ArchSampler: Architecture-Aware Memory Sampling Library for In-Memory Applications

Jian Zhou, Jun Wang
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, US
{jzhou, jwang}@eecs.ucf.edu

***Abstract*—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 times slow-down for nave and default implementations of in-memory data sampling. Accordingly, we propose *ArchSampler*, an architecture-aware sampling library. The main idea is to exploits the 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.

I. Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as Phase-Change Memory (PCM)[1], [2] and Memristor[3], are promising candidates for building future memory systems[1]. Compared to DRAM, emerging NVMs promise high densities, near-zero idle power, and persistent applications. 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 number 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[4], [5]. For instance, in a recent work [5], 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 [6]. Figure 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.

In this paper, we propose *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

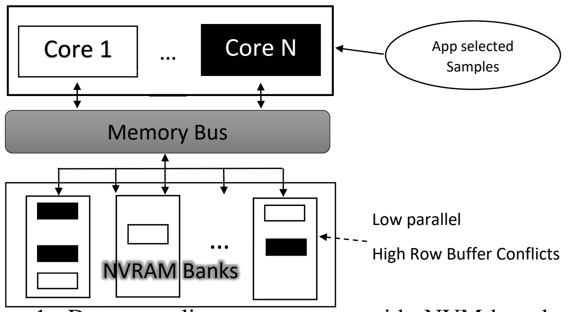


Fig. 1: Data sampling on systems with NVM-based main memories.

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)[7], 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).

The rest of the paper is organized as follows. First, in Section II, we present an overview of the architecture of emerging NVM technologies and briefly introduce the state-of-the-art sampling techniques. We additionally quantify the overhead of architecture-oblivious implementations. Section III details the design and implementation of ArchSampler. Later, in Section IV, we start with discussing our evaluation methodology and the workloads. Later, we evaluate ArchSampler, discuss its potentials and compare with architecture-oblivious implementations. Finally, we conclude our work in Section V.

II. 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 [8].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, the improved processing performance of sampling plays a key role in responsive analytics jobs.

NVM-Based memories are logically organized as groups of banks within 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

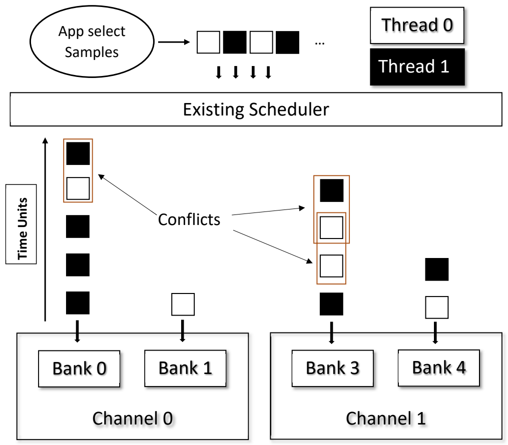


Fig. 2: Conventional Random Sampling.

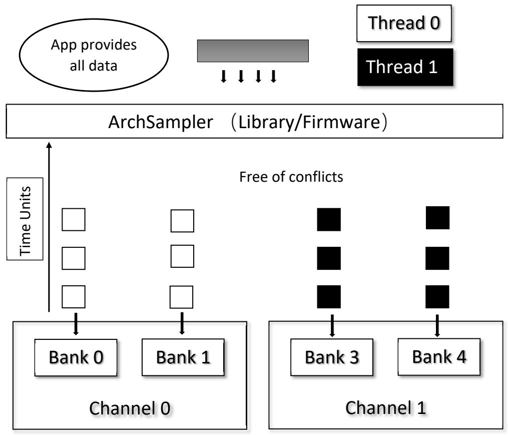


Fig. 3: Conceptual example showing benefits of Bank-aware Sampling.

banks. Therefore, the performance of NVMs is sensitive to the data access pattern.

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, 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 3 shows a conceptual example that applications 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 analyze the problem of conventional random sampling when NVMs are involved.

