A Study of Sorting Algorithms on Approximate Memory

Shuang Chen[1,2*] Shunning Jiang[1,2*] Bingsheng He[2] Xueyan Tang[2]

[1]Shanghai Jiao Tong University, Shanghai, China
[2]Nanyang Technological University, Singapore

ABSTRACT

Hardware evolution has been one of the driving factors for the redesign of database systems. Recently, approximate storage emerges in the area of computer architecture. It trades off precision for better performance and/or energy consumption. Previous studies have demonstrated the benefits of approximate storage for applications that are tolerant to imprecision such as image processing. However, it is still an open question whether and how approximate storage can be used for applications that do not expose such intrinsic tolerance. In this paper, we study one of the most basic operations in database–sorting on a hybrid storage system with both precise storage and approximate storage. Particularly, we start with a study of three common sorting algorithms on approximate storage. Experimental results show that a 95% sorted sequence can be obtained with up to 40% reduction in total write latencies. Thus, we propose an *approx-refine* execution mechanism to improve the performance of sorting algorithms on the hybrid storage system to produce precise results. Our optimization gains the performance benefits by offloading the sorting operation to approximate storage, followed by an efficient refinement to resolve the unsortedness on the output of the approximate storage. Our experiments show that our *approx-refine* can reduce the total memory access time by up to 11%. These studies shed light on the potential of approximate hardware for improving the performance of applications that require precise results.

1. INTRODUCTION

Approximate computing [15, 41, 42, 52, 54, 60] is a hot research area that trades the accuracy/precision of application results for performance and/or energy efficiency. In all the previous studies [15, 41, 42, 52, 54, 60], approximate computing targets at the applications with intrinsic tolerance to inaccuracies in computation, such as computer vision and image processing. Meanwhile, the idea of approximate computing has been expanded to storage (i.e., *approximate storage* [36, 54, 48]). Nowadays, the scaling of common memory technologies like DRAM and flash is approaching its limit. New memory technologies, such as non-volatile memory (NVRAM), are emerging. For example, multi-level cell phase change memory (MLC PCM), the leading contender among NVRAMs, is shown to be a competitive choice for databases [47, 12, 27], and other applications [54].

In recent years, several kinds of approximate storage have been proposed to improve the energy efficiency and performance of memory systems [36, 48, 51, 35, 54]. They trade the precision of results for increased write performance or reduced energy consumption. A previous study shows 1.7× improvement on write latencies with quality loss under 10% for MLC PCM [54]. This is significant in PCM performance, because write latency issues almost drag the adoption of PCM reported by previous studies [30, 34]. Approximate storage also imposes little burden on manufacturers in the sense that it requires only reasonable modification to the existing hardware [54].

Despite the performance and power advantage of approximate hardware, its application scope is limited to the applications with inherent error tolerances, as in all the previous studies [15, 41, 42, 52, 54, 60]. Yet, many real applications are without intrinsic tolerance, which we define to be *precise computing*. The results of these applications are often required to be precise. For example, an imprecision in bank account storage can lead to millions of dollars in loss. A natural and challenging question is whether approximate storage can still be used for improving the performance and/or energy efficiency of precise computing.

In this paper, we study sorting, an important operation in (database) systems impacting the performance of various operators and algorithms [32], in the context of approximate memory. With heavier sorting workloads arising in high performance computing, engines of web indexing, and database systems, memory performance becomes an increasingly serious problem in conventional sorting algorithms [23]. The sorting algorithm requires a precisely sorted output, based on the input data. We conduct our study in two steps.

Step 1. We study the trade-off between sortedness and write performance if sorting is performed in the approximate memory *only*. The results and findings in this step are considered as the best performance/energy gains that we can achieve by leveraging approximate memory.

Step 2. To expand traditional approximate computing to a broader scope, how can we leverage the approximate

*Shuang Chen and Shunning Jiang contribute equally to this work and are listed alphabetically.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

 ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882908

memory to improve the performance of sorting algorithms, but still produce *precise results*?

Particularly, we revisit three classic and popular sorting algorithms: *quicksort*, *mergesort*, and *radixsort* (the first two are comparison-based, while the last is not) using approximate main memory. Simulation results show that without the requirement of precise outputs, *quicksort* and *radixsort* can get a nearly sorted sequence while reducing $30\% - 40\%$ write latencies on the approximate memory.

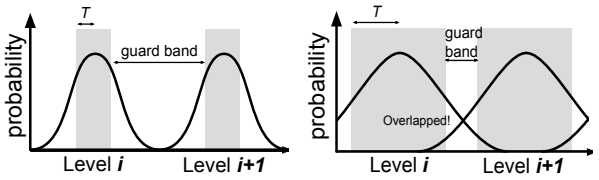
Unlike previous studies that focus on approximate computing with approximate hardware, we design a sorting mechanism on the hybrid memory system with both precise and approximate memory for guaranteeing precise results. We use a novel algorithmic level execution mechanism on hybrid approximate/precise memory to produce precise results. Specifically, we propose an *approx-refine* mechanism in which the approximate memory acts as an accelerator. We first copy the input data from the precise memory to the approximate memory, and then perform an existing sorting algorithm on the approximate memory. Finally, the approximate results are refined to become precise in the precise memory. If the sorting algorithm can deliver a nearly sorted output on approximate memory, only a lightweight refinement is needed afterwards. As a result, the cost of refinement and data copies between precise memory and approximate memory can be compensated by the gain of offloading the sorting algorithm to the approximate hardware. Our *approx-refine* scheme is generally applicable to different sorting algorithms with little modification. Simulation results show that, up to 11% write latencies can be saved using the *approx-refine* mechanism in hybrid precise/approximate memory. However, *mergesort* does not show any benefit under the hybrid execution. Therefore, approximate storage is not favored by every sorting algorithm.

This paper has made the following major contributions. First, we showcase that approximate hardware can also be used for improving the performance of *precise* computing, which broadens the application scope of approximate hardware. Previously, approximate hardware is only used for approximate computing. Second, we develop and evaluate common sorting algorithms on the hybrid storage systems with both precise and approximate storage, and demonstrate the system and architectural insights of achieving precise computation on approximate hardware. To the best of our knowledge, this is the first systematic study of sorting algorithms on approximate storage.

The rest of the paper is organized as follows. Section 2 introduces the approximate memory model. In Section 3, we study sorting using approximate memory only, without the requirement of precise results. In Section 4, to produce precise results, we propose the *approx-refine* mechanism and introduce the detailed design of the refinement. Section 5 presents the experimental results using the *approx-refine* mechanism. Section 6 reviews the related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

In this paper, we focus on phase change memory (PCM), as PCM is a kind of promising NVRAM and is shown to be a competitive choice for databases [47, 12, 27]. This section briefly introduces the basics and modeling of precise memory, and the approximate memory model from Sampson et al. [54]. The model from Sampson et al. captures the performance and precision tradeoff on PCM. For more details on the modeling of PCM, we refer readers to the following papers: approximate memory modeling [54], unidirectional shift [67], PCM programming feedback control loop [44], and successive approximation analog-to-digital-converter (ADC) for PCM systems [46].



(a) Precise MLC (b) Approximate MLC

Figure 1: Differences between a precise (a) and approximate (b) multi-level cell (adapted from [54]).

2.1 Modeling Multi-Level Cell

NVRAM stores data as an analog value. For density scaling purpose, multi-level cell (MLC) stores multiple bits in one cell by dividing the range of the underlying analog value into more levels than single-level cell (SLC). SLC exposes only two states (logic "1" and "0"). The target analog range of each digital level in MLC is narrower than SLC. The multiple levels are separated by *guard bands*, which is controlled by the width of target analog range T as illustrated in Figure 1(a). The curves in Figure 1(a) show the probability of the analog value read from a cell after writing to it. In the precise MLC, the guard band is wide enough to distinguish two consecutive levels safely.

We adopt the MLC model from Sampson et al. [54], which abstracts the read/write model of MLC as follows. Let v be the internal analog value of a cell. In a cell write operation, a digital value d is first converted to the corresponding analog value v_d, Due to the non-determinism of analog writes, v would then be set to $WRITE(v_d)$ instead of v_d, where $WRITE$ is a function to produce an acceptable analog value by an iterative *program-and-verify* (P&V) process within a PCM write. A cell read operation retrieves an analog value $READ(v)$ and quantizes it to a digital value, where $READ$ is a function involving the perturbation. In the following, we present more details on $WRITE$ and $READ$.

2.1.1 MLC Write

Cell write operations are inherently non-deterministic on account of the analog nature. Due to process variation and material fluctuations, a single *cell write* pulse is usually insufficient to program the target analog value precisely. Hence, in both industry and academia, most MLC designs, including PCM [5, 7, 16, 43, 44, 46] and Flash [57], adopt an iterative *program-and-verify* (P&V) process to guarantee that the analog value is programmed within the exact value range.

Each write operation first resets the analog value to zero, and then iteratively performs P&V until the actual analog value reaches the target range (i.e., the grey area in Figure 1). Pseudo code of a cell write operation is shown in **Function** WRITE. The non-determinism of each programming iteration can be modeled as a normal distribution $N(\mu, \sigma^2)$ [54]. Positive constant β reflects the disturbance

of a single programming step. T is an important parameter for the P&V process, and represents the width of target analog range. Both T and β determine the average number of programming iterations to finish a cell write.

