Approximate Storage in Solid-State Memories

Adrian Sampson
University of Washington
asampson@cs.washington.edu

Jacob Nelson
University of Washington
nelson@cs.washington.edu

Karin Strauss
Microsoft Research
kstrauss@microsoft.com

Luis Ceze
University of Washington
luisceze@cs.washington.edu

ABSTRACT

Memories today expose an all-or-nothing correctness model that incurs significant costs in performance, energy, area, and design complexity. But not all applications need high-precision storage for all of their data structures all of the time. This paper proposes mechanisms that enable applications to store data approximately and shows that doing so can improve the performance, lifetime, or density of solid-state memories. We propose two mechanisms. The first allows errors in multi-level cells by reducing the number of programming pulses used to write them. The second mechanism mitigates wear-out failures and extends memory endurance by mapping approximate data onto blocks that have exhausted their hardware error correction resources. Simulations show that reduced-precision writes in multi-level phase-change memory cells can be 1.7× faster on average and using failed blocks can improve array lifetime by 23% on average with quality loss under 10%.

Categories and Subject Descriptors

B.3.4 [**Memory Structures**]: Reliability, Testing, and Fault Tolerance; D.4.2 [**Operating Systems**]: Storage Management—*main memory, secondary storage*

General Terms

Reliability, Performance

Keywords

Approximate computing, storage, error tolerance, phase-change memory

1. INTRODUCTION

Many common applications have intrinsic tolerance to inaccuracies in computation. Techniques under the umbrella of *approximate computing* [2, 8, 12, 14, 22, 24, 26, 38, 41] exploit this tolerance to trade off accuracy for performance or energy efficiency. Applications in domains like computer vision, media processing, machine learning, and sensor data analysis can see large efficiency gains in exchange for small compromises on computational accuracy. This trade-off extends to storage: tolerance to errors in both transient and persistent data is present in a wide range of software, from servers to mobile devices.

Meanwhile, the semiconductor industry is beginning to encounter limits to further scaling of common memory technologies like DRAM and Flash. As a result, new memory technologies and techniques are emerging. Multi-level cells, which pack more than one bit of information in a single cell, are already commonplace and phase-change memory (PCM) is imminent. But both PCM and Flash wear out over time as cells become stuck. Furthermore, multi-level cells are slower to write due to the need for tightly controlled iterative programming.

Memories traditionally address wear-out issues and implement multi-level cell operation in ways that ensure perfect data integrity 100% of the time. This has significant costs in performance, energy, area, and complexity. These costs are exacerbated as memories move to smaller device feature sizes along with more process variation. By relaxing the requirement for perfectly precise storage—and exploiting the inherent error tolerance of approximate applications—failure-prone and multi-level memories can gain back performance, energy, and capacity.

In this paper, we propose techniques that exploit data accuracy trade-offs to provide *approximate storage*. In essence, we advocate exposing storage errors up to the application with the goal of making data storage more efficient. We make this safe by: (1) exploiting application-level inherent tolerance to inaccuracies; and (2) providing an interface that lets the application control which pieces of data can be subject to inaccuracies while offering error-free operation for the rest of the data. We propose two basic techniques. The first technique uses multi-level cells in a way that enables higher density or better performance at the cost of occasional inaccurate data retrieval. The second technique uses blocks with failed bits to store approximate data; to mitigate the effect of failed bits on overall value precision, we prioritize the correction of higher-order bits.

Approximate storage applies to both persistent storage (files or databases) as well as transient data stored in main memory. We explore the techniques in the context of PCM, which may be used for persistent storage (replacing hard disks) or as main memory (replacing DRAM) [21,36,47], but the techniques generalize to other technologies such as Flash. We simulate main-memory benchmarks and persistent-storage datasets and find that our techniques improve write latencies by 1.7× or extend device lifetime by 23% on average while trading off less than 10% of each application's output quality.

We begin by describing the programming models and hardware–software interfaces we assume for main-memory and persistent approximate storage. Next, Sections 3 and 4 describe our two approximate-storage techniques in detail. Section 5 describes our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'46, December 7–11, 2013, Davis, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2638-4/13/12…$15.00.

evaluation of the techniques using a variety of error-tolerant benchmarks and Section 6 gives the results for these experiments. Finally, we enumerate related work on storage and approximate computing and conclude.

