mining procedure need to be performed.

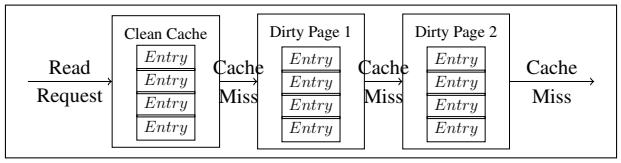


Figure 3.6: Dirty Entry Index.

As discussed above, the percentage of dirty entries in the cache that are accessed is very low. Furthermore, the write frequencies of different mapping translation pages are highly skewed. Unfortunately, this information cannot be easily utilized in current cache management because dirty entries and clean entries are mixed. To solve this problem, we designed a dirty entry index in the Mapping Cache. In this solution, the clean cache and dirty cache are managed separately. There are several benefits to managing a separate dirty page index in the mapping cache. First, the contention between clean and dirty caches can be reduced. In traditional cache management, the clean and dirty caches are mixed in a single data structure with a flag to distinguish them. However, the probability of a dirty page being read is very low. Thus, leaving dirty pages in the cache for a long time increases the chance that a clean entry will be accessed in the future. Second, it is easier to utilize the skewed update frequency of mapping translation pages in the dirty page eviction strategy.

To capture the write skewness, we build an update frequency table for the translation pages based on historical traces. Note that in order to minimize the interference to the online workloads, all of the data mining including correlation mining and update frequency mining are done in the background. We assume that the mining tasks are scheduled when the system is idle, or the system load is low. We then select an update frequency threshold U for the frequently updated translation pages. In the cache management, we manage the dirty entries in a way that the entries in one

translation page are grouped as a dirty page node in the cache. While we evict a dirty page, we first check if it is a frequently updated translation page. If the evicted dirty page is a frequently updated translation page, we then check the number of dirty entries in this dirty page. If the number of dirty entries is smaller than a predefined threshold, we prevent this dirty page from being evicted.

Evaluation

Evaluation Setup

In order to evaluate the performance of Correlation-Aware Page-level FTL, we developed a simulator under the Flashsim platform [102]. We compared CPFTL with DFTL [43], SFTL [90], TPFTL [91] and the optimal FTL. The main feature of these FTLs are listed as follows:

1. **DFTL** [43] is a very basic Page-level FTL with a mapping entry level LRU cache. We compare CPFTL against DFTL to demonstrate the performance gains relative to the baseline FTL.

2. **SFTL** [90] is a Page-level FTL with a mapping page level LUR cache that is implemented to leverage spatial locality. The best case scenario for SFTL is if workloads repeatedly access mapping entries within the same mapping page. We compare CPFTL with SFTL to demonstrate the performance gains under workloads in which spatial locality is very low.

3. **TPFTL** [91] is a Page-level FTL that performs batch read and write operations in a single translation page to exploit both temporal and spatial locality. We compare the performance enhancement provided by CPFTL with TPFTL to demonstrate the benefits of correlation mining and skew aware dirty entry index.

4. **The Optimal FTL** is a Page-level FTL that stores all mapping entries in DRAM space. We use this configuration to indicate the upper bound of the performance.

Workload Traces

In the evaluation, we choose three enterprise traces to study the efficiency of different FTLs. The Financial trace is a set of random-dominant workloads that was collected from an OLTP application running at a large finical institution. WebSearch was collected from a popular search engine. Its request sizes are larger than those of the financial trace and they have a very low temporal locality. TPCC is an on-line transaction processing benchmark, which has a larger portion of write and more intensive request than Financial workloads. Table 3.1 presents the features of the workloads.

Table 3.1: Enterprise-Scale Workload Characteristics.

Workloads	Avg.Req.Size read/write(KB)	Read (%)	Avg.Req.Arrv. Time (ms)
Financial	2.3/2.9	82.3	11.08
Websearch	15.15/8.6	99.9	2.99
TPCC	8/8.2	65.7	4.35

Table 3.2: Preprocessed Workload Characteristics.

Workloads	Avg.Req.Size read/write(KB)	Read (%)	Avg.Req.Arrv. Time (ms)
Financial	2.1/2.4	65.6	9.08
Websearch	17.5/9.8	99.9	1.33
TPCC	8/8	57.9	3.7

The majority of computer systems usually have their own data buffer on the host side. The temporal locality can be very effectively utilized by the host software. As a result, the workloads inside the disk drivers have a much lower locality. In this paper we mainly focus on the workloads inside of disk drives, which unfortunately do not have much temporal and spatial locality to utilize. To simulate the host side buffer, we use a host side LRU cache to pre-process the traces and recollect the I/O traces that are not hit in the host side buffer. The preprocessed workload characteristics are listed in Table 3.2. For the evaluation, the host side buffer is set to 5% of the total storage size. The evaluation results show that correlation aware design is a better fit for FTL, which mainly works under an environment with low locality.

Evaluation Methodology

We run our selected application traces by using a trace-driven simulation method on the simulation platform Flashsim [102]. The SSD parameters in our simulation are taken from [91] and listed in Table 3.3. To serve the workloads, we set the SSD capacity as large as the logical address space of the traces. The mapping cache is set as large as the mapping table of the block-level FTL plus the Global Translation Diction (GTD) and Correlation Prediction Dictionary (CPD) size, which is in proportion to the SSD capacity. Specifically, the capacity of the simulated SSD is 16GB and the mapping cache size is 288KB (256KB + 16KB GTD + 16KB CPD). All the experiments are done by using minimal amount of memory necessary for implementing the FTL. The extra DRAM space can be used for data buffer or correlation mining. We evaluate each FTL by recording the average response times of the I/O operations under the I/O traces collected from enterprise-scale applications. We also perform detailed analyses on the internal cache hit ratio, the number of page reads/writes, and the block erases in the SSDs.

To validate the effectiveness of Correlation Aware FTL, we use learning and testing methods. We

divide the workload traces into multiple training and testing pairs. We mine the correlations by using training traces and test the correlations by using testing traces. Because different workloads have different patterns, we repeatedly do training and testing under different configurations to get the best results. The configuration details are discussed in the latter part of the results.