Function WRITE(v_d) (adopted from [54])

Input: v_d - the analog value to be programmed
Output: v - the actual analog value after P&V

begin
  $v \leftarrow 0$; $\#P \leftarrow 0$;
  `/* This while loop represents P&V process. The number of iterations #P is inversely proportional to cell write performance. */`
  while v *has not reached* $[v_d - T, V_d + T]$ **do**
    $v \leftarrow v + N(v_d - v, |\beta(v_d - v)|)$;
    $++\#P$;
  end
  return v ;
end

2.1.2 MLC Read

Reading from MLC involves retrieving an analog value and quantizing it. The latency of a quantization is linear in the number of bits, as suggested by Qureshi et al. [46]. In practice, cells suffer from variation and unidirectional shift. The read model of Sampson et al. [54] is given as below.

$$READ(v) = v + N(\mu, \sigma^2) \cdot log_{10} t_w$$

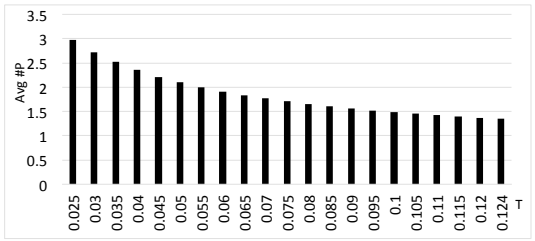
where $N(\mu, \sigma^2)$ represents the variation modeled as a normal distribution [67], and t_w is the time elapsed after the cell write operation. $log_{10} t_w$ is the multiplier on account to drift. The drift is modeled as the logarithm of time elapsed since the cell was written.

2.2 Approximate MLC

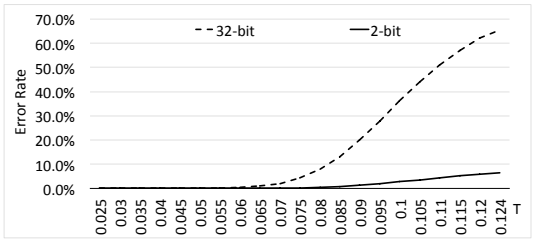
An approximate MLC configuration reduces the guard band to provide better write performance and energy efficiency. The basic idea is to reduce the number of P&V iterations by increasing the parameter T; intuitively, larger target range leads to less latency. Figure 1(b) shows the differences of an approximate MLC from the precise one.

To study the average number of P&V iterations ($\#P$) of MLC with different precisions, we take 4-level MLC as an example and vary T from 0.025 (almost precise) to 0.125 (no guard band), and for each value of T, we conduct a Monte-Carlo simulation: writing a random value to a 4-level MLC cell/a random 32-bit number to sixteen concatenated 2-bit cells, for 100,000,000 times respectively. Fig. 2 shows the write performance and error rate of 4-level MLC cell.

Overall, the approximate cell model used in this paper can be simply considered as a probabilistic error model with parameter T. Varying T implies different trade-offs between error rate and write performance. When T is small ($T \leq 0.025$), the memory is precise because of the large guard band. With the increase of T, the guard band shrinks and errors occur with higher probability, which results in a smaller $\#P$ and hence better write performance. Take $T = 0.1$ as an example. The average number of iterations in a P&V process ($\#P$) is halved in comparison with



(a) Avg #P



(b) Error Rate

Figure 2: The impact of T on write performance and accuracy for 2-bit MLC cell (from Monte-Carlo simulation).

$T = 0.025$, showing around 50% reduction in cell write latency.

For the analysis in Section 4.3, we define

$$p(t) = \frac{avg.\ \#P\ when\ T = t}{avg.\ \#P\ when\ T = 0.025} \approx \frac{avg\ \#P\ when\ T = t}{3}$$

as the ratio of the numbers of P&V iterations required on approximate memory that on precise memory. The lower $p(t)$ is, the more the write latency is reduced, and the higher the degree of imprecision in the approximate memory is.

2.3 Interfaces for Approximate Memory

The approximate MLC used in this paper can be produced by the same manufacturing process as precise MLC, except for a different analog range width, which piles little pressure on memory chip manufacturers. Hence, an approximate memory module can be integrated with other precise modules on the same memory channel.

The programming interfaces and assembly language support used in this paper are similar to previous literature [54]. Specifically, method *approx_alloc*($size$) allocates an array on approximate memory and returns a pointer. All memory access statements to an approximate array are compiled to *ld.approx* and *st.approx*. The OS kernel is modified to allow *approx_alloc* to allocate space only on approximate DIMMs, and to translate *ld/st.approx* back to normal *ld/st* with approximate array addresses.

We study sorting algorithms on a hybrid memory system with both precise and approximate memory support. Figure 3 shows how the approximate memory is integrated into the system. The juxtaposition of precise memory and approximate memory lies upon the storage system; OS kernel and assembly language are enhanced to support memory page allocation and translation of hybrid memory system. CPU and cache architecture in the case of hybrid memory system is the same as original system.

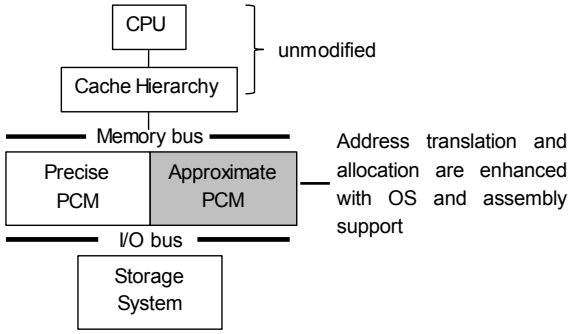


Figure 3: Memory System with Approximate Memory

L1 cache	32KB, LRU, write through
L2 cache	2MB, 4-way, LRU, write through
L3 cache	32MB, 8-way, LRU, 10ns access latency, write through
Main Memory	8GB PCM, 4KB page, 4 ranks of 8-bank each, 32 entries write queue/bank, 8 entries read queue/bank, read priority scheduling
Precise PCM Latency	T=0.025 data read: 50ns, data write: 1μs

Table 1: Memory simulator parameters

Category	Parameter	Note
Level	$L = 4$	each cell stores 2 bits
Read model	$\mu = 0.067$ $\sigma = 0.027$	read fluctuation
	$t = 10^5 s$	elapsed time for *drift*
Write model	$\beta = 0.035$	write fluctuation
	$T = 0.025$	precise case: #P=2.98
	$T \in (0.025 \ldots 0.125)$	approximate cases

Table 2: Parameters for precise and approximate MLC. We reuse the parameter of Sampson et al. [54].

3. SORTING IN APPROXIMATE MEMORY

We start with a study of common sorting algorithms on approximate memory. The entire main memory used in this section's experiments is approximate memory; the imprecision of approximate memory causes unsortedness in the output of sorting executions. In this section, we focus on two issues: 1) how the approximate memory would impair different sorting algorithms, and 2) the trade-off between sortedness and write performance.

3.1 Sorting Algorithms

The following three classic sorting algorithms are studied: *mergesort*, *quicksort* and *radixsort*. Our implementation has inherited various implementation techniques from the previous studies (e.g., [6, 4]). To reduce writes, some of the write-optimized techniques including write combining by software managed buffers [4] are adopted whenever appropriate. More details are given below. The input data are assumed to be an array X of n elements (or *records*).

Mergesort Mergesort is a comparison-based divide-and-conquer sorting algorithm, and is commonly used in databases. A mergesort execution on array X requires $nlog_2n$ data write operations. At the first level of mergesort, the input chunk size is designed to fit into the L2 cache.

Quicksort Quicksort is also a commonly used comparison-based sorting algorithm. On average, it takes $nlog_2n$ data comparisons and $\frac{nlog_2n}{2}$ data writes to sort n items. The average time complexity of quicksort is $O(nlog_2n)$, but its worst case is $O(n^2)$. We implement a randomized quicksort algorithm. The pivot is chosen randomly to reduce the probability of worst cases.

Radix sort Radixsort is a non-comparative sorting algorithm which sorts elements by grouping those who share the same value in some significant position. Two variants are studied here: *Least Significant Digit Radix Sort* (LSD) and *Most Significant Digit Radix Sort* (MSD). LSD starts from the least significant digit and moves towards the most significant digit, while MSD works the other way around. We implement a simple version of LSD and MSD using queues as buckets. We have adopted and tuned multi-pass partitioning in radixsort like the previous study [4]. The number of bins in each pass for LSD and MSD is an important tuning parameter, and we evaluate 3-bit, 4-bit, 5-bit and 6-bit for the study, i.e., with 8, 16, 32 and 64 buckets, respectively. Unless stated otherwise, LSD and MSD represent 6-bit LSD and 6-bit MSD, respectively, since they usually achieve the best total write latency among different numbers of bins.

3.2 Methodology

As no approximate memory product is available in market and no simulator with PCM support is open-source, we ran experiments using an in-house memory simulator which models PCM reads and writes in detail. Because of large data sets, the simulator is trace-driven, where traces are obtained from the actual execution on a real machine. The machine is configured with Intel Xeon Processor E5-1650 and 8GB DDR3 SDRAM. For simplicity, only one core is used while collecting memory traces.

Table 1 gives the detailed parameters of our memory simulator. The settings of those parameters are consistent with previous literatures [30, 12]. For simplicity, we assume a write-through cache, which ensures that every data write must go to the main memory. As write performance is a dominated factor for sorting algorithms on PCM, we mainly look at the total latencies spent on memory writes. Table 2 gives the detailed parameters of our MLC PCM modeling which are inherited from [54]. We use 4-level (2-bit) PCM cell for our simulation, as 2-bit MLC is the most popular choice in NAND Flash reported by Micron Inc. [37]. As suggested in previous studies [8, 38, 54], the raw bit error rate (RBER) of precise MLC is typically 10^{-8}.