2. INTERFACES FOR APPROXIMATE STORAGE

Approximate computing lets programs eschew traditional guarantees of perfect precision when they are unnecessary. If a large portion of a program's work is error-tolerant, the hardware can operate more efficiently [8, 12, 14, 24, 26]. While previous work has considered reducing the energy spent on DRAM and SRAM storage [14, 24, 38], modern non-volatile memory technologies also exhibit properties that make them candidates for storing data approximately. By exploiting the synergy between these properties and application-level error tolerance, we can alleviate some of these technologies' limitations: limited device lifetime, low density, and slow writes.

Approximate storage augments memory modules with software-visible precision modes. When an application needs strict data fidelity, it uses traditional *precise* storage; the memory then guarantees a low error rate when recovering the data. When the application can tolerate occasional errors in some data, it uses the memory's *approximate* mode, in which data recovery errors may occur with non-negligible probability.

This work examines the potential for approximate storage in PCM and other solid-state, non-volatile memories. These technologies have the potential to provide persistent mass storage and to serve as main memory. For either use case, the application must determine which data can tolerate errors and which data needs "perfect" fidelity. As with previous work that exposes approximation to software, we assume that programmers need to make this decision—approximating data indiscriminately can lead to broken systems [2, 12, 14, 15, 38].

The next sections describe the approximation-aware programming models for main memory and persistent mass storage along with the hardware–software interface features common to both settings. In general, each block (of some appropriate granularity) is logically in either *precise* or *approximate* state at any given time. Every read and write operation specifies whether the access is approximate or precise. These per-request precision flags allow the storage array to avoid the overhead of storing per-block metadata. The compiler and runtime are responsible for keeping track of which locations hold approximate data. Additionally, the interface may also allow software to convey the relative importance of bits within a block, enabling more significant bits to be stored with higher accuracy.

Prior work on approximate computing has suggested that different applications can tolerate different error rates [14, 38] and our experimental results (Section 6) corroborate this. However, rather than providing interfaces to control precision levels at a fine grain, we apply a uniform policy to all approximate data in the program. Empirically, we find that this level of control is sufficient to achieve good quality trade-offs for a variety of applications.

2.1 Approximate Main Memory

PCM and other fast, resistive storage technologies may be used as main memories [21, 36, 47]. Previous work on approximate computing has examined applications with error-tolerant memory in the context of approximate DRAM and on-chip SRAM [14, 24, 38]. This work has found that a wide variety of applications, from image processing to scientific computing, have large amounts of error-tolerant stack and heap data. We extend the programming model and hardware–software interfaces developed by this previous work for our approximate storage techniques.

Programs specify data elements' precision at the programming language level using EnerJ's type annotation system [38]. Using these types, the compiler can statically determine whether each memory access is approximate or precise. Accordingly, it emits load and store instructions with a precision flag as in the Truffle ISA [14].

2.2 Approximate Persistent Storage

While DRAM and SRAM are typically limited to transient storage, both Flash and emerging technologies like PCM can be used for persistent, disk-like storage. Interfaces to persistent storage include filesystems, database management systems (DBMSs), or, more recently, flat address spaces [10, 45].

Use cases for approximate mass storage range from server to mobile and embedded settings. A data-center-scale image or video search database, for example, requires vast amounts of fast persistent storage. If occasional pixel errors are acceptable, approximate storage can reduce costs by increasing the capacity and lifetime of each storage module while improving performance and energy efficiency. On a mobile device, a context-aware application may need to log many days of sensor readings to model user behavior. Here, approximate storage can help relieve capacity constraints or, by reducing the cost of accesses, conserve battery life.

We assume a storage interface resembling a key–value store or a flat address space with smart pointers (e.g., NV-heaps [10] or Mnemosyne [45]), although the design also applies to more complex interfaces like filesystems and relational databases. Each object in the store is either approximate or precise. The precision level is set when the object is created (and space is allocated).

2.3 Hardware Interface and Allocation

In both deployment scenarios, the interface to approximate memory consists of read and write operations augmented with a precision flag. In the main-memory case, these operations are load and store instructions (resembling Truffle's `stl.a` and `ldl.a` [14]). In the persistent storage case, these are blockwise read and write requests.

The memory interface specifies a granularity at which approximation is controlled. In PCM, for example, this granularity may be a 512-bit block. The compiler and allocator ensure that precise data is always stored in precise blocks. (It is safe to conservatively store approximate data in precise blocks.)