Table 3.3: SSD parameters in our simulation

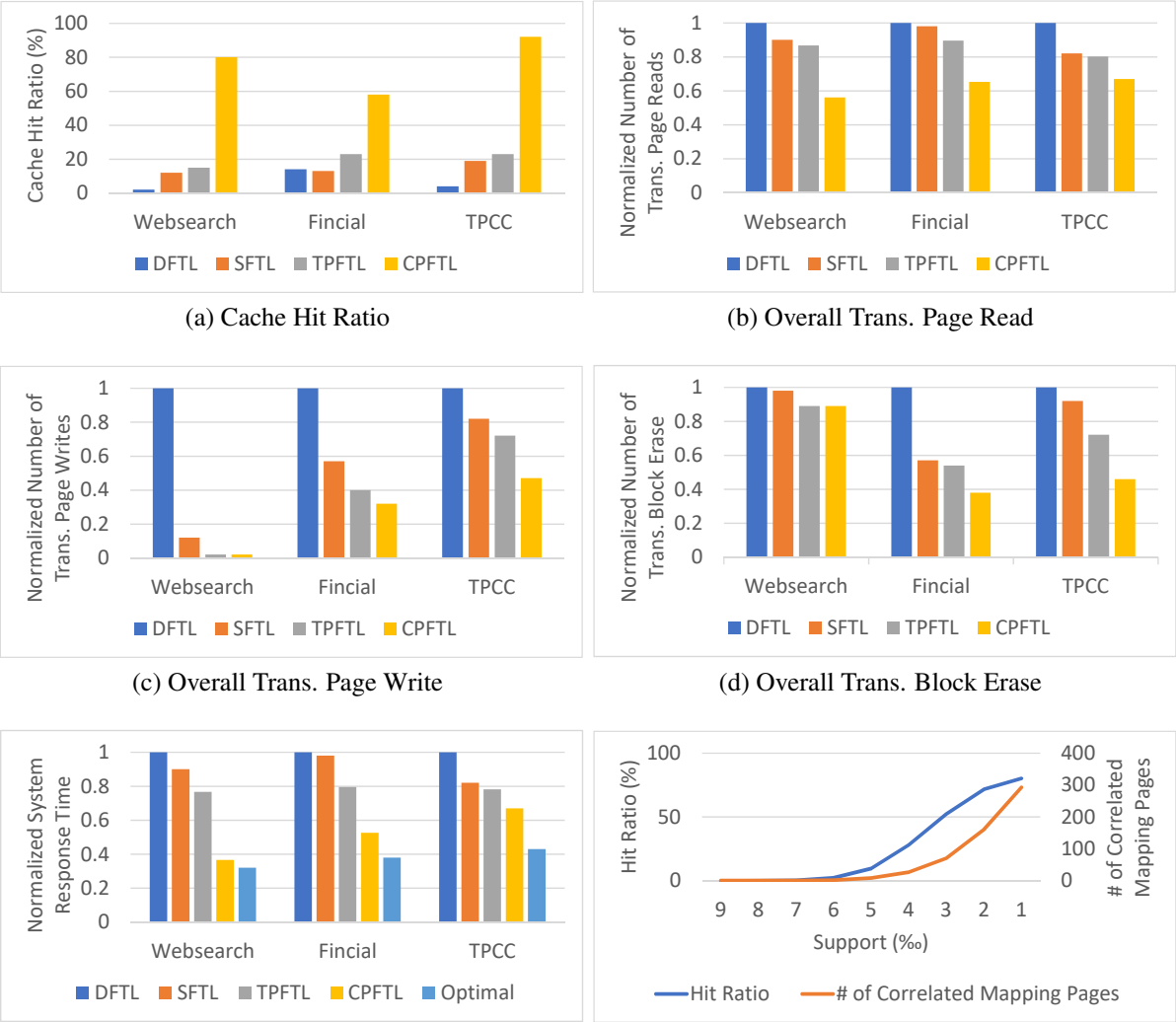
Flash Page Size	4KB
Flash Block Size	256KB
Page Read Latency	25us
Page Write Latency	200us
Block Erase Latency	1.5ms
Over-provision Space	15%

Experiment Results

Cache Hit Ratio

Figure 3.7a shows the cache hit ratios, including both reads and writes, of different FTLs under the three workloads. For the Financial workloads, SFTL slightly reduces the cache hit ratio compared with DFTL because of the weak spatial locality. TPFTL improves the hit ratios by an average of 11.5% compared with DFTL by leveraging the sequential workloads in the trace. CPFTL improves the hit ratios by 58.3% compared with DFTL by leveraging the access correlation. For the Web-Search workloads, TPFTL and SFTL achieve a 15.2% and 12.1% hit ratio respectively because the relatively larger sized requests have better spatial locality. CPFTL achieves higher hit ratios than TPFTL and DFTL by an average of 37%. We can conclude that CPFTL succeeds in maintaining a

relatively high hit ratio in various workloads. This is because of the correlation aware prefetching and the workload adaptive dirty page eviction policy in CPFTL. The prefetching batch loads the correlated mapping pages under low the temporal locality. The latter more aggressively removes the dirty mapping in the cache. By contrast, DFTL only exploits the temporal locality, while S-FTL and TPFTL radically exploit the spatial locality.



(a) Cache Hit Ratio

(b) Overall Trans. Page Read

(c) Overall Trans. Page Write

(d) Overall Trans. Block Erase

(e) System Response Time

(f) Impact of Correlation Mining

Figure 3.7: Performance of CPFTL

Numbers of Translation Page Reads

Figure 3.7b shows the normalized numbers of translation page reads, during the address translation phase and GC operations of different FTLs under the three workloads. Each value is normalized to that of DFTL and a value of 1 means they are equal. Note that a higher hit ratio leads to fewer translation page reads required by mapping entry loadings. Also, reduced GC overhead leads to fewer translation page move required by GC processes. By placing the correlated mapping entry in one flash page, CPFTL reduces the numbers of translation page reads by 57.5% for the Financial workloads, 49.3% for the WebSearch workloads, and 44% for TPCC workloads compared to DFTL. The reason why the reductions of translation page reads of CPFTL for the Financial and TPCC workloads is less than those for WebSearch is because the Financial and TPCC workloads have more random writes, so the GC overhead is larger.

Numbers of Translation Page Writes

Figure 3.7c shows the normalized numbers of translation page writes during both the address translation update and the GC operations of different FTLs under the three workloads. Each value is normalized to that of DFTL and a value of 1 means that they are equal. Compared to DFTL, CPFTL reduces the number of translation page writes by an average of 99.8%, 50.5%, and 31.4% for the WebSearch, Financial and TPCC workloads respectively. The WebSearch workloads have only a few write operations and majority of the writes are sequential, so the reduction of translation page writes is outstanding compared with DFTL, which uses a per dirty entry update scheme. The reason why CPFTL can reduce the number of page writes is because the replacement unit of SFTL is a full page. Because of this, it eliminates the reads required for writing back dirty entries. The reason why CPFTL can reduce more translation page writes than TPFTL is because the skew aware dirty entry management can reduce the dirty entry updating and GC overhead.

Numbers of Translation Block Erase

Figure 3.7d shows the normalized numbers of translation block erases during the GC process. For WebSearch workloads, only a few translation page updates are incurred during address translation, and almost no valid pages are migrated during GC operations because most writes are sequential. Thus, only a few block erases have been introduced and the reduction of block erases in CPFTL is low. Accordingly, compared with DFTL, the translation block erase counts of CPFTL decreased by an average of 58.5% and 41.8% for the Financial and TPCC workloads respectively. As compared with TPFTL, which performs batch updates of the dirty entries, CPFTL decreases the number of block erases by 16% and 25.9% for the Financial and TPCC workloads respectively. The reason for this reduction is that the skew aware dirty entry index can further reduce the skewness of the translation page update frequency. The reduction of block erases for the Financial workloads is smaller than for the TPCC workloads. This is because the Financial workloads have fewer writes than TPCC workloads and the Page-level LRU replacement policy in TPFTL can keep some of the frequently updated mapping entries in the cache.