A 32-bit integer is stored in sixteen concatenated 2-bit cells [54]. Input data contain an array of keys and an array of payload in the form of integers (i.e., record IDs). The key values are uniformly distributed 32-bit integers. The payload (record IDs) is essential in database workloads to complete the entire query processing according to the sorted results [49]. In the next section, the record IDs are stored

in the precise memory, and are used to refine and "recover" the totally sorted data.

3.3 Measure of Sortedness

Suppose an array of n elements, X, is to be sorted in increasing order. During the execution in the approximate memory, the elements of X may become imprecise, and thus the final X may not be fully sorted. To describe the sortedness of a sequence X after sorting in the approximate memory, we choose the measure *Rem* [9], which is defined as

$$Rem(X) = n - max\{k | X \text{ has an ascending subseq. of length } k\}.$$

Intuitively, *Rem* indicates the number of elements that should be removed to produce a sorted sequence. *Rem* ratio (i.e., $\frac{Rem(X)}{n}$) should be small if the array X is nearly sorted.

There are other measures of sortedness, as surveyed in the previous study [20]. Still, we choose the definition of *Rem*, because it is more intuitive than other measures like Inv (the number of inversion pairs in a sequence [20]). It motivates us to design a lightweight refine algorithm introduced in Section 4.2. More details on how to leverage the sortedness are given in Section 4.

3.4 Experimental Results

To study the correlation between the proportion of imprecise elements and the sortedness (i.e. *Rem* ratio), we vary the size of guard band by tuning T to sort $16,000,000$ (16M) random integers in the approximate memory. Since our target is to study the imprecision rather than to recover the sorted data, the payload array is not accessed in the study of this section. We define *write reduction* as the percentage of total memory write latency saved from using approximate memory compared with using precise memory only. More formally, total memory write latency is abbreviated to $TMWL$, then

$$Write\ Reduction = 1 - \frac{TMWL\ using\ approximate\ memory}{TMWL\ using\ precise\ memory} \quad (1)$$

When $T = 0.025$, the guard band is large and the memory can be considered precise [54]. As mentioned in Section 2.1.1, T should be less than $\frac{1}{8}$ to provide guard bands, i.e., $T < 0.125$. We vary T from 0.025 to 0.1. The cases where $T > 0.1$ are omitted because the imprecision behavior dominates the execution. Results are shown in Figure 4. The error rate denotes the proportion of elements in X whose values deviate from their original values after sorting in the approximate memory. For all the sorting algorithms, both error rate and *Rem* ratio grow rapidly when $T > 0.06$.

However, as shown in Figure 2(a), the average number of P&V iterations drops slowly when $T > 0.06$. Also, write reduction is increasing slower with larger T as shown in Figure 4(c). As a result, T should not be tuned to be arbitrarily large. These results of error and acceleration are consistent with those in the previous study [54].

To give more intuition, we take $n = 160,000$ as an example and visualize the shape of X after sorting. The results for larger input sizes (e.g., 16 million) are similar. The initial X contains $160,000$ random integers, which are uniformly distributed between 0 and $2^{32} - 1$. The goal is to sort the $160,000$ integers in increasing order. We show how X looks like after the sorting execution in the approximate memory with different precisions in Figures 5, 6, and 7, when $T = 0.03, 0.055$, and 0.01, respectively. The X-axis of each figure represents n indexes, which ranges from 1 to n. The Y-axis is the value of each index after sorting in the approximate memory. A strictly increasing line represents a fully sorted sequence.

T	Quicksort	LSD	MSD	Mergesort
0.03	0.0019%	0.0009%	0.0007%	0.0025%
0.055	1.92%	1.02%	1.00%	55.80%
0.1	96.89%	95.68%	83.82%	99.95%

Table 3: *Rem* ratio of X after quicksort, LSD, MSD and mergesort in the approximate memory

Table 3 gives the detailed *Rem* ratio of X after quicksort, LSD, MSD and mergesort in the approximate memory. When T is small, errors hardly occur and X is almost sorted as shown in Figure 5. As T increases, more errors occur. When $T = 0.055$ in Figure 6, write latency is reduced by 33% (shown in Figure 2(a)) while X is still nearly sorted after quicksort and radixsort with the *Rem* ratio lower than 2%. The remaining elements are just like noises, and the amount is relatively small. However, if T keeps increasing, errors can burst almost exponentially (as observed in Figure 4(a)). When $T = 0.1$ in Figure 7, though write latency is reduced by 50%, X is still in chaos after sorting in the approximate memory, with the high *Rem* ratio over 90% for most sorting algorithms.

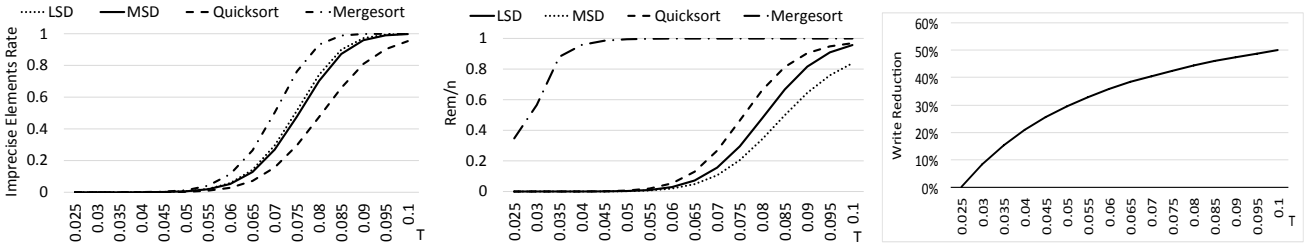
3.5 Discussions on Different Algorithms

Our study clearly shows that sorting algorithms have very different behaviors on approximate memory. As shown in Figure 4(b), Figure 6(d) and Figure 7(d), mergesort exposes different behavior from quicksort and radixsort, rendering itself vulnerable to imprecision. The execution of mergesort starts with divide-and-conquer, and the number of elements involved in a merge execution increases in later merge runs, so does the number of imprecise elements. Erroneous behaviors caught in the merge execution exacerbate in the last run, which involves the most imprecise elements, resulting in a sequence not being able to be labeled as "nearly sorted". LSD also has similar behaviors with mergesort, but not all errors in LSD matter. Unlike mergesort, errors in lower bits does not cause disorders in later runs because it only looks at specific bits in each pass. Therefore, LSD is much more tolerant to imprecision than mergesort.

In contrast, quicksort and MSD have rather smooth tradeoff between imprecision and performance. In each run of quicksort, if the picked pivot divides the sequence nearly equally, the first half should be strictly smaller than the second half. This fact makes fewer elements involved in later dividing processes, by which the detrimental effect of an imprecise element is minimized. The same reason applies to MSD.

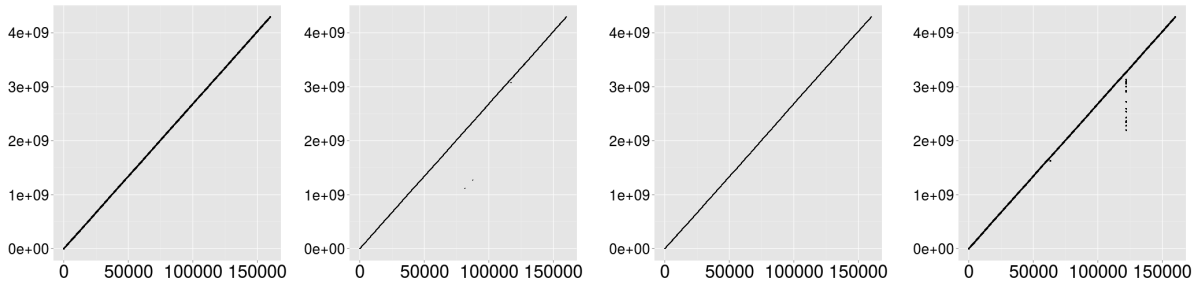
4. SORTING UNDER APPROX-REFINE

Approximate storage is designed for those applications that do not require accurate results or are tolerant of imprecision. However, in many scenarios of sorting, especially in database applications, the results are required to be precise. Therefore, unlike existing work in the area of approximate computing [15, 41, 42, 52, 54, 60], this paper intends to take advantage of approximate storage while producing pre-



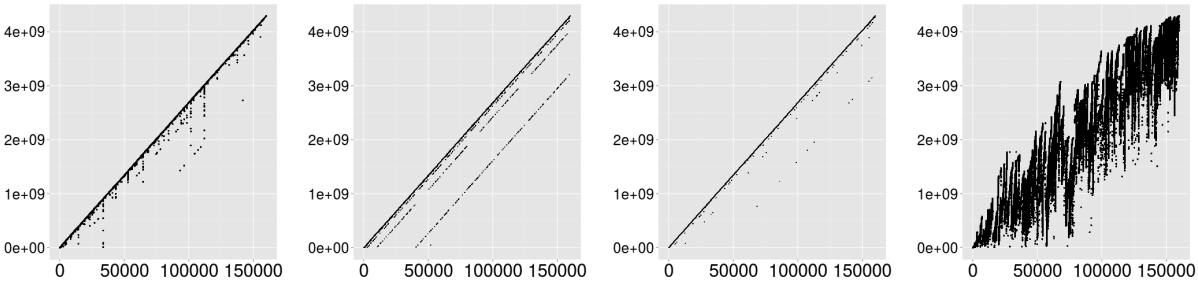
(a) Error rate (b) *Rem* ratio (c) Write reduction

Figure 4: Sorting 16M random integers in the approximate memory.