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 [4]. 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 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% [9], [10].

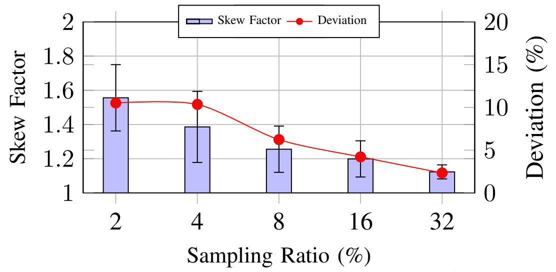
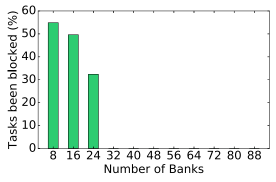


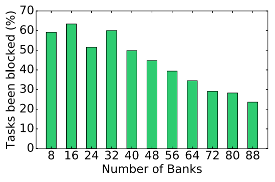
Fig. 4: Workload imbalance issue (Skew factor = max/avg NVM rows requested from banks).

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 and graph sampling [11]. 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 5, we did the computational analysis for the possibility of contention issue. The results show that, when the sampling ratio is lower than 10%, up to 60% of the memory request will be blocked.



(a) Sampling Ratio 100%.



(b) Sampling Ratio 10%.

Fig. 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.

III. Design and Optimizations

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

A. *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 earlier in Section II, 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.

B. *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

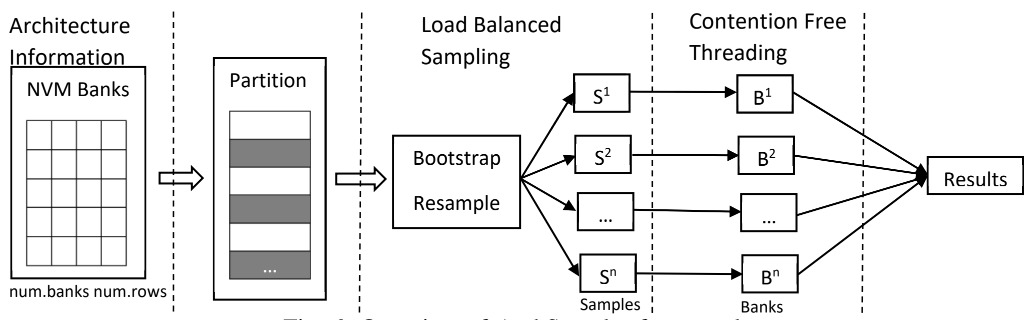


Fig. 6: Overview of ArchSampler framework.

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.

C. 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 [12]. Because of its simplicity and effectiveness, it can be used in almost any type of data and application [13], [4].

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.

D. IMPLEMENTATION: PUTTING IT ALL TOGETHER

To better understand how ArchSampler can be used by applications, Figure 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.

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.

IV. Evaluation

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

A. Methodology

To evaluate our design, we use the Structural Simulation Toolkit (SST) [7], 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 [14], 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 [15], [16], we use 150ns as our default latency. For PCM write latency, similar to [17], we use 500ns write latency of PCM cells. Table I 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 run all of our benchmarks from start until completion.

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

TABLE I: Simulated system configuration.

To mimic real-world sampling applications, we developed standalone multi-threaded microbenchmarks. Our implemented microbenchmarks resemble real-world workloads, such as word-count, calculate the average rating and finding the elements with the highest average rating on Amazon reviews and rating datasets[18], [19]. We also study two synthetic reads and write workloads. A complete list of workloads and the corresponding algorithm and dataset are shown in Table II.

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.

The addresses allocated by malloc are virtual addresses as seen by the program. As our algorithms work in close coordination 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 to have 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.

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

TABLE II: List of evaluated applications.

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 thread 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 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.