Average Response Time

Figure 3.7e shows the normalized average system response times of different FTLs under the three workloads. Each value is normalized to the average system response time of DFTL. For read dominated WebSearch workloads, CPFTL can provide close to an optimal performance. This is because of the high cache hit ratio and the low number of page writes. For the Finical and TPCC workloads, although CPFTL improved the Cache Hit Ratio and reduced the number of translation page reads and writes to flash memory, the reduction of response time is not as dramatic. This is because GC operations on data blocks account for a considerable proportion of the response time and CPFTL uses a GC mechanism that is similar to the other FTLs mentioned in this paper. Since

CPFTL primarily targets low temporal and spatial locality workloads, it is not surprising to see that CPFTL achieves the greatest advantages in these types of workloads. CPFTL was able to reduce response time by 63.4%, 47.5%, and 32.9%, under the WebSearch, Financial and TPCC workloads respectively.

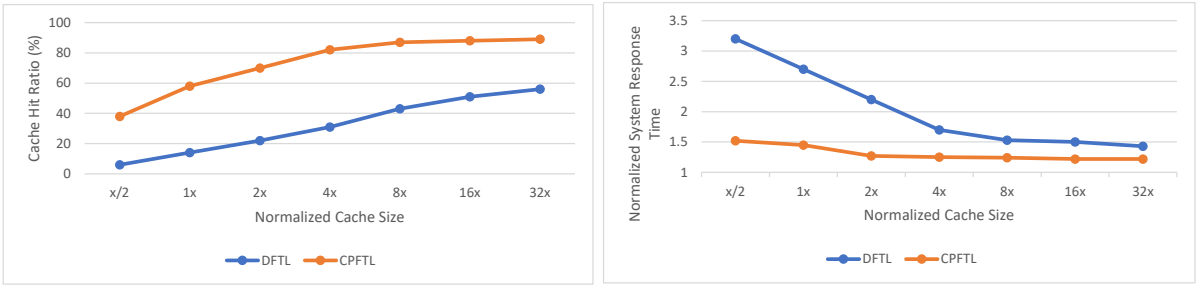
Impact of Correlation Mining

Figure 3.7f shows the impact of Correlation Mining on the Cache Hit Ratio for the WebSearch workloads. The WebSearch workloads are read dominated, so the cache hit ratio in address translation is one of the key factors that impact performance. As we expected, when the I/O traces are filtered by the host buffer, the temporal locality at the device level is insufficient. We change different *s*upport thresholds in the correlation mining. A large support threshold implies that only the access sequences with very high frequencies are used as correlation. An extremely large *s*upport threshold means no correlation are used. Because the workloads in SSDs have very low locality, we use a relatively low *s*upport threshold as compared with what C-minter uses. As shown in the figure, the cache hit ratio is only 1.6% when we use 9‰ support. This is because we do not mine enough correlations in this configuration. While we reduce the support threshold to 1‰, we can increase the hit ratio to as high as 80.2%, which is very good for the last level of cache. The other impact we need to notice is that the number of correlations mined increases exponentially. The number of correlated mapping pages increases from 1 to 293, which indicates an additional cost to mine and search in these correlation tables.

Impact of Cache Size

In this subsection, we see the impact of the cache size on CPFTL. Figure 3.8 shows the cache hit ratio and system response time for Financial workloads. Each cache size is normalized to a

multiple of the standard size, denoted by 1x. For example, x/2, 1x and 2x stand for cache sizes of 128KB, 256KB and 512KB for a SSD with 16GB capacity, respectively. The system response time is normalized to the response time when all the mappings are cached. We can see that the hit ratio of CPFTL increases with increasing cache size and reach 89% when using 8x cache size. The response time of CPFTL reduces with increasing cache size and reach 1.22 when using 8x cache size. In all settings, CPFTL consistently has higher hit ratio and lower response time than DFTL.



(a) Cache Hit Ratio

(b) System Response Time

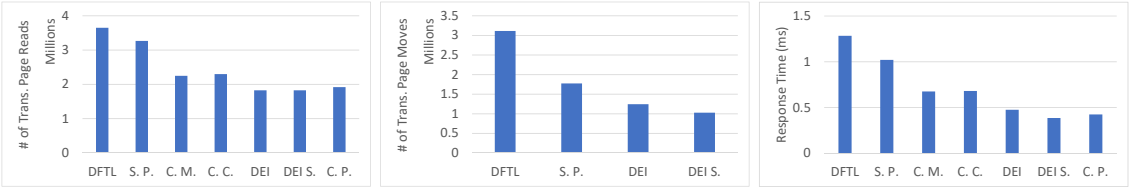
Figure 3.8: Impact of Cache Size.

Impact of Design Components

To give further insight into the techniques employed by CPFTL, we take the Financial workload as an example to investigate the benefits of each technique. Specifically, we evaluate six typical CPFTL configurations with and without the techniques. Each configuration is denoted by a monogram of enabled techniques, where S. P. means sequential prefetch and batch dirty entry updating are enabled; C. M. means correlation mining is enabled; C. C. means clustering aware correlation mining is enabled; DEI means dirty entry indexing is enabled; DEI S. means skew aware dirty entry index is enabled. Note that for the C. M. and C. C. configurations, we use a full search instead of our predictive data structure to locate the correlated mapping entries. We only count a page read

when there is a correlation hit. This consumes many CPU cycles, but our purpose is to determine the upper bounds of the correlated mapping table.

Figure 3.9a shows the numbers of translation page reads during both the address translation phase and GC operations with different configurations under the three workloads. The figure shows that correlation aware prefetch contributes most to the reduction of translation page reads. Clustering aware correlation mining has a comparable performance to the original mining algorithm, indicating that we can obtain the majority of the correlations by using clustering information. The dirty entry index also reduced the number of page reads by reducing the translation page updating frequency. However, the skew aware dirty entry index does not noticeably change page reads. While we enabled predictive correlation prefetch, we increased the page read count by 2.6%. This is because errors in the estimations of correlation prefetch could lead to unnecessary page reads in the correlated translation pages.



(a) Overall Trans. Page Reads (b) Trans. Page Move by GC (c) System Response Time

Figure 3.9: Impact of Design Components.

Figure 3.9b shows the numbers of translation page moves during GC operations with different configurations under the three workloads. In this experiment, we study the garbage overhead reduction of dirty entry index. Because the correlated mapping table does not contribute to the reduction of the page moves triggered by GC processes, we do not compare it in this experiment but study it's overhead in the next subsection. As expected, DEI reduced page moves by 17% as compared with S. P. and DEI S. further reduced the translation page moves by 7%. This is because

the dirty entry index can provide "in-page" locality aware mapping updates and the skew aware design can prevent frequently updated regions from being evicted too early by LRU algorithms.