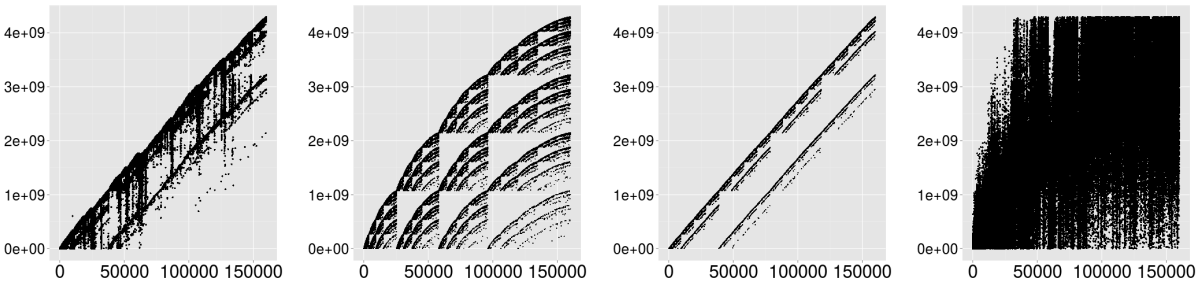
(a) Quicksort (b) LSD (c) MSD (d) Mergesort

Figure 5: Sequence X after sorting in the approximate memory when $T = 0.03$.



(a) Quicksort (b) LSD (c) MSD (d) Mergesort

Figure 6: Sequence X after sorting in the approximate memory when $T = 0.055$.



(a) Quicksort (b) LSD (c) MSD (d) Mergesort

Figure 7: Sequence X after sorting in the approximate memory when $T = 0.1$.

cise results. We propose a mechanism called *approx-refine* to refine the approximate outputs to precise results. The *approx-refine* mechanism makes use of both approximate memory and precise memory. Particularly, the previous section clearly shows the performance benefits of sorting algorithms on approximate memory, which can generate nearly sorted output in the approximate memory. We propose a lightweight refinement stage to transform the nearly sorted output to a totally sorted output. Also, the total cost of sorting on the approximate memory and the overhead caused by refinement should be smaller than that of sorting on the precise memory only. By dividing the sorting algorithm into multiple stages, we offload the main sorting algorithm to approximate memory, which is also similar to the concept of accelerator. In fact, a number of previous studies have similar designs [25, 28].

During the design and implementation, we take two special considerations into account. First, the *approx-refine* mechanism should be applicable to an arbitrary sorting algorithm on the precise memory. To the end of our design, an arbitrary sorting algorithm (e.g., three common sorting algorithms in our study) can be used as a component in our approx-refine mechanism. Second, the refinement should be lightweight and adaptive to the degree of unsortedness. Intuitively, the higher the sortedness of the output from the sorting on approximate memory, the smaller overhead the refinement stage should incur. In our design, the refinement scheme of *approx-refine* is adaptive to *Rem* (defined in Section 3).

4.1 Overview

As precise sorting is often required in applications for data management purposes where each tuple has a key along with a record ID. We assume that the input data to be sorted contains n <Key, ID> pairs, which are stored in two arrays $Key0$ and ID. The final precise output should contain a fully sorted sequence of n <Key, ID> pairs in increasing order of key values. All the inputs and outputs are stored in the precise memory. Approximate memory is only used during the sorting process.

The *approx-refine* process contains five stages, including *warm-up*, *approx preparation*, *approx stage*, *refine preparation*, and *refine stage*. We introduce the five stages as follows. A running example is given in Figure 8.

Warm-up Both $Key0$ and ID are in the precise memory initially. This study assumes that the input data is in the main memory. If the data is initially in the hard disk, we need to adopt more advanced external memory sorting algorithms [49], for which the proposed *approx-refine* scheme can be used in their in-memory sorting steps.

Approx Preparation Array $Key0$ is copied from precise memory to approximate memory (denoted as $\widetilde{Key}$). During this process, some of the keys may become imprecise (e.g. the sixth element changes from 35 to 32 in Figure 8). That means, $\widetilde{Key}$ is an imprecise version of $Key0$, and the degree of imprecision depends on the approximate memory configuration. Array ID remains in the precise memory.

Approx Stage We perform a sorting algorithm on $\widetilde{Key}$ together with the corresponding array ID. We study the three sorting algorithms described in Section 3.1: mergesort, quicksort and radix sort. The sorting algorithm we deploy in this stage is almost the same as the one in the precise memory, except for memory operations. During the sorting process, all reads and writes of keys are performed in the approximate memory, and operations on ID are in the precise memory. More keys become imprecise with more writes performed during the process. As a result, disorders are generated due to the occasional imprecision. For example, in Figure 8, $Key0_5 < Key0_6$ but $\widetilde{Key}_5 > \widetilde{Key}_6$ in the approximate memory, which results in the disorder of their corresponding IDs (ID_5 and ID_6).

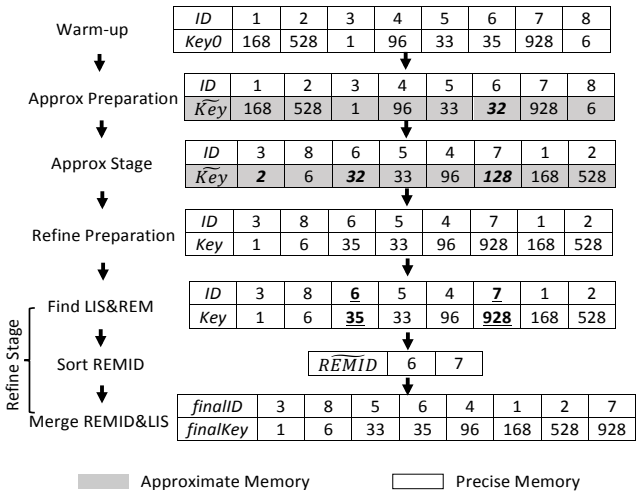


Figure 8: Overview of approx-refine for sorting. Imprecise elements are marked in bold and italic. Disorders after the approx stage are marked in bold and underlined.

Refine Preparation After the *approx* stage, we are only able to obtain a nearly sorted sequence of keys $\{Key_i\} = \{Key0_{ID_i}\}$. For example, ID_3 and ID_4 form an inversion pair because $Key_3 > Key_4$. In fact, to save more memory writes, we do not generate array Key in the *refine preparation* stage. We always make use of array $Key0$ and $\{Key_i\} = \{Key0_{ID_i}\}$ to get Key with memory reads. The notation of Key is only defined for convenience.

Refine Stage The goal of this stage is to adjust Key and ID to achieve fully sorted results in the precise memory. The increasing array of key values is denoted as $finalKey$, and the corresponding array of record IDs is denoted as $finalID$. We give the detailed design of the *refine stage* in Section 4.2. This is essentially an off-loading computing model, where the majority of the sorting algorithm is off-loaded to be performed on approximate memory in **Approx Stage**.

4.2 Refine Stage

The sequence Key is almost sorted after the *approx stage*. Our goal in the *refine stage* is to design a lightweight algorithm to refine the results with partial disorders and produce strictly sorted sequences. The algorithm should be efficient enough with as few memory write operations as possible, as writes are the dominated cost on NVRAM.

An intuitive solution is to use some adaptive sorting algorithms that benefit from the presortedness in the input sequence. Many existing adaptive sorting algorithms [20] benefiting from presortedness target at the asymptotic time complexity but not the number of memory writes. Thus,

they typically introduce $3n$ or even more memory writes (n is the number of input records). In contrast, our refine algorithm, with a lightweight and efficient heuristics, introduces fewer than $3n$ memory writes and has a low time complexity as well. Note that, intuitively, the lower bound of the number of memory writes should be $2n$, n for keys and another n for record IDs. The refine algorithm introduced below has fewer than $3n$ memory write operations, which is very close to the lower bound.

The *refine stage* is further decomposed into 3 steps.

Step 1. Find an ascending subsequence of the maximum length (also known as *Longest Increasing Subsequence (LIS)* [55]) in array Key. This step is motivated by Figure 6 and the measure of sortedness Rem introduced in Section 3.3. LIS is already sorted, while the remaining elements in Key are not.

Step 2. Sort the remaining elements in Key in increasing order. Together with LIS, we would get two sorted subsequences afterwards.

Step 3. Merge the two sorted subsequences to get the final sorted sequence.

We now present the implementation details of each step.