B. 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.

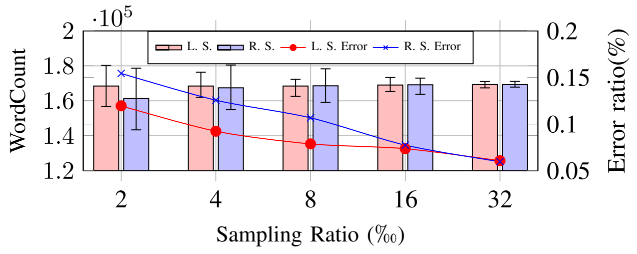


Fig. 7: Results of WordCount under different sampling ratio. (L. S.: Load-balanced sampling with estimated error bound, R. S.: random sampling with estimated error bound, L. S. Error: error rate of load-balanced sampling, R. S. Error: error rate of random sampling.)

1) Approximate Accuracy: 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 a 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 7 and Figure 8, the results show that the bank-aware sampling

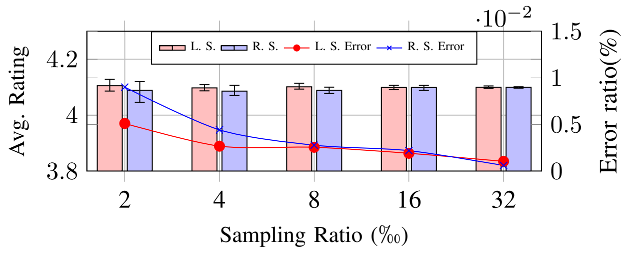


Fig. 8: Results of Average Rating under different sampling ratio.

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 [20].

2) Performance of Approximate Query: To quantify the potentials of ArchSampler, we study its impact on the performance of approximate queries. The 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.

In our experiments, we compare the four sampling and task scheduling schemes previously discussed in Section IV-A. In ArchSampler, we use bank-aware sampling and contention-free task scheduling scheme. To study the effect of contention-free threading, we then use a 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 9, 10 and 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 a logarithmic scale.

Figure 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 10 shows the latency summation of

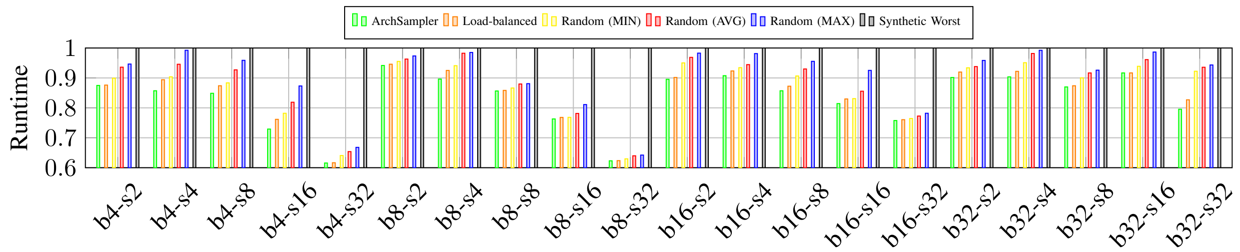


Fig. 9: Normalized Runtime performance of WordCount. Note the performance is normalized to the synthetic worst case. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

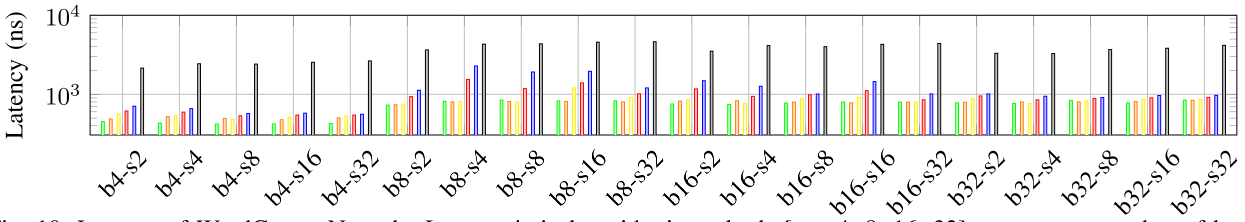


Fig. 10: Latency of WordCount. Note the Latency is in logarithmic scale. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