Figure 3.9c shows the system response time under the three workloads. The figure shows that correlation aware prefetch is the greatest contributor to performance improvements. The dirty entry index reduced access latency more than translation page reads. This is because garbage collection has a greater impact on the SSD's performance than translation page reads. While we enabled predictive correlation prefetch, we slightly increased the page read counts by 2.6%. This is because there is some error estimation lead to the unnecessary page read in the correlated translation pages.

CPFTL Overhead

The additional overhead of CPFTL as compared with other FTLs mainly consist of two parts: the overhead of correlation mining and the overhead of updating the correlated mapping table. While the CPU time required to mine the correlations is reasonably small and the correlations under a given workload are very stable, the CPU usage is not a big concern [82]. However, the maximum memory consumption is the major bottleneck to port correlation mining to an SSD firmware. Table 3.4 shows the peak memory usage while mining the correlations for different workloads. While using C-miner [82], the peak memory used for mining the WebSearch, Financial and TPCC are 8.7GB, 4.9GB, and 8.2GB respectively. The memory consumption is simply too large to be able to port the C-miner to the SSD firmware directly. The reason for such a large memory consumption is that in C-miner, many subsequences are constructed from a trace in the correlation mining. The clustering aware correlation mining divides the traces into multiple correlation regions and mines one region in each iteration. It is not surprising to see that clustering aware correlation mining consumes less memory. The peak memory used for mining the WebSearch, Finical and TPCC

traces are 78MB, 132MB and 97MB respectively. Because our implementation for correlation is a JVM-based prototype, it is expected that we can further reduce the overhead while porting to SSD firmware.

The overhead related to updating the correlation table has two parts. First, the correlation table must be periodically rebuilt. Because the data correlations are very stable and the overhead to rebuild the correlation table is very small, this overhead is not a major concern. The other overhead is related to updating the correlation table while new data are written to the correlated pages or correlated data are moved by GC processes. To observe this, we calculate the number of update operations in the correlated mapping pages associated with write and GC operations. In Table 3.4, we can see that for the WebSearch workloads, none of the correlations are updated. This is because only a few write and GC operations are introduced. For the Financial and TPCC workloads, 7.6% and 15.3% of the correlations are updated. More correlations are updated in the TPCC workloads than in the Financial workloads because the TPCC workloads have more write operations and more GC operations.

Table 3.4: Overhead of CPFTL: Peak Memory Usage and Percentage of Correlation Updates

Workloads	C. M.	C. C.	Correlation Table Updates (%)
Websearch	8.7GB	78MB	0.0
Financial	4.9GB	62MB	7.6
TPCC	8.2GB	97MB	15.3

Emulation of CPFTL

To evaluate the on-line performance of CPFTL, we developed a SSD Emulator in Linux kernel and deployed experiments under different file systems. We implement the DFTL and CPFTL for comparison. In the DFTL, a basic LRU cache is enabled for the mapping table. In the CPFTL, a page-level correlation mining algorithm launched periodically in the background. We then prefetch the correlated mapping entries. The emulator is open sourced at: https://github.com/janzhou/SSD-Emulator.

Design of Emulator

Figure 3.10 shows the architecture of the SSD emulator design. At its top level, the design is divided into two parts: part of the emulator works at the kernel-space, while the other works at the user-space. The coordinated working of these two makes up the emulator. The purpose of this division is mainly to segregate the tasks to the points where it can be performed well. The kernel-space gives the advantage of dealing with the LBA requests directly, which can be further used to perform the I/O. Moreover, the kernel has the ability to reserve large amounts of contiguous memory at boot time, which would further be used as the emulated flash memory. However, the library support present inside the kernel space is very limited. Modern FTLs uses complex algorithms, such as data mining, in order to perform the duties. For example, the in-house developed correlation-aware FTL uses bloom filter algorithm in order to find the correlations among the LBA requests and define the mappings accordingly. The development and test cycle time might increase if we start implementing the algorithms in the kernel space. Moreover, debugging the kernel code has always been a rather difficult task than the user space code. On the other hand, user-space enjoys the luxury of various libraries provided through numerous programming languages interfaces such as C, C++, Java, Python, or Scala. As a result, it would be effective to move the FTL part of

the emulator to the user-space.