Step 1: Find LIS&REM. For simplicity, we denote LIS as the *Longest Increasing Subsequence* in Key, and REM as the remaining elements in Key (the size of REM is Rem). $LISID$ and $REMID$ are the corresponding record IDs of LIS and REM. To find LIS and REM, classical algorithms [55] have time complexity of at least $O(nlogn)$ and introduce at least $2n$ intermediate outputs to record states for the optimal solution (costly memory writes). To reduce the overhead of refinement to the maximum extent, we develop some heuristics to get an approximate LIS (denoted as $\widetilde{LIS}$) with time complexity of $O(n)$ and almost no intermediate memory writes. The corresponding approximate $LISID, Rem, REM$ and $REMID$ under the heuristics are denoted as $\widetilde{LISID}$, $\widetilde{Rem}$, $\widetilde{REM}$ and $\widetilde{REMID}$. The heuristics make best use of the near-sortedness of Key after the *approx stage* and are relatively easy to implement.

```
1  /*
2  Input:
3    Array ID: all the record IDs after the
         approx stage
4    Array Key0: all the original key values
         in the precise memory
5  Output:
6    Array REMID to record pairs that are not
         in LIS
7  */
8  std::vector<keyType> REMID;
9  LIStail=Key0[ID[1]]; //assume the first key
       is in LIS(X)
10 for (i=2; i<n; ++i) {
11   if (Key0[ID[i]]>=LIStail && Key0[ID[i]]<=
         Key0[ID[i+1]]) {
12   //Key[i]/Key0[ID[i]] is in approximate
         LIS as it forms a non-decreasing
         order with its two neighbors,
13     LIStail=Key0[ID[i]];
14   }
15   else {
16     REMID.push_back(ID[i]);
17   }
18 }
19 if (LIStail>Key0[ID[n]]) {
20   REMID.push_back(ID[n]);
21 }
```

Listing 1: Find LIS&REM

Listing 1 shows the pseudo code of the first step in the *refine stage*. It finds <Key, ID> pairs whose key values are not in $\widetilde{LIS}$, e.g., bold and underlined pairs in Figure 8. The heuristics to get $\widetilde{LIS}$ scan ID only once and append a record ID to $\widetilde{REMID}$ if the corresponding key value does not form an ascending order with its left and right keys. In Figure 8, the third key value 35 and the sixth key value 928 are greater than their right key values, therefore, the third and the sixth record IDs are in $\widetilde{REMID}$.

In total, it only introduces Rem intermediate data write operations in this step. Note that we only record $\widetilde{REMID}$ here to save more intermediate memory writes during the *refine stage*. Also, recording $\widetilde{REMID}$ is sufficient to get all information that we need, including $\widetilde{REM}$, $\widetilde{LIS}$, $\widetilde{LISID}$. $\widetilde{REM}$ can be easily obtained by $\{\widetilde{REM}_i\} = \{Key0_{\widetilde{REMID}_i}\}$ though it introduces an additional data read. However, as a PCM write is very much slower than a PCM read, it deserves replacing a PCM write with a PCM read. $\widetilde{LISID}$ consists of all elements in ID except for those in $\widetilde{REMID}$. $\widetilde{LIS}$ can then be obtained by $\{\widetilde{LIS}_i\} = \{Key0_{\widetilde{LISID}_i}\}$

Step 2: Sort $\widetilde{REMID}$. Since Key is almost sorted, REM is small ($|REM| << n$) and thus we do not need to further extract any sorted subsequence from REM. In this step, we directly sort $\widetilde{REMID}$ in increasing order of their key values. Since $|REM| << n$, we can choose any reasonable algorithm to sort $\widetilde{REMID}$, because it does not make any significant difference. For simplicity, we just use the sorting algorithm in the *approx stage* to sort $\widetilde{REMID}$. For example, if we apply quicksort in the *approx stage* to sort n <Key, ID> pairs, we still use quicksort to sort $\widetilde{REMID}$ with $\widetilde{Rem}$ items. If Key is nearly sorted as expected and our heuristics work well, Rem should be very small and $\widetilde{Rem} \approx Rem$. Thus, the intermediate memory writes during sorting is negligible comparing to the total write latency in the *approx stage*. However, if Key is not nearly sorted, significant overhead will be introduced to sort $\widetilde{REMID}$, and the approximate memory will not act as an accelerator.

```
1  /*
2  Input:
3    Array Key0: all the original keys in the
         precise memory
4    Array ID: all the record IDs after the
         approx stage
5    Array REMID after being sorted
6  Output:
7    Array finalKey & finalID: n <Key, ID>
         pairs in increasing order of key
         values
8  */
9
10 std::set<int> REMIDset;
11 //REMIDset is used to identify LISID later
12 for (i=0; i<REMID.size(); ++i) {
13   REMIDset.insert(REMID[i]);
14 }
15
```

```
16  LISptr=1;
17  REMptr=0;
18  finalptr=1;
19  while (LISptr<=n) {
20  //find the next element in LISID
21    while (LISptr<=n && REMIDset.count(ID[
          LISptr])>0) ++LISptr;
22    if (LISptr>n) break;
23  //merge two sorted subsequences
24    if (REMptr<REMID.size() && Key0[REMID[
          REMptr]]<Key0[ID[LISptr]]) {
25      finalID[finalptr]=REMID[REMptr];
26      finalKey[finalptr++]=Key0[REMID[REMptr
            ++]];
27    }
28    else {
29      finalID[finalptr]=ID[LISptr++];
30      finalKey[finalptr]=Key0[finalID[
            finalptr]];
31      finalptr++;
32    }
33  }
34  while (REMptr<REMID.size()) {
35    finalID[finalptr]=REMID[REMptr];
36    finalKey[finalptr++]=Key0[REMID[REMptr
          ++]];
37  }
```

Listing 2: Merge

Step 3: Merge $\widetilde{REMID}$&$\widetilde{LISID}$. Listing 2 gives the pseudo code of the final merge step in the *refine stage*. As we only have $\widetilde{REMID}$, we need to rescan array ID and find elements in $\widetilde{LISID}$. This strategy introduces more data reads but reduces data write operations. This write-limiting ideology also appears in previous study [63]. The merge step introduces $2n + \widetilde{Rem}$ data writes, among which $2n$ (n for array $finalID$ and n for array $finalKey$) are unavoidable.

4.3 Cost Analysis

Though approximate memory trades off precision for performance, the overhead of the *refine stage* offsets some of the benefits. We make a mathematical analysis to quantify the benefit that approximate memory brings under the *approx-refine* mechanism. Because memory write operations dominate the total execution time of a sorting algorithm on NVRAM [12], we analyze the total memory write latency under the *approx-refine* mechanism. We compare the results with sorting entirely in the precise memory to show the role of approximate memory.

Formally, let

$$\alpha_{alg}(n) = \#memory\ writes\ of\ keys\ using\ alg\ to\ sort\ n\ elements$$

For instance, suppose sorting is performed entirely in main memory, $\alpha_{quicksort}(n) \approx \frac{nlog_2 n}{2}$, $\alpha_{mergesort}(n) \approx nlog_2 n$. Obviously, $\alpha_{alg}(n)$ is a monotone increasing function of n.

Unlike the *write reduction* defined by Equation 1 in Section 3.4, in the case of the *approx-refine* mechanism, *write reduction* is defined as

$$Write\ Reduction = 1 - \frac{TMWL\ under\ approx\&refine}{TMWL\ using\ precise\ memory} \quad (2)$$

For simplicity, we assume that the memory write latency is a constant. Therefore, memory write latency is proportional to the number of memory write operations. Therefore, if total equivalent number of precise memory writes is abbreviated to $TEPMW$, then we define the following function to represent the write latency reduction of our proposed approx-refine scheme in comparison with the one running in the precise memory only.

$$WR_{alg}(n,t) = 1 - \frac{TEPMW\ under\ approx\&refine}{TEPMW\ under\ traditional\ sorting} \quad (3)$$

where one approximate memory operation is equivalent to $p(t)$ (defined in Section 2.2) precise memory operation, and traditional sorting is performed using precise memory only.

The total number of writes for traditional sorting in the precise memory is $2\alpha_{alg}(n)$: $\alpha_{alg}(n)$ times for key values and $\alpha_{alg}(n)$ times for record IDs.

As for the *approx-refine* mechanism, we calculate the effective number of precise memory write operations in different stages respectively.

1. In the *approx preparation* stage, n key write operations are needed to copy all the keys from precise memory to approximate memory. The equivalent number of write operations in the precise memory is $p(t)n$.
2. In the *approx* stage, $\alpha_{alg}(n)$ data write operations of keys in the approximate memory and $\alpha_{alg}(n)$ data write operations of record IDs in the precise memory are needed. In total, the equivalent number of write operations in the precise memory is $(p(t)+1)\alpha_{alg}(n)$.
3. In the *refine* stage, all the write operations are performed in the precise memory. The first step of the *refine* stage needs $\widetilde{Rem}$ data writes to record $\widetilde{REMID}$.
4. In the second step of the *refine* stage, $\alpha_{alg}(\widetilde{Rem})$ data writes are needed to sort $\widetilde{REMID}$.
5. Finally, $\widetilde{Rem} + 2n$ write operations are needed for the final merge.

In total, during the hybrid execution, $(p(t)+1)\alpha_{alg}(n) + 2\widetilde{Rem} + (2+p(t))n + \alpha_{alg}(\widetilde{Rem})$ effective precise memory writes are performed.

Finally,

$$WR_{alg}(n,t) = \frac{1-p(t)}{2} - \frac{\widetilde{Rem} + (1+0.5p(t))n}{\alpha_{alg}(n)} - \frac{\alpha_{alg}(\widetilde{Rem})}{2\alpha_{alg}(n)} \quad (4)$$

Therefore, the effect of our *approx-refine* mechanism is related to the guard band of the approximate memory, the size of dataset, the sorting algorithm, and the effects of our heuristics. To maximize WR, we definitely hope to minimize $p(t)$ and $\widetilde{Rem}$. However, given the sorting algorithm and n, $p(t)$ decreases but $\widetilde{Rem}$ increases with the increase of t. We will show the trade-off between $p(t)$ and $\widetilde{Rem}$ in the experiments of Section 5. With obtaining WR in the cost analysis, we can decide whether the approx-refine approach on the hybrid memory is better than the sorting algorithm on precise memory only, and switch between the two approaches accordingly.