To maintain this property, the allocator uses two mechanisms depending on whether the memory supports software control over approximation. With software control, as in Section 3, the program sets the precision state of each block implicitly via the flags on write instructions. (Reads do not affect the precision state.) In a hardware-controlled setting, as in Section 4, the operating system maintains a list of approximate blocks reported by the hardware. The allocator consults this list when reserving space for new objects. Section 4.2 describes this OS interface in more detail.

In the main memory case and when using object-based persistent stores like NV-heaps [10], objects may consist of interleaved precise and approximate data. To support mixed-precision objects, the memory allocator must lay out fields across precise and approximate blocks. To accomplish this, the allocator can use one of two possible policies: *ordered layout* or *forwarding pointers*. In ordered layout, heterogeneous objects lay out their precise fields (and object header) first; approximate fields appear at the end of the object. When an object's range of bytes crosses one or more block boundaries, the blocks that only contain approximate fields may be marked as approximate. The prefix of blocks that contain at least one precise byte must conservatively remain precise. With forwarding pointers,

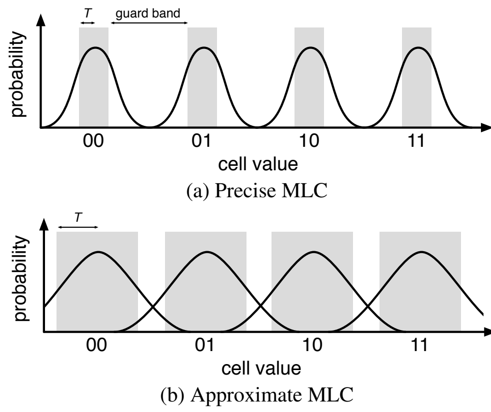


Figure 1: The range of analog values in a precise (a) and approximate (b) four-level cell. The shaded areas are the target regions for writes to each level (the parameter T is half the width of a target region). Unshaded areas are *guard bands*. The curves show the probability of reading a given analog value after writing one of the levels. Approximate MLCs decrease guard bands so the probability distributions overlap.

in contrast, objects are always stored in precise memory but contain a pointer to approximate memory where the approximate fields are stored. This approach incurs an extra memory indirection and the space overhead of a single pointer per heterogeneous object but can reduce fragmentation for small objects.

To specify the relative priority of bits within a block, accesses can also include a data element size. The block is then assumed to contain a homogenous array of values of this size; in each element, the highest-order bits are most important. For example, if a program stores an array of double-precision floating point numbers in a block, it can specify a data element size of 8 bytes. The memory will prioritize the precision of each number's sign bit and exponent over its mantissa in decreasing bit order. Bit priority helps the memory decide where to expend its error protection resources to minimize the magnitude of errors when they occur.

3. APPROXIMATE MULTI-LEVEL CELLS

PCM and other solid-state memories work by storing an analog value—resistance, in PCM's case—and quantizing it to expose digital storage. In multi-level cell (MLC) configurations, each cell stores multiple bits. For *precise* storage in MLC memory, there is a trade-off between access cost and density: a larger number of levels per cell requires more time and energy to access. Furthermore, protections against analog sources of error like drift can consume significant error correction overhead [30]. But, where perfect storage fidelity is not required, performance and density can be improved beyond what is possible under strict precision constraints.

An *approximate MLC* configuration relaxes the strict precision constraints on iterative MLC writes to improve their performance and energy efficiency. Correspondingly, approximate MLC writes allow for denser cells under fixed energy or performance budgets. Since PCM's write speed is expected to be substantially slower than DRAM's, accelerating writes is critical to realizing PCM as a main-memory technology [21]. Reducing the energy spent on writes conserves battery power in mobile devices, where solid-state storage is commonplace.

Our approach to approximate MLC memory exploits the underlying analog medium used to implement digital storage. Analog reads and writes are inherently imprecise, so MLCs must incorporate *guard bands* that account for this imprecision and prevent storage

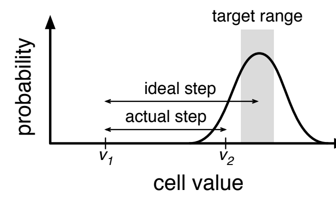


Figure 2: A single step in an iterative program-and-verify write. The value starts at v_1 and takes a step. The curve shows the probability distribution from which the ending value, v_2, is drawn. Here, since v_2 lies outside the target range, another step must be taken.

errors. These guard bands lead to tighter tolerances on target values, which in turn limit the achievable write performance. Approximate MLCs reduce or eliminate guard bands to speed up iterative writes at the cost of occasional errors. Figure 1 illustrates this idea.