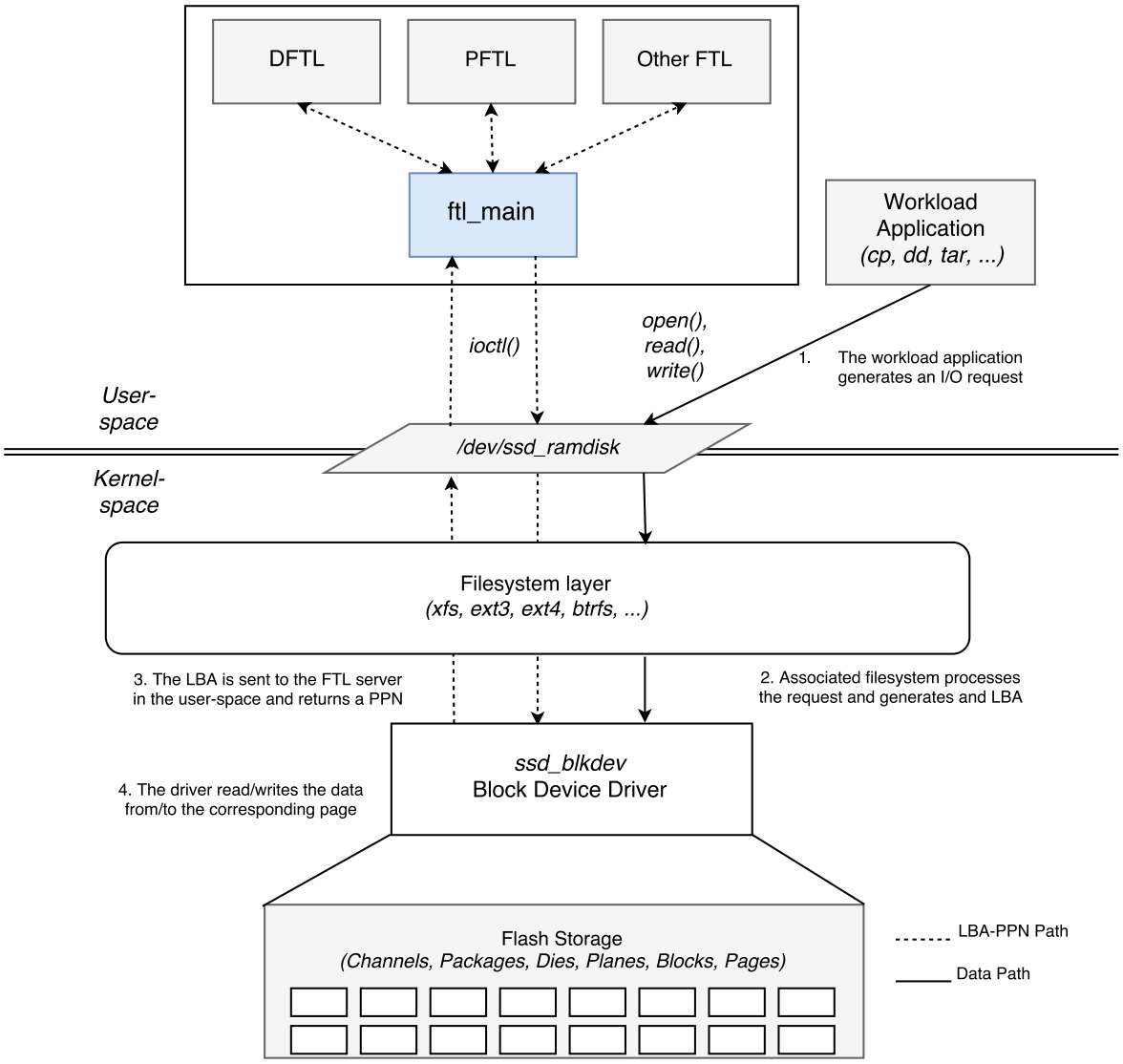


Figure 3.10: Architecture of SSD emulator

Kernel Level Block Device Driver

A block device driver, inside the Linux Kernel, serves as an interface between the filesystem layer and its corresponding hardware. It basically intercepts all of the filesystem requests, addressed through LBAs, and forwards them to the underlying SSD. The block driver emulates the SSD storage space by performing I/O onto the pre-allocated main memory. Although, as the read/write latencies would be different for SSDs and DRAM based main memories, explicit delays were introduced to perform the I/O. To allocate large memory and avoid memory swap in Linux Kernel, we have hacked the code of the persistent memory (*pmem*) emulator module of the Linux Kernel. The *pmem* driver offers the flexibility to allocate the desired amount of physical memory at boot time. The memory allocated would be further used by the block device driver in order to logically partition it into dies, planes, blocks and pages.

User-space FTL Server

Traditionally, FTLs are simple firmware code which runs on the SSD whose main objective was to provide a mapping layer between the LBAs and the Physical Page Numbers (PPNs). However, its responsibilities have grown over time. FTL provides other features such as Garbage Collection, Cache management, data relocation, ECC, et. al. As a result, even the computation requirement grew and hence being one of the reasons for moving the job of FTL onto the host. As mentioned earlier, the FTL implementation is moved to the user-space due to the comforts we gain by the user-space libraries provided by various programming languages. The current FTL implementations are done in Scala, although can be switched to any programming language easily. All the FTL needs to do is learn a set of *ioctl* commands supported by the block driver. The present design of the FTL is further divided into two layers (Figure 3.10). The top layer is the one which provides the desired FTL functionalities. The bottom layer is a generic FTL layer which acts as a server to the

block driver. Its main purpose is to provide APIs to the top layers, such that the development of the FTL becomes easier and would be able to integrate the FTL logic to the emulator a lot quicker. Basically, it acts as an interface between the top layer's FTL functionality and the block driver. In the current implementation, we have integrated the emulator with Direct Page-mapped FTL, DFTL[43] and CPFTL.

Putting It All Together

The emulator reserves a desired amount of memory at system boot time, which is later acquired by the block driver when the module is inserted into the kernel. The block driver then passes through a few initialization steps, such as setting up the channel worker threads, allocating corresponding channel request queues, other necessary data structures required to negotiate with the kernel and finally let the kernel know that the block driver is ready to accept I/O requests. At this point, it creates a device node, `/dev/ssd_ramdisk` through which the workload applications could read/write data from/to disk. The FTL layer, upon its launch, also interacts with the block driver through the same device node, although to exchange the LBA-PPN information through a set of *ioctl* commands.

When the user starts a workload application with the emulator disk as its target, the filesystem layer inside the kernel assembles the request and forwards it to the block device driver. The requests are addressed in the form of LBAs. The LBAs are then forwarded to the FTL, running in the user-space, which applies its algorithm and returns back a PPN to the driver. Based on a round-robin fashion, the received PPNs are allotted a particular channel and the request is then forwarded to the corresponding channel queue. If the channel queue has any data lying in it, the respective channel thread will be woken up and would perform the I/O in a FIFO manner, thus satisfying the request.

Validation

In this section, we present the validation results of our emulator. The evaluation was conducted under the environment shown in the Table 3.5.

Table 3.5: Evaluation Platform

Processor	Intel Xeon Hex core E5-2603 @ 1.6 GHz
Memory	64 GB, of which 32 GB reserved for the emulator
Operating System	Linux-4.3.3
Evaluation tool	iometer

We have validated the emulator against one of the commercial SSDs. Due to commercial reasons, the vendor and the model are not disclosed. The tests were conducted using the popular *iometer* tool [103], whose workload operates directly on the raw disk, instead of going through the filesystem layer. Basically, an emulator mimics the internal behavior of it subject. In its current stage of development, our emulator mimics a very basic architecture of an SSD. As a result, we have only validated the behavioral patterns varied against different request sizes, but not the value to value comparison with the commercial SSD.

Table 3.6: SSD Parameters for Evaluation

Emulator Page Read latency	25 μsec
Emulator Page Write latency	300 μsec
Emulator Block Erase latency	3000 μsec
Parallelism	4-Way Channel level Parallelism
FTL	Page-level FTL

Usually, the vendors of the commercial SSDs do not disclose its internal characteristics, such as the read/write latencies, the amount of parallelism involved, internal DRAM size and so on. As a result, we cannot predict the performance of the SSD and tune our emulator in a similar fashion.

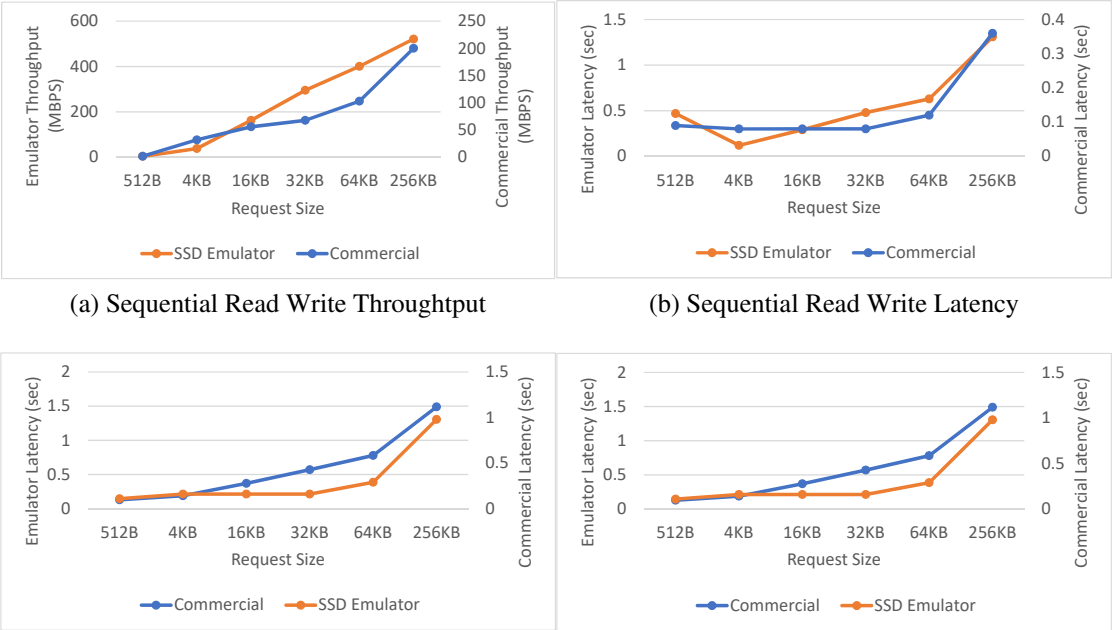
Hence, in our emulator design, we had come up with a very simple design for evaluation, which is given in Table 3.6.