5. EVALUATION OF APPROX-REFINE

In this section, we evaluate the *approx-refine* mechanism in hybrid approximate/precise memory, and compare the results with traditional sorting in the precise memory only.

We use the same methodology as the simulation studies in Section 3.2.

We focus on the impact of using approximate memory in sorting algorithms. We evaluate *write reduction* (defined by Equation 2 in Section 4.3) that quantifies the benefit of hybrid execution. As analyzed in Section 4.3, write reduction is determined by the algorithm, the input size and the precision of approximate memory. Therefore, we evaluate the influence of T and n under quicksort, mergesort, and radix sort (LSD, MSD).

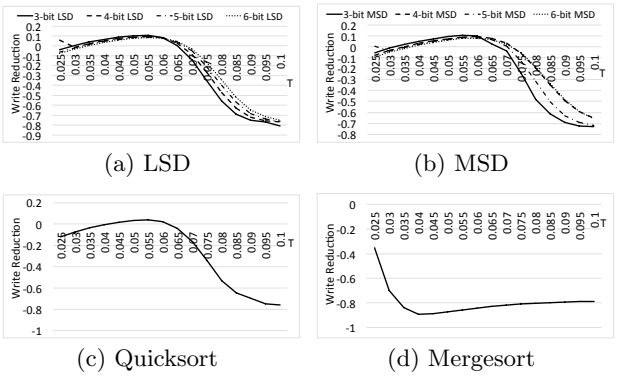
Impact of T We vary T from 0.025 to 0.1 at intervals of 0.005, and show the write reduction of all sorting algorithms with 16M (16,000,000) records in Figure 9. We have the following observations:

1. Except for mergesort, all the algorithms achieve the maximum write reduction when $T = 0.055$. Radix sort can achieve a maximal write reduction of about 10%, while quicksort achieves write reduction up to 4%. Mergesort does not show any gain using approximate memory, and we have discussed the underlying reasons in Section 3.5.

2. When $T \leq 0.03$, write reduction may be negative because as shown in Equation 4, $p(t) \approx 1$, therefore $\frac{1-p(t)}{2} \approx 0$. Since the remaining two items are always positive, it is highly possible that write reduction is negative.

3. When $T \geq 0.07$, write reduction may also become negative because the overhead of refinement is significant. Since $p(t) \approx 0$ and $\widetilde{Rem} \approx n$, we have $\frac{1-p(t)}{2} - \frac{\alpha(\widetilde{Rem})}{2\alpha(n)} \approx 0$. As the second item is always positive, the final write reduction is negative.

4. Both LSD and MSD have slightly decreasing write reduction with more bins mainly because the total write latency decreases and the fixed overhead of copying keys in *approx preparation* takes a larger fraction in the total write latency.



(a) LSD (b) MSD

(c) Quicksort (d) Mergesort

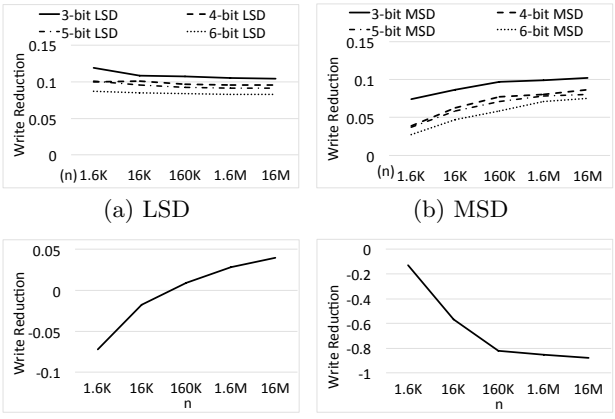
Figure 9: Relationship between write reduction and T.

Impact of n In this experiment, we set $T = 0.055$ because it produces the largest write reduction for most sorting algorithms. We change the size of input elements n from 1.6K (16000) to 16M (16,000,000). 3-bit LSD, 3-bit MSD and quicksort achieve the maximal write reduction of $11\%, 10.3\%$ and 4% respectively. From Figure 10, we see larger write reduction with large n except for LSD, which partially demonstrates the scalability of the *approx-refine* mechanism for most sorting algorithms. Equation 4 in Section 4.3 has indicated the scalability.

1. Since $\alpha_{quicksort}(n) \approx \frac{nlog_2 n}{2}$ and $\widetilde{Rem}$ is $O(n)$ (observed from experiments), given t, $WR_{quicksort}(n,t)$ is a monotone increasing function with respect to n. Thus, the performance gain of approx-refine increases for larger inputs in quicksort.

2. Since $\frac{\alpha_{MSD}(n)}{n}$ is a constant and $\widetilde{Rem}$ is also $O(n)$, given t, $WR_{MSD}(n,t)$ is a monotone increasing function with respect to n. Thus, the performance gain of approx-refine increases for larger inputs in MSD.

3. Like MSD, $\frac{\alpha_{LSD}(n)}{n}$ is also a constant. However, $\widetilde{Rem}$ is not $O(n)$ under LSD, which introduces more overhead in the *refine* stage. It shows some similar behavior of mergesort as discussed in Section 3.5. Therefore, given t, $WR_{LSD}(n,t)$ is not a monotone increasing function with respect to n.



(a) LSD (b) MSD

(c) Quicksort (d) Mergesort

Figure 10: Relationship between write reduction and n.

We set $T = 0.055$ and $n = 16M$, and show the *normalized* total write latency under the approx-refine mechanism for each algorithm in Figure 11 (normalized to the write latency of 3-bit-LSD in the *approx* stage). Write latencies are further decomposed into *approx* and *refine*. Both LSD and MSD have smaller total write latencies with more bins (6-bit achieves the best performance). 6-bit MSD radixsort and Quicksort are the most efficient sorting algorithms with the least memory write latencies. The overhead of *refine* is negligible except for merge sort.

Summary. We have the following key findings. First, approximate memory is beneficial for sorting algorithms that require precise output, with up to 11% of write reduction in our experiments. The improvement is achieved through the proposed approx-refine scheme, i.e., taking advantage of approximate memory as well as a carefully designed and lightweight refinement stage. Second, sorting algorithms can

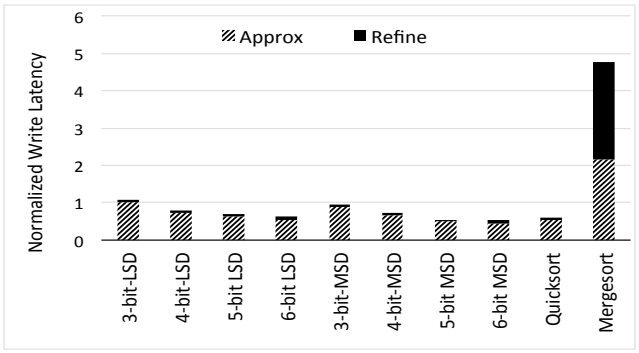


Figure 11: Breakdown of write latencies in *approx* and *refine* stages.

have very different performance behaviours. Radix sort and quick sort generally acquire reasonable performance gain whereas using approximate hardware has negative impact on merge sort.

The performance study can be extended in a couple of ways. First, we have tested the impact of some of the more advanced techniques including multithreading, memory prefetching and non-temporal streaming instructions in our implementations. We observed insignificant impacts of these factors. Second, we have extended and evaluated our approx-refine proposal in another approximate memory model [51] for energy saving. The preliminary results demonstrate the energy saving by 13.4%. More details can be found in Appendix A. Third, since approx-refine has the flexibility of allowing us to explore other sorting implementations, we have evaluated an open-source implementation of radixsort [45] which has used SIMD instructions extensively and histogram-based implementation. The preliminary results demonstrate the write latency reduction of around 10%. More details can be found in Appendix B. Finally, we are further extending a more complete memory model for approximate memory. Currently, we assume that the performance of random writes is the same as that of sequential writes. A more detailed model of PCM should capture the performance difference between random and sequential writes. In the *approx* stage, most write operations of the studied algorithms are random writes on PCM. However, in the *refine* stage, most writes are instead sequential writes. With a more detailed model of PCM, the approx-refine scheme should receive a higher gain of write reduction.

6. RELATED WORK

We review the related work in the following categories: approximate computing, sorting, and hybrid storage systems.

6.1 Approximate Computing

Approximate computing has attracted a lot of research attention. A large body of previous work has tackled approximate computing from various aspects: from architecture and hardware design, software and programming language support, model and tolerance analysis to approximate processing in databases. We refer the readers to a recent survey on approximate computing [26].

Architecture and Hardware Design. Approximate hardware design has been a hot research topic, due to the limited scaling of hardware. Mohapatra et al. [40] presented an algorithm/architecture co-design of voltage-scalable, process variation aware motion estimator. It leverages a fact in a video system that, not all computations are equally significant. Esmaeilzadeh et al. [18] proposed efficient mapping of disciplined approximate programming onto hardware. Venkataramani et al. [62] proposed a design of approximate circuits, an embodiment of quality programmable processor architecture [60], and a programmable and quality-configurable neuromorphic processing engine to execute approximate neural networks [61]. Sampson et al. [54] proposed approximate storage for non-volatile memory. Rahimi et al. [48], Lucas et al. [36], Song et al. [35] and Ranjan et al. [51] explored the energy efficient approximate storage in the context of different memory architectures (such as associative memristive memory, DRAM, and spintronic memory). We conjecture that our approx-refine scheme can be applicable to those approximate memory designs. This study adopts the approximate memory design from PCM [54]. Also, we focus on how to leverage the approximate memory to improve the performance of sorting algorithms to produce precise results.