3.1 Multi-Level Cell Model

The basis for MLC storage is an underlying analog value (e.g., resistance for PCM or charge for Flash). We consider this value to be continuous: while the memory quantizes the value to expose digital storage externally, the internal value is conceptually a real number between 0 and 1.[1] To implement digital storage, the cell has n discrete *levels*, which are internal analog-domain values corresponding to external digital-domain values. As a simplification, we assume that the levels are evenly distributed so that each level is the center of an equally-sized, non-overlapping band of values: the first level is $\frac{1}{2n}$, the second is $\frac{3}{2n}$, and so on. In practice, values can be distributed exponentially, rather than linearly, in a cell's resistance range [3,29]; in this case, the abstract value space corresponds to the logarithm of the resistance. A cell with $n = 2$ levels is traditionally called a single-level cell (SLC) and any design with $n > 2$ levels is a multi-level cell (MLC).

Writes and reads to the analog substrate are imperfect. A write pulse, rather than adjusting the resistance by a precise amount, changes it according to a probability distribution. During reads, material nondeterminism causes the recovered value to differ slightly from the value originally stored and, over time, the stored value can change due to drift [46]. Traditional (fully precise) cells are designed to minimize the likelihood that write imprecision, read noise, or drift cause storage errors in the digital domain. That is, given any digital value, a write followed by a read recovers the same digital value with very high probability. In approximate storage, the goal is to increase density or performance at the cost of occasional digital-domain storage errors.

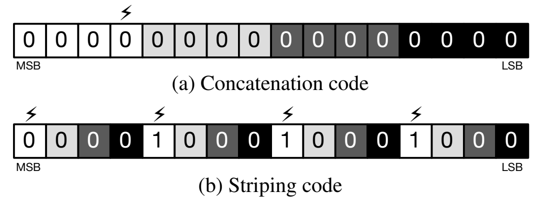
Put more formally, let v be a cell's internal analog value. A write operation for a digital value d first determines l_d, the value level corresponding to d. Ideally, the write operation would set $v = l_d$ precisely. Realistically, it sets v to $w(l_d)$ where w is an error function introducing perturbations from the ideal analog value. Similarly, a read operation recovers a perturbed analog value $r(v)$ and quantizes it to obtain a digital output.

The number of levels, n, and the access error functions, w and r, determine the performance, density, and reliability of the cell. Current designs trade off performance for density—a dense cell with many levels requires tighter error functions and is thus typically slower than sparser cells. Approximate storage cells trade off the third dimension, reliability, to gain in performance, density, or both.

[1] At small feature sizes, quantum effects may cause values to appear discrete rather than continuous. This paper does not consider these effects.

```
def w(v_t):
  v = 0
  while |v_t − r(v)| > T:
    size = v_t − v
    v += N(size, P · size)
  return v
```

Figure 3: Pseudocode for the write error function, w, in PCM cells. Here, $N(\mu, \sigma^2)$ is a normally distributed random variable with average μ and variance σ^2. The parameter T controls the termination criterion and P reflects the precision of each write pulse.



(a) Concatenation code

(b) Striping code

Figure 4: Two codes for storing 16-bit numbers in four 4-bit cells. Each color indicates a different cell. A single-level error leads to a bit flip in the indicated position. In (a), this is the lowest-order bit in the white cell. In (b), the white cell holds the binary value 0111, which is one level away from 1000.

Write error function.

A single programming pulse typically has poor precision due to process variation and nondeterministic material behavior. As a result, MLC designs for both Flash and PCM adopt iterative program-and-verify (P&V) mechanisms [35, 43]. In PCM, each P&V iteration adjusts the cell's resistance and then reads it back to check whether the correct value was achieved. The process continues until an acceptable resistance value has been set. To model the latency and error characteristics of iterative writes, we consider the effect of each step to be drawn from a normal distribution. The write mechanism determines the ideal pulse size but applies that pulse with some error added. Figure 2 illustrates one iteration in this process.