Figure 3.11 shows the validation with respect to throughput and latency, varying the request size from 512 Bytes to 256 KB. The workload is chosen to be a mix of 75% reads and 25% writes. Each test was executed for thirty minutes and the average of it was taken. Figure 3.11(a) shows the sequential Read-Write throughput. We see that, as the request size increases the commercial SSD's throughput increases, so does the emulator's, although at different rates. As stated earlier, the validation is carried out on a behavioral level (pattern), rather than on the basis of values. Similar is the observation with the random Read-Write patterns shown in Figure 3.11(c). Figure 3.11(b) shows the average response time of the SSDs. It can be observed from the figure the unpredictable nature of the commercial SSD. The response time decreased from 512 Bytes to 4 KB, but increased from there. Again after 64 KB, the rate of increase grew further. However, the performance of the SSD emulator is linearly decreasing at a slower rate as the request size decreases, and further, after 64 KB, it grew rapidly. Similar is the case with the Figure 3.11(d) which shows that the emulator's behavior is almost traced as that of the commercial SSD.

Usually, systems fragment the request size to a granularity of 4 KB. Referring in terms of values, all the four performance figures show that for a request size of 4 KB, the performance of both the SSDs have an error margin of approximately 10%. Hence, to summarize, the traceable patterns of the SSD emulator with that of the commercial SSD and the similar performance observation at 4 KB request size validates the design of the emulator.

Filesystem Evaluation

As an application of our emulator, we have evaluated the performance of various filesystems against varying FTLs. We have taken two of the most commonly used filesystems, *ext4* and *btrfs*



(a) Sequential Read Write Throughtput

(b) Sequential Read Write Latency

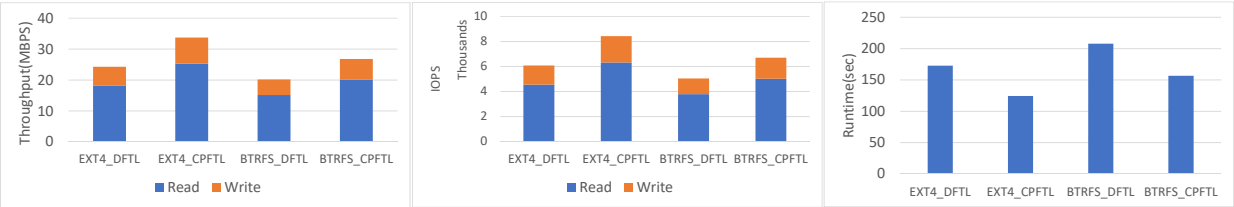
(c) Random Read Write Throughtput

(d) Random Read Write Latency

Figure 3.11: Validation of Emulator

and compared each of its performance with respect to throughput, IOPS and total I/O time against two FTLs. The results are presented in Figure 3.12. We have implemented and applied two of the most simple FTLs, DFTL and CPFTL (CPFTL). DFTL uses a cache to store the page-table, such that obtaining the mapping information need not go through another page read operation. The evaluation was performed using the popular *fio* utility, which creates a file on a mounted partition, perform the I/O and populate the results. In the validation section, as the results were more *true* for a request size of 4 KB, we have applied the same here for evaluating the filesystems. A random read-write (75% read and 25 % write) workload was generated such that it would perform I/O to a 4 GB file. Furthermore, *fio* was configured such that all the I/Os bypasses the host buffer and performs a direct I/O.

Eyeballing through the performance results in Figure 3.12 shows that, by varying different FTLs,



(a) (b) (c)

Figure 3.12: Performance evaluation of filesystems under varying FTLs

for the same filesystem, results could vary drastically. As an example, in Figure 3.12a), which compares the throughput, we see that upon moving from DFTL to CPFTL, the *ext4's* performance improved by 42%. This makes sense because the workload over here is a random I/O and it might be the case that DFTL is thrashing, such that more swaps of the cache are taking place than the actual I/O. This case is not possible in CPFTL as the I/O is performed predictively, and hence perform better. Moreover, for the same FTL, we see that *ext4* performs better in comparison to *btrfs*. The same performance results propagate through the comparison for IOPS and total execution time taken to perform the I/O on the 4 GB file.

Discussion & Related Work

In previous sections, we described the general characteristics shared by many earlier FTLs. However, there are some exceptions to many of these characteristics that are worth noting.

While most of the work on FTLs has focused on adapting to workload requirements, several designs have focused on other aspects of FTLs. Both log-buffer based hybrid FTLs [104, 105] and workload-adaptive hybrid FTLs [81] are proposed, where both the block-level mapping and page-level mapping are employed to manage flash memory. Gupta et al. [43] proposed the demand-based page- level FTL, called DFTL, which leverages temporal locality by adopting a segmented LRU

replacement algorithm to avoid the inefficiency of hybrid FTLs and to reduce the RAM requirement for the page-level mapping table. Also, several variants of page-level FTL [90, 106, 91] proposed different cache algorithms to leverage the "in-page" locality by batch loading and updating the mapping entries within one page. SHRD [107] proposed to improve spatial locality by sequentializing the workloads in host. This work improves the "in-page" locality by placing the correlated mapping table within one correlated mapping page.

Stoica leveraged the write locality by isolating the frequently updated pages on flash [99]. Kim isolated the write operations from different applications by passing the process ids to SSDs to reduce the garbage collection contention [108]. ASA-FTL [109] separates hot/cold data in FTL. This work absorbs the frequent mapping entry updates in the cache.

C-Miner [82] leverages block correlation in data prefetch process. Trapfetch [110] detects bursts of disk reads, determines the appropriate addresses and breakpoints to prefetch the data. However, these prefetch technologies inevitably results in the FTL to read more translation pages in a correlation-unaware manner and decrease performances. uFLIP-OC [111] study how the I/O pattern impact the parallelisms in open-channel SSD. This work is the first to leverage block level correlation in FTL.

CHAPTER 4: CONCLUSIONS

In this dissertation, both top-down and bottom-up approach have been used to optimize the performance of application and data storage. In the top-down approach, we avoid I/O contention by expose storage level information and redesign the data processing engine. In the bottom-up approach, we study the I/O correlation in the SSD firmware and redesign the mapping and cache management scheme. In summary, we make the following conclusions.