Software and Programming Language Support. A number of software and programming systems have been proposed to exploit the tradeoff on precision and performance/energy consumption of approximate hardware. Green [3] is a simple and flexible framework to trade off QoS and energy consumption with controlled approximation. Esmaeilzadeh et al. [19] defined a programming model which helps offload approximable code regions to *neural processing unit* to trade precision for improved performance. Enerj [53] is proposed to isolate the approximate part from precise executions using type qualifiers. ApproxIt [68] is a lightweight quality estimator which dynamically scales the approximation quality over successive iterations. Still, most of those studies focus on applications with intrinsic tolerance to inaccuracy in computation.

Model and tolerance analysis. Models and tools have been developed to analyze the resilience of an application to imprecision. Nair et al.[41, 42] outlined the model and technique of approximate computing. Chippa et al. [15] proposed a framework to characterize the resilience of applications. ASAC [52] automatically generates annotations by sensitivity analysis to allow approximation.

Approximate processing in databases. Approximate query processing in databases [1, 11, 24, 56, 2] and sensor networks [58, 59] mainly leverage pre-computed synopses, sampling engines, or temporal and spatial localities to produce approximate answers within a certain error bar. Those studies do not leverage approximate hardware.

6.2 Sorting

The design and implementation of sorting algorithms are heavily impacted by architectures. They have been revisited in parallel architectures such as GPUs [25] and FPGAs [33], and emerging memories such as PCM [63] and Flash [65]. On PCM, write performance is a key problem for sorting, which involves a lot of memory writes [63]. The refinement stage in this study is write-optimized, which reduces writes with reasonably more reads in the design.

Another category of related work to our refinement stage is adaptive sorting algorithms [9, 39]. Many adaptive algorithms have been developed for nearsortedness [20, 9, 39]. However, they are mostly algorithms with theoretically good

asymptotic time complexity. They are far more complicated to implement in practice, and are not optimized with writes. Thus, we use simple and lightweight heuristics to leverage the nearsortedness.

6.3 NVRAM and Hybrid Memory Systems

Previous papers have explored PCM as main memory [29, 34]. Recently, researchers have been revisiting database algorithms to better utilize PCM. Chen et al. [12] calibrated B^+-tree and join algorithms to favor PCM. Viglas et al. [63] proposed a write-limited notion to reduce write operations of sorts and joins. Chi et al. [14] also proposed to reduce write operations of $B+$-tree implementation. Data consistency problems have been addressed for index structures [66, 13]. New on-line transaction processing engines have been redesigned on NVRAM (e.g., [64, 22]). This paper extends the scope of PCM to approximate PCM, and proposes an efficient and general mechanism to provide precise sorting executions. Our approach is orthogonal to those previous studies, because approximate memory improves the performance by reducing programming iterations of each write operation.

Previously proposed hybrid storage often integrates two different memories such as PCM/DRAM [31, 50, 21], on/off-package DRAM [17], and LPDRAM/RLDRAM [10]. Our hybrid mechanism integrates the same material with different precision configuration, only by varying the guard band range. Memory modules needed by our mechanism are exactly the same in terms of material.

7. CONCLUSIONS

Approximate hardware has become an emerging research dimension in computer architecture for performance and energy consumption. Yet, most existing studies on approximate computing concentrate on applications with intrinsic tolerance to inaccuracy in computation. This paper argues that approximate hardware can also be used for improving the performance of applications with precise output requirement. Specifically, we exploit approximate memory to reduce write latencies for sorting algorithms on PCMs. We study how the precision of PCM affects the sortedness of the final output and propose a novel *approx-refine* mechanism to produce precise results. This mechanism first sorts in approximate storage to get nearly sorted sequences, and then reorders the sequence in precise storage with a lightweight recovery heuristics. We have studied three common algorithms including quicksort, merge sort and radix sort. Simulation results show that the total write latency is reduced by up to 11%. Sorting algorithms can have very different performance behaviors on approximate hardware. Radix sort and quick sort generally acquire reasonable performance gain whereas using approximate hardware has negative impact on merge sort. Therefore, we conclude that approximate memory is able to unleash the memory bound of sorting algorithms, and has the potential to be a special accelerator for sorting algorithms. Though system interfaces should be carefully redesigned to support hybrid approximate/precise main memory, the modification in hardware is lightweight and easy to implement.

As for future work, the *approx-refine* scheme has raised an interesting design strategy for leveraging approximate hardware for database systems. This paper makes the first and important step of utilizing approximate hardware for database operations. More research efforts have to be made in the future, e.g., other database operations (such as aggregations) on approximate hardware and reducing the overhead of refinement stage.

8. ACKNOWLEDGMENTS

The work of Shunning Jiang and Shuang Chen was done when they were visiting students at Nanyang Technological University, Singapore. This work is supported by Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2012-T2-2-067, and Academic Research Fund Tier 1 under Grant 2013-T1-002-123.

9. REFERENCES

[1] 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*, SIGMOD '14, pages 481–492. ACM, 2014.

[2] 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*, pages 29–42. ACM, 2013.

[3] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[5] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. Calvi, et al. A bipolar-selected phase change memory featuring multi-level cell storage. *Solid-State Circuits, IEEE Journal of*, 44(1):217–227, 2009.

[6] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *J. Exp. Algorithmics*, 12:2.2:1–2.2:23, June 2008.

[7] A. Cabrini, S. Braga, A. Manetto, and G. Torelli. Voltage-driven multilevel programming in phase change memories. In *Memory Technology, Design, and Testing, 2009. MTDT'09. IEEE International Workshop on*, pages 3–6. IEEE, 2009.

[8] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 521–526. IEEE, 2012.

[9] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, 1993.

[10] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *Microarchitecture (MICRO),*

2012 45th Annual IEEE/ACM International Symposium on, pages 13–24. IEEE, 2012.

[11] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM SIGMOD Record*, volume 30, pages 295–306. ACM, 2001.

[12] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31, 2011.

[13] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[14] P. Chi, W.-C. Lee, and Y. Xie. Making b+-tree efficient in pcm-based main memory. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 69–74. ACM, 2014.

[15] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

[16] X. Dong and Y. Xie. Adams: Adaptive mlc/slc phase-change memory design for file storage. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 31–36. IEEE, 2011.

[17] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, volume 47, pages 301–312. ACM, 2012.

[19] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[20] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, Dec. 1992.

[21] S. Gao, B. He, and J. Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 263–272, New York, NY, USA, 2015. ACM.

[22] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. Pcmlogging: Reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2401–2404, New York, NY, USA, 2011. ACM.

[23] P. Garcia. Multithreaded architectures and the sort benchmark. In *Proceedings of the 1st international workshop on Data management on new hardware*, page 1. ACM, 2005.

[24] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

[25] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM, 2006.

[26] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*, pages 1–6, May 2013.

[27] B. He. When data management systems meet approximate hardware: Challenges and opportunities. *Proceedings of the VLDB Endowment*, 7(10), 2014.

[28] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[29] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 3–14. ACM, 2010.

[30] 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*, pages 1–10. IEEE, 2012.

[31] D. Kim, S. Lee, J. Chung, D. H. Kim, D. H. Woo, S. Yoo, and S. Lee. Hybrid dram/pram-based main memory for single-chip cpu/gpu. In *Proceedings of the 49th Annual Design Automation Conference*, pages 888–896. ACM, 2012.

[32] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[33] D. Koch and J. Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[34] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[35] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. *ACM SIGPLAN Notices*, 47(4):213–224, 2012.

[36] J. Lucas, M. Alvarez-Mesa, M. Andersch, and B. Juurlink. Sparkk: Quality-scalable approximate storage in dram. 2014.

[37] Micron. MLC NAND Flash. http://www.micron.com/products/nand-flash/mlc-nand, 2015.

[38] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. Bit error rate in nand flash memories. In

Reliability Physics Symposium, 2008. IRPS 2008. IEEE International, pages 9–19. IEEE, 2008.

[39] A. Moffat, G. Eddy, and O. Petersson. Splaysort: Fast, versatile, practical. *Software: Practice and Experience*, 26(7):781–797, 1996.

[40] D. Mohapatra, G. Karakonstantis, and K. Roy. Significance driven computation: a voltage-scalable, variation-aware, quality-tuning motion estimator. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 195–200. ACM, 2009.

[41] R. Nair. Models for energy-efficient approximate computing. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 359–360. ACM, 2010.

[42] R. Nair. Big data needs approximate computing: Technical perspective. *Commun. ACM*, 58(1), Dec. 2014.

[43] T. Nirschl, J. Philipp, T. Happ, G. Burr, B. Rajendran, M. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, et al. Write strategies for 2 and 4-bit multi-level phase-change memory. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 461–464. IEEE, 2007.

[44] A. Pantazi, A. Sebastian, N. Papandreou, M. Breitwisch, C. Lam, H. Pozidis, and E. Eleftheriou. Multilevel phase change memory modeling and experimental characterization. *Proceedings of EPCOS*, 2009.