Two parameters control the operation of the P&V write algorithm. First, iteration terminates when the stored value is within a threshold distance T from the target value. Setting $T < \frac{1}{2n}$ as in Figure 1 provides *guard bands* between the levels to account for read error. The value of T dictates the probability that a read error will occur. Second, the variance of the normal distribution governing the effect of each pulse is modeled as a constant proportion, P, of the intended step size. These parameters determine the average number of iterations required to write the cell.

Figure 3 shows the pseudocode for writes, which resembles the PCM programming feedback control loop of Pantazi et al. [28]. Section 5.2 describes our methodology for calibrating the algorithm's parameters to reflect realistic PCM systems.

Each constituent write pulse in an PCM write can either increase or decrease resistance [28,29,31]. Flash write pulses, in contrast, are unidirectional, so writes must be more conservative to avoid costly RESET operations in the case of overprogramming [42]. While this paper focuses on PCM, we also modeled approximate Flash MLCs and found that the technique also improves Flash writes, but the benefit is smaller (1.26× write speedup versus PCM's 1.7×).

Read error function.

Reading from a storage cell is also imprecise. PCM cells are subject to both *noise*, random variation in the stored value, and *drift*, a gradual unidirectional shift [32]. We reuse the model and parameters of Yeo et al. [46]. Namely, the sensed analog value $r(v)$ is related to the written value v as $r(v) = v + \log_{10} t \cdot N(\mu_r, \sigma_r^2)$ where t is the time, in seconds, elapsed since the cell was written. The parameters μ_r and σ_r are the mean and standard deviation of the error effect respectively.

The same error function, with t equal to the duration of a write step, is used to model errors during the verification step of the write process. We use $t = 250$ ns [19, 33] for this case.

Section 5.2 details the parameters used in our experiments.

Read quantization.

A read operation must determine the digital value corresponding to the analog value $r(v)$. We assume reads based on a successive approximation analog-to-digital converter (ADC), which has previously been proposed for use with PCM systems that can vary their level count [35]. The latency for a successive approximation ADC is linear in the number of bits (i.e., $\log n$).

Model simplifications.

While this model is more detailed than some recent work, which has used simple closed-form probability distributions to describe program-and-verify writes [19, 33], it necessarily makes some simplifications over the full complexity of the physics underlying PCM.

For simplicity, our model does not incorporate differential writes, a technique that would allow a write to begin without an initial RESET pulse [19]. The write algorithm also does not incorporate the detection of hard failures, which is typically accomplished by timing out after a certain number of iterations [43]. Hard failure detection is orthogonal to the approximate MLC technique.

We measure write performance improvement in terms of the number of iterations per write. While some MLC write techniques use different durations for differently sized pulses [3, 27, 28], we expect the pulses to have approximately the same average time in aggregate. Previous work, for example, has assumed that each step takes 250 nanoseconds [19,33]. Furthermore, since our evaluation focuses on performance and energy, we do not model any potential lifetime benefits afforded by the technique's reduction in write pulses.

Finally, our model assumes for simplicity that the value range has uniform guard band sizes—in terms of our model, the threshold T is constant among levels. Asymmetric guard bands could exploit the fact that drift is unidirectional. This optimization is orthogonal to the approximate MLC technique, which simply decreases the size of guard bands relative to their nominal size in a precise configuration.

3.2 Encoding Values to Minimize Error

MLC systems typically divide the bits from a single cell among different memory pages [43]. Using this technique, some pages consist of high-order bits from many cells while other pages consist entirely of low-order bits. In approximate MLCs, low-order bits are the least reliable. So this traditional strategy would lead to pages with uniformly poor accuracy. Here, we use a different approach in order to represent all approximate values with acceptable accuracy.

If each cell has n levels, then individual cells can each represent $\log n$ bits. If a program needs to store $\log n$-bit numbers, then the error characteristics of a single cell are advantageous: a single-level error—when the cell stores $l - 1$ or $l + 1$ after attempting to write l—corresponds to a integer error of 1 in the stored value.

But we also need to combine multiple cells to store larger numbers. We consider two approaches. Concatenation (Figure 4a) appends the bits from the constituent cells to form each word. Striping (Figure 4b) interleaves the cells so that the highest-order bits of each cell all map to the highest-order bits of the word.