- We have presented ApproxSSD, a framework that optimizes performance of approximate data processing and the parallel storage and processing system. ApproxSSD leverages the on-disk data layout to reduce the possible workload imbalance and I/O contentions among SSDs, and thus speed up the overall performance. Instead of executing the data sampling at the application level, ApproxSSD provides a sampling ratio in the parallel storage engine. The parallel storage engine then launches multiple data sampling threads for each disk to maximize the disk throughput. Additionally, we developed a straggler drop which captures the run-time disk performance and balances the amount of data served by each disk. We have prototyped ApproxSSD in Scala, which is open sourced on https://github.com/janzhou/approxssd. Experimental results show that our prototype system outperforms the baseline solution by up to 2.7 times faster at 10% sampling ratio under WordCount workloads.

- We have presented ArchSampler, a sampling-based parallel data approximate framework for future systems that are equipped with NVM-based main memories. ArchSampler leverages the in-memory data layout to reduce the potential imbalance in accessesing to the memory banks and contentions on memory banks, i.e., bank conflicts., and thus reduces the overall memory access latency and speed up the performance. First, ArchSampler takes the data to banks mapping into consideration while selecting samples. Because the sampling-

based approximation is flexible on which data should be selected as samples, ArchSampler can effectively balance the workload among banks without reducing the approximate accuracy. Second, to remove the bank-level contentions, ArchSampler embraces the concept of shared-nothing threads by restricting the data assigned to one thread to span only one bank (or unique set of banks). Our evaluation shows that ArchSampler outperforms the random sampling by up to **1.62** (*1.20* on average).

- We developed a Correction-aware Page-level FTL (CPFTL), which leverages sematic link information hidden in workloads. We observed two reasons that applications usually cannot fully exploit the peak performance of SSDs. First, correlated data access often requires mapping entries that are scattered on multiple translation pages. As a result, the cache hit ratio at the FTL level is very low and a large portion of the overall page read inside SSDs are from internal translation pages. Second, because of write skewness, a small portion of the translation pages are updated more often than others. Consequently, the GC process within the translation blocks needs to move more clean pages. Based on these observations, we then propose a new Correlation-Aware Page-Level FTL, called CPFTL. With consideration of data correlations, CPFTL employs correlated a mapping page to organize the correlated mapping entries in the same page. In addition, CPFTL use a correlation prediction dictionary to estimation and locate the correlated mappings when requests arrive. Finally, we use a skew-aware dirty entry index to perform batch updates for the dirty entries and reduce the GC overhead in translation pages. Extensive evaluations with enterprise workloads show that CPFTL can efficiently improve the cache hit ratio in FTLs and reduce the overall page read inside SSDs under workloads with extremely low temporal locality.

LIST OF REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, ACM, 2007.

[2] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: a 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 477–488, ACM, 2010.

[3] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[4] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 31–46, 2012.

[5] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[6] A. Zlateski, K. Lee, and H. S. Seung, "Znn-a fast and scalable algorithm for training 3d convolutional networks on multi-core and many-core shared memory machines," *arXiv preprint arXiv:1510.06706*, 2015.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance.," in *USENIX Annual Technical Conference*, 2008.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-

memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012.

[9] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*.

[10] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 301–316, 2014.

[11] M. B. Cohen, Y. T. Lee, C. Musco, C. Musco, R. Peng, and A. Sidford, "Uniform sampling for matrix approximation," in *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, ACM, 2015.

[12] P. Li, K. W. Church, and T. J. Hastie, "Conditional random sampling: A sketch-based sampling technique for sparse data," in *Advances in neural information processing systems*, 2006.

[13] L. P. Yaroslavsky, G. Shabat, B. G. Salomon, I. A. Ideses, and B. Fishbain, "Nonuniform sampling, image recovery from sparse data and the discrete sampling theorem," *JOSA A*, 2009.

[14] J. Wang, J. Yin, J. Zhou, X. Zhang, and R. Wang, "Datanet: A data distribution-aware method for sub-dataset analysis on distributed file systems," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 504–513, IEEE, 2016.

[15] X. Zhang, J. Wang, and J. Yin, "Sapprox: enabling efficient and accurate approximations on sub-datasets with distribution-aware online sampling," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 109–120, 2016.

[16] A. Czumaj and V. Stemann, "Randomized allocation processes," in *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, IEEE.

[17] M. Raab and A. Steger, "balls into binsa simple and tight analysis," in *Randomization and Approximation Techniques in Computer Science*, Springer, 1998.

[18] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," in *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[19] M. Jung, W. Choi, J. Shalf, and M. T. Kandemir, "Triple-a: a non-ssd based autonomic all-flash array for high performance storage systems," in *ACM SIGPLAN Notices*, vol. 49, pp. 441–454, ACM, 2014.

[20] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler.," in *FAST*, 2012.

[21] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pp. 1–12, IEEE, 2013.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, USENIX Association, 2006.

[23] O. Rodeh and A. Teperman, "zfs-a scalable distributed file system using object disks," in *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, IEEE, 2003.

[24] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, p. 16, ACM, 2014.

[25] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 383–397, ACM, 2015.

[26] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[27] S. Lohr, *Sampling: Design and Analysis*. Cengage Learning, 2009.

[28] B. Efron, *Bootstrap methods: another look at the jackknife*. Springer, 1992.

[29] "Wikimedia data dumps." `https://dumps.wikimedia.org/backup-index.html`.

[30] "Wikimedia page views." `https://dumps.wikimedia.org/other/analytics/`.

[31] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42, ACM, 2013.

[32] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," 2015.

[33] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM.

[34] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," 2014.

[35] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[36] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[37] X. Meng and M. W. Mahoney, "Low-distortion subspace embeddings in input-sparsity time and applications to robust linear regression," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, ACM, 2013.

[38] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007.

[39] Y. Hua, H. Jiang, and D. Feng, "Fast: near real-time searchable data analytics for the cloud," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2014.

[40] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for hadoop: Time to rethink?," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 20, ACM, 2013.

[41] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: processing billion-node graphs on an array of commodity ssds," in *FAST*, 2012.

[42] S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung, "Design and analysis of flash translation layers for multi-channel nand flash-based storage devices," *Consumer Electronics, IEEE Transactions on*, 2009.

[43] A. Gupta, Y. Kim, and B. Urgaonkar, *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*. ACM, 2009.

[44] D. Zheng, D. Mhembere, R. Burns, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," 2014.

[45] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: a revelation from millions of hours of disk and ssd deployments," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 263–276, 2016.

[46] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[47] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 2–2, USENIX Association, 2012.

[48] R. Frickey, "Data integrity on 20nm ssds," *Flash Memory Summit*, 2012.

[49] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error patterns in mlc nand flash memory: Measurement, characterization, and analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 521–526, EDA Consortium, 2012.

[50] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10, IEEE, 2010.

[51] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud analytics: Do we really need to reinvent the storage stack?," in *HotCloud*, 2009.

[52] W. Tantisiriroj, S. Patil, and G. Gibson, "Data-intensive file systems for internet services: A rose by any other name...(cmu-pdl-08-114)," *Parallel Data Laboratory*, p. 9, 2008.

[53] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: reconciling hdfs and pvfs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 67, ACM, 2011.