[45] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 755–766. ACM, 2014.

[46] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 153–162. ACM, 2010.

[47] 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*, 37(3):24–33, 2009.

[48] A. Rahimi, A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta. Approximate associative memristive memory for energy-efficient gpus. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1497–1502. EDA Consortium, 2015.

[49] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[50] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[51] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan. Approximate storage for energy efficient spintronic memories. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[52] P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac: automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 95–104. ACM, 2014.

[53] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[54] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36. ACM, 2013.

[55] C. Schensted. Longest increasing and decreasing subsequences. In *Classic Papers in Combinatorics*, pages 299–311. Springer, 1987.

[56] J. Spiegel and N. Polyzotis. Tug synopses for approximate query answering. *ACM Transactions on Database Systems (TODS)*, 34(1):3, 2009.

[57] K. Takeuchi, T. Tanaka, and T. Tanzawa. A multipage cell architecture for high-speed programming multilevel nand flash memories. *Solid-State Circuits, IEEE Journal of*, 33(8):1228–1238, 1998.

[58] X. Tang and J. Xu. Adaptive data collection strategies for lifetime-constrained wireless sensor networks. *Parallel and Distributed Systems, IEEE Transactions on*, 19(6):721–734, 2008.

[59] X. Tang and J. Xu. Optimizing lifetime for continuous data aggregation with precision guarantees in wireless sensor networks. *IEEE/ACM Transactions on Networking (TON)*, 16(4):904–917, 2008.

[60] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. ACM, 2013.

[61] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 27–32. ACM, 2014.

[62] S. Venkataramani, K. Roy, and A. Raghunathan. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1367–1372. EDA Consortium, 2013.

[63] S. D. Viglas. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow*, 2014.

[64] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[65] C.-H. Wu. Data sorting in flash memory. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1847–1849, New York, NY, USA, 2012. ACM.

[66] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[67] S. Yeo, N. H. Seong, and H.-H. S. Lee. Can multi-level cell pcm be reliable and usable? analyzing the impact of resistance drift. In *the 10th Ann. Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2012.

[68] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

APPENDIX

A. EVALUATION WITH ANOTHER APPROXIMATE STORAGE

To demonstrate that our *approx-refine* mechanism can be applied to other approximate memory models, we make a preliminary study by using the model from Ranjan et al [51]. In the following, we briefly introduce the memory model, followed by the preliminary experimental results.

Approximate Spintronic Memory Model: Spintronic memory has great potential as future on-chip memories, which is a kind of NVRAM. Ranjan et al. [51] proposed an approximate model of spintronic memory to improve its energy efficiency. By adjusting the voltage and/or current of the magnetic tunnel junction (MTJ), they can achieve reduced read and write energy, resulting in an increased probability of read and write errors. As the energy consumption of a write operation can be over an order of magnitude higher than that of a read operation in spintronic memories, we mainly focus on the energy saving on writes by adopting the approximate memory model, and assume that reads are always precise for simplicity. Particularly, we revisit the three sorting algorithms on the approximate design of spintronic memories [51], and study the energy saving of our proposed approx-refine scheme.

Experimental Results: We explore four different configurations on the tradeoff between energy saving and precision in the approximate spintronic memory model. For each memory write operation, the energy consumption saving per write on the approximate memory is 5%, 20%, 33% and 50% of the original write energy in precise spintronic memory, with about 10^{-7}, 10^{-6}, 10^{-5} and 10^{-4} write error probability per bit respectively. The higher energy consumption saving per write, the higher error probability on the approximate memory.

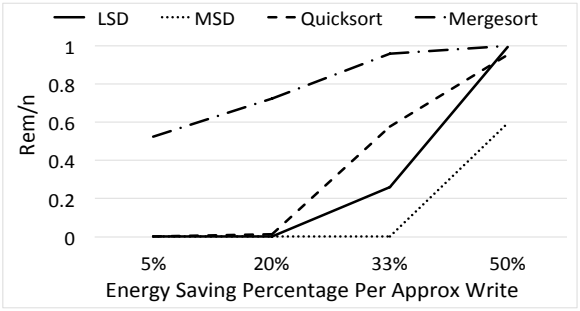
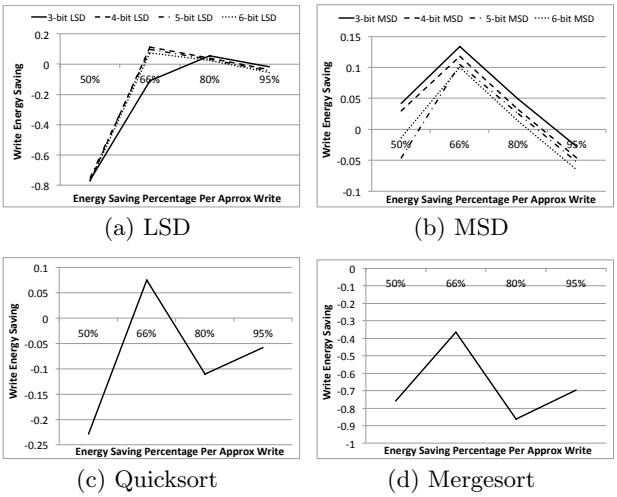


Figure 12: *Rem* ratio under the approximate memory model [51].

With $16M$ random 32-bit integers as an input, Figure 12 shows the change of *Rem* ratio with the percentage of write energy saving after quicksort, 6-bit LSD, 6-bit MSD and mergesort. When saving only 5% energy per write, errors are rare and the input sequence is nearly sorted. However, when saving 50% energy per write (error rate per bit is about 10^{-4}), the input sequence is still almost random after sorting in approximate memory. This observation is similar to that in Section 3.

Figure 13 shows the total write energy saving under different write error probabilities after applying our *approx-refine* mechanism in the approximate spintronic memory. We set



(a) LSD (b) MSD

(c) Quicksort (d) Mergesort

Figure 13: Write energy reduction in comparison with the precise memory model.

$n = 16M$. Except for mergesort, all the other three algorithms achieve better write energy consumption when the energy consumption saving of a write operation on the approximate model is 20% or 33% of the energy consumption per write on the precise model. Radix sort can achieve a maximal write energy saving of about 13.4%, while quicksort achieves write energy reduction up to 7.5%.

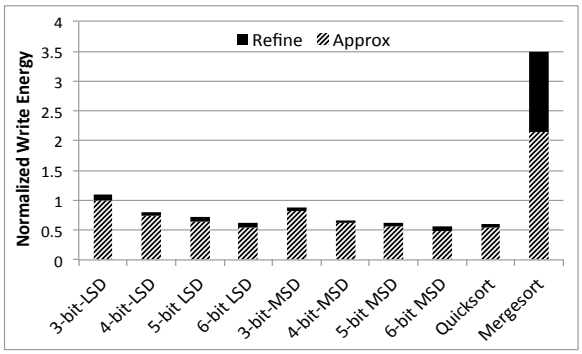


Figure 14: Breakdown of write energy in *approx* and *refine* stages.

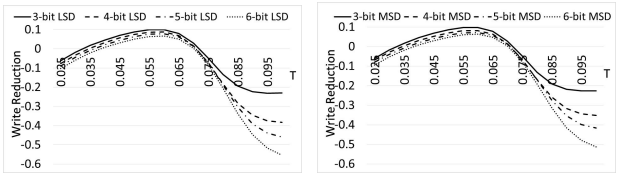
We set the write energy saving per write in approximate spintronic memory as 33% of the precise write energy and $n = 16M$. We show the normalized total write energy consumption under the approx-refine mechanism for each algorithm in Figure 14 (normalized to the write energy of 3-bit-LSD in the *approx* stage).

Write energy is further decomposed into *approx* and *refine*. We can see that the energy consumption of the *refine* stage is mostly negligible except for merge sort.

Summary: This study shows that our approx-refine scheme can achieve up to 13.4% energy saving on the approximate spintronic memory model [51]. Through this study, we demonstrate that our *approx-refine* mechanism is not restricted to any specific model, but can be generally applied to many other approximate memory model.

B. EVALUATION WITH ANOTHER SORT IMPLEMENTATION

To demonstrate that our *approx-refine* mechanism is applicable to other sorting implementation, we make a preliminary study by using an open-source implementation of radix sort by Polychronious et al. [45]. The implementation includes both LSD and MSD, which uses SIMD instruction heavily and also includes a histogram-based scheme for better memory locality. We use the same methodology as presented in Section 3.2.



(a) LSD (b) MSD

Figure 15: Write Reduction of LSD and MSD of Polychronious's implementation [45] with approximate memory model from Sampson et al [53].

We change T from 0.025 to 0.1 and plot the write reduction of 3-bit to 6-bit LSD and MSD with $n = 16M$ in Figure 15. We have the following observations:

1. All of these algorithms achieve the most write reduction when $T = 0.055$ or $T = 0.06$. This is consistent with the results of our LSD and MSD implementations.
2. 3-bit LSD and MSD can achieve around 10% write reduction. 6-bit LSD and MSD can achieve around 5% write reduction. The gain is slightly smaller than the study of LSD and MSD using our own implementations. We further investigate this performance difference. This is due to different implementations of radix-sorts during each pass. The histogram-based scheme in this implementation further reduces the number of writes. Therefore, with the same overhead of the *refine* stage, there is smaller write reduction in this implementation. Note that, we have also tested the implementation by enabling/disabling the SIMD and NUMA options and get almost the same write reductions.