An ideal code would make errors in high bits rare while allowing more errors in the low bits of a word. With the straightforward

concatenation code, however, a single-level error can cause a high-order bit flip: the word's $\log n$-th most significant bit is the *least* significant bit in its cell. The striping code mitigates high-bit errors but does not prevent them. In the example shown in Figure 4b, the white cell stores the value 0111, so a single-level error can change its value to 1000. This error causes a bit flip in the word's most significant bit. Gray coding, which some current MLC systems use [19], does not address this problem: single-level errors are as likely to cause flips in high-order bits as in low-order bits.

We evaluate both approaches in Section 6 and find, as expected, that the striping code mitigates errors most effectively.

3.3 Memory Interface

MLC blocks can be made precise or approximate by adjusting the target threshold of write operations. For this reason, the memory array must know which threshold value to use for each write operation. Rather than storing the precision level as metadata for each block of memory, we encode that information in the operation itself by extending the memory interface to include precision flags as described in Section 2. This approach, aside from eliminating metadata space overhead, eliminates the need for a metadata read on the critical path for writes.

Read operations are identical for approximate and precise memory, so the precision flag in read operations goes unused. A different approximate MLC design could adjust the cell density of approximate memory; in this case, the precision flag would control the bit width of the ADC circuitry [35].

Overheads.

Since no metadata is used to control cells' precision, this scheme carries no space overhead. However, at least one additional bit is necessary in each read and write request on the memory interface to indicate the operation's precision. If multiple threshold values are provided to support varying precision levels, multiple bits will be needed. Additional circuitry may also be necessary to permit a tunable threshold value during cell writes. Our performance evaluation, in Section 5, does not quantify these circuit area overheads.

4. USING FAILED MEMORY CELLS

PCM, along with Flash and other upcoming memory technologies, suffers from cell failures during a device's deployment—it "wears out." Thus, techniques for hiding failures from software are critical to providing a useful lifespan for a memory [21]. These techniques typically abandon portions of memory containing uncorrectable failures and use only failure-free blocks [34, 39, 40]. By employing otherwise-unusable failed blocks to store *approximate* data, it is possible to extend the lifetime of an array as long as sufficient intact capacity remains to store the application's *precise* data.

The key idea is to use blocks with exhausted error-correction resources to store approximate data. Previous work on approximate storage in DRAM [24] and SRAM [14] has examined *soft* errors, which occur randomly in time and space. If approximate data is stored in PCM blocks with failed cells, on the other hand, errors will be *persistent*. That is, a value stored in a particular failed block will consistently exhibit bit errors in the same positions. We can exploit the fact that failure positions are known to provide more effective error correction via *bit priorities*.

4.1 Prioritized Bit Correction

In an error model incorporating stuck-at failures, we can use error correction to concentrate failures where they are likely to do the least harm. For example, when storing a floating-point number, a bit error is least significant when it occurs in the low bits of the mantissa and most detrimental when it occurs in the high bits of the exponent or the sign bit. In a uniform-probability error model, errors in each location are equally likely, while a deterministic-failure model affords the opportunity to protect a value's most important bits.

A correction scheme like error-correcting pointers (ECP) [39] marks failed bits in a block. Each block has limited correction resources; for example, when the technique is provisioned to correct two bits per block (ECP_2), a block becomes unusable for precise storage when three bits fail. For approximate storage, we can use ECP to correct the bits that appear in high-order positions within words and leave the lowest-order failed bits uncorrected. As more failures appear in this block, only the least-harmful stuck bits will remain uncorrected.

4.2 Memory Interface

A memory module supporting failed-block recycling determines which blocks are approximate and which may be used for precise storage. Unlike with the approximate MLC technique (Section 3), software has no control over blocks' precision state. To permit safe allocation of approximate and precise data, the memory must inform software of the locations of approximate (i.e., failed) blocks.

When the memory module is new, all blocks are precise. When the first uncorrectable failure occurs in a block, the memory issues an interrupt and indicates the failed block. This is similar to other systems that use page remapping to retire failed segments of memory [18, 47]. The OS adds the block to a pool of approximate blocks. Memory allocators consult this set of approximate blocks when laying out data in the memory. While approximate data can be stored in any block, precise data must be allocated in memory without failures. Eventually, when too many blocks are approximate, the allocator will not be able to find space for all precise data—at this point, the memory module must be replaced.