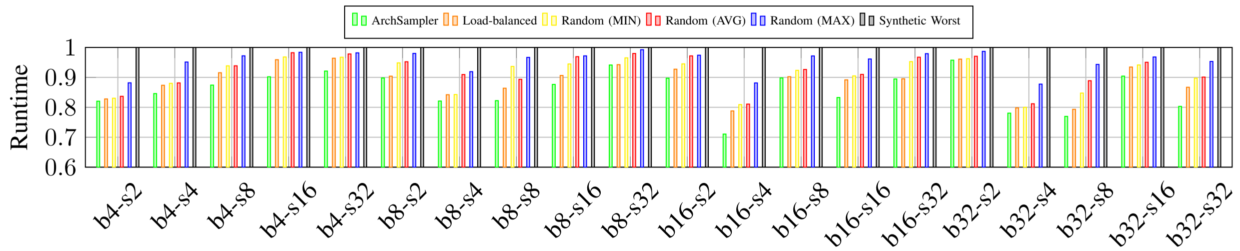


Fig. 11: Normalized Runtime performance of Avg. Rating. Note the performance is normalized to the synthetic worst case. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

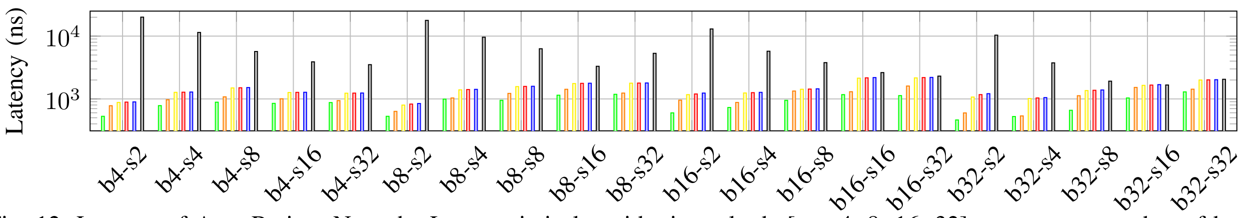


Fig. 12: Latency of Avg. Rating. Note the Latency is in logarithmic scale. bn[n = 4, 8, 16, 32] represent n number of banks. sm[m= 2, 4, 8, 16, 32] represent $m‰$ sampling ratio.

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 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 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 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%.

3) 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 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 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

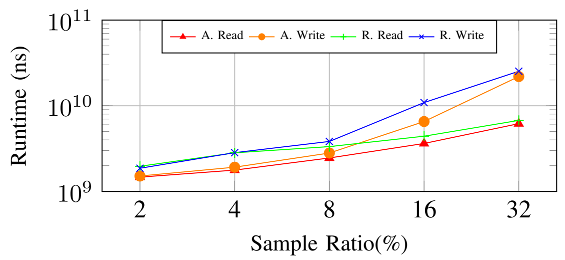


Fig. 13: Runtime performance for Synthetic Workloads. Note the Y label is in logarithmic scale. A. stands for ArchSampler. R. stands for Random.

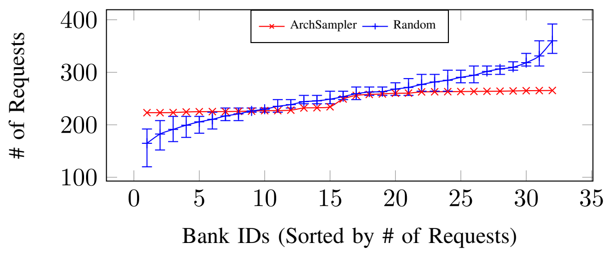


Fig. 14: Load balance in Synthetic Workloads.

at various sampling ratios. Figure 14 shows the number of 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).

V. Conclusion

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 accessing 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).

Acknowledgment

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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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*.

[5] 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.

[6] Linux, “Linux Direct Access of Files (DAX).” https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.