[54] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor, "Parallel access to files in the vesta file system," in *Supercomputing'93. Proceedings*, pp. 472–481, IEEE, 1993.

[55] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem.," in *FAST*, pp. 359–374, 2017.

[56] J. Kang, C. Hu, T. Wo, Y. Zhai, B. Zhang, and J. Huai, "Multilanes: Providing virtualized storage for os-level virtualization on manycores," *ACM Transactions on Storage (TOS)*, vol. 12, no. 3, p. 12, 2016.

[57] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–11, IEEE, 2015.

[58] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, 2013.

[59] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[60] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, *Overcoming the challenges of crossbar resistive memory architectures*, pp. 476–488. Institute of Electrical and Electronics Engineers Inc., 3 2015.

[61] HP, "The Machine: A new kind of computer." `http://www.hpl.hp.com/research/systems-research/themachine/`.

[62] Intel, "Intel 3D XPoint." `http://newsroom.intel.com/docs/DOC-6713`.

[63] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[64] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "Incapprox: A data analytics system for incremental approximate computing," in *Proceedings of the 25th International Conference on World Wide Web*, pp. 1133–1144, International World Wide Web Conferences Steering Committee, 2016.

[65] Linux, "Linux Direct Access of Files (DAX)." `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[66] A. Rodrigues, R. Murphy, P. Kogge, and K. Underwood, "The structural simulation toolkit: A tool for exploring parallel architectures and applications," *Tech. Rep. SAND 2007-0044C*, 2007.

[67] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1295–1306, 2014.

[68] M. De Choudhury, Y.-R. Lin, H. Sundaram, K. S. Candan, L. Xie, A. Kelliher, *et al.*, "How does the data sampling strategy impact the discovery of information diffusion in social media?," *ICWSM*, vol. 10, pp. 34–41, 2010.

[69] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.

[70] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*.

[71] J. Felsenstein, "Confidence limits on phylogenies: an approach using the bootstrap," *Evolution*, vol. 39, no. 4, pp. 783–791, 1985.

[72] A. Kulesa, M. Krzywinski, P. Blainey, and N. Altman, "Points of significance: sampling distributions and the bootstrap," *Nature Methods*, vol. 12, no. 6, pp. 477–478, 2015.

[73] A. Awad, G. R. Voskuilen, A. F. Rodrigues, S. D. Hammond, R. J. Hoekstra, and C. Hughes, "Messier: A detailed nvm-based dimm model for the sst simulation framework.," tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2017.

[74] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–10, IEEE, 2015.

[75] Q. Wang, D. Wang, and C. Hou, "Exploiting write power asymmetry to improve phase change memory system performance.," *Frontiers of Computer Science*, vol. 9, no. 4, pp. 566–575, 2015.

[76] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in mlc phase change memory," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–10, IEEE, 2012.

[77] J. J. McAuley and J. Leskovec, "From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 897–908, ACM, 2013.

[78] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 165–172, ACM, 2013.

[79] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica, "Blink and it's done: interactive queries on very large data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1902–1905, 2012.

[80] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 471–484, ACM, 2014.

[81] D. Park, B. Debnath, and D. Du, "Cftl: A convertible flash translation layer adaptive to data access patterns," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, pp. 365–366, ACM, 2010.

[82] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-miner: Mining block correlations in storage systems.," in *FAST*, vol. 4, pp. 173–186, 2004.

[83] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 144–154, IEEE, 2001.

[84] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the 41st annual*

IEEE/ACM International Symposium on Microarchitecture, pp. 222–233, IEEE Computer Society, 2008.

[85] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *ACM SIGOPS Operating Systems Review*, vol. 32, pp. 115–126, ACM, 1998.

[86] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: Exploiting disk layout and access history to enhance i/o prefetch.," in *USENIX Annual Technical Conference*, vol. 7, pp. 261–274, 2007.

[87] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-aware prefetching at the storage server.," in *USENIX Annual Technical Conference*, pp. 377–390, 2008.

[88] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Proceedings of the 2009 First International Conference on Advances in System Simulation*, pp. 125–131, IEEE, 2009.

[89] "Umass trace repository," in *http://traces.cs.umass.edu/*.

[90] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-ftl: An efficient address translation for flash memory by exploiting spatial locality," in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pp. 1–12, IEEE, 2011.

[91] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level ftl to optimize address translation in flash memory," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 12, ACM, 2015.

[92] M. Athanassoulis and A. Ailamaki, "Bf-tree: Approximate tree indexing," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1881–1892, 2014.

[93] K. Ren and G. A. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system.," in *USENIX Annual Technical Conference*, pp. 145–156, 2013.

[94] C. Ruemmler and J. Wilkes, "Unix disk access patterns," in *USENIX Winter*, vol. 93, pp. 405–420, 1993.

[95] D. S. Roselli, J. R. Lorch, T. E. Anderson, *et al.*, "A comparison of file system workloads.," in *USENIX annual technical conference, general track*, pp. 41–54, 2000.

[96] S.-W. Lee and B. Moon, "Design of flash-based dbms: an in-page logging approach," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 55–66, ACM, 2007.

[97] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: random write considered harmful in solid state drives.," in *FAST*, p. 12, 2012.

[98] H. Kim and S. Ahn, "Bplru: A buffer management scheme for improving random writes in flash storage.," in *FAST*, vol. 8, pp. 1–14, 2008.

[99] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 733–744, 2013.

[100] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 75–88, ACM, 2014.

[101] A. Pagh, R. Pagh, and S. S. Rao, "An optimal bloom filter replacement," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 823–829, Society for Industrial and Applied Mathematics, 2005.

[102] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pp. 125–131, IEEE, 2009.

[103] T. IOMETER, "Iometer: I/o subsystem measurement and characterization tool," *Open source code distribution: http://www. iometer. org*, 1997.

[104] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, p. 18, 2007.

[105] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[106] M. Wang, Y. Zhang, and W. Kang, "Zftl: A zone-based flash translation layer with a two-tier selective caching mechanism," in *Communication Technology (ICCT), 2012 IEEE 14th International Conference on*, pp. 578–588, IEEE, 2012.

[107] H. Kim, D. Shin, Y. Jeong, and K. H. Kim, "Shrd: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device.," in *FAST*, pp. 271–284, 2017.

[108] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pp. 183–189, USENIX Association, 2015.

[109] W. Xie, Y. Chen, and P. C. Roth, "Asa-ftl: An adaptive separation aware flash translation layer for solid state drives," *Parallel Computing*, vol. 61, pp. 3–17, 2017.

[110] J. Won, O. Kwon, J. Ryu, J. Hur, I. Lee, and K. Kang, "Trapfetch: A breakpoint-based prefetcher for both launch and run-time," in *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, pp. 2766–2771, IEEE, 2017.

[111] I. L. Picoli, C. V. Pasco, B. Jonsson, L. Bouganim, and P. Bonnet, "uflip-oc: Understanding flash i/o patterns on open-channel solid-state drives," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, p. 20, ACM, 2017.