To provide traditional error correction for precise data, the memory system must be able to detect hard failures after each write [39]. We reuse this existing error detection support; the precision level of the write operation (see Section 2) determines the action taken when a failure is detected. When a failure occurs during a precise write, the module either constructs ECP entries for all failed bits if sufficient entries are available or issues an interrupt otherwise. When a failure occurs during an approximate write, no interrupt is issued. The memory silently corrects as many errors as possible and leaves the remainder uncorrected.

To make bit prioritization work, the memory module needs information from the software indicating which bits are most important. Software specifies this using a value size associated with each approximate write as described in Section 2. The value size indicates the homogenous byte-width of approximate values stored in the block. If a block represents part of an array of double-precision floating point numbers, for example, the appropriate value size is 8 bytes. This indicates to the memory that the bits at index i where $i \equiv 0 \bmod 64$ are most important, followed by 1 mod 64, etc. When a block experiences a new failure and the memory module must choose which errors to correct, it masks the bit indices of each failure to obtain the index modulo 64. It corrects the bits with the lowest indices and leaves the remaining failures uncorrected.

This interface for controlling bit prioritization requires blocks to contain homogeneously sized values. In our experience, this is a common case: many of the applications we examined use approximate `double[]` or `float[]` arrays that span many blocks.

Overheads.

Like the approximate MLC scheme, failed-block recycling requires additional bits for each read and write operation in the mem-

ory interface. Messages must contain a precision flag and, to enable bit priority, a value size field. The memory module must incorporate logic to select the highest-priority bits to correct in an approximate block; however, this selection happens rarely because it need only occur when new failures arise. Finally, to correctly allocate new memory, the OS must maintain a pool of failed blocks and avoid using them for precise storage. This block tracking is analogous to the way that flash translation layers (FTLs) remap bad blocks.

5. EVALUATION

Approximate storage trades off precision for performance, durability, and density. To understand this trade-off in the context of real-world approximate data, we simulate both of our techniques and examine their effects on the quality of data sets and application outputs. As with previous work on approximate computing [15, 38, 41], we use application-specific metrics to quantify quality degradation.

We first describe the main-memory and persistent-data benchmarks used in our evaluation. We then detail the MLC model parameters that dictate performance and error rates of the approximate MLC technique. Finally, we describe the model for wear-out used in our evaluation of the failed-block recycling technique.

5.1 Applications

We use two types of benchmarks in our evaluation: main-memory applications and persistent data sets. The main-memory applications are Java programs annotated using the EnerJ [38] approximation-aware type system, which marks some data as approximate and leaves other data precise. The persistent-storage benchmarks are static data sets that can be stored 100% approximately.

For the main-memory applications, we adapt the annotated benchmarks from the evaluation of EnerJ. An in-house simulator intercepts loads and stores to collect access statistics and inject errors. The applications are chosen from a broad range of domains for their tolerance to imprecision. We examine a 3D triangle intersection kernel from a game engine (jmeint), a ray tracing image renderer (raytracer), a visual bar code recognizer for mobile phones (zxing), and five scientific kernels from the SciMark2 benchmark suite (fft, lu, mc, smm, and sor). For the benchmarks with vector or matrix output, the error metric is the mean pointwise entry difference. For the benchmarks with all-or-nothing output correctness, jmeint and zxing, the metric is the proportion of correct decisions. For raytracer, the metric is the mean pixel value difference. More details on the annotation and quality assessment of these benchmarks can be found in the evaluation of EnerJ [38].

For persistent storage, we examine four sets of approximate data. The first, sensorlog, consists of a log of mobile-phone sensor readings from an accelerometer, thermometer, photodetector, and hydrometer. The data is used in a decision tree to infer the device's context, so our quality metric is the accuracy of this prediction relative to a fully-precise data set. The second, image, stores a bitmap photograph as an array of integer RGB values. The quality metric is the mean error of the pixel values. The final two data sets, svm and ann, are trained classifiers for handwritten digit recognition based on a support vector machine and a feed-forward neural network. In both cases, the classifiers were trained using standard algorithms on the "pendigits" data set from the UCI Machine Learning Repository [1]. The data set consists of 3498 training samples and 7494 testing samples, each of which comprises 16 features. Then, the classifier parameters (support vectors and neuron weights, respectively) are stored in approximate memory. The SVM uses 3024 support vectors; the NN is configured with a sigmoid activation function, two hidden layers of 128 neurons each, and a one-hot output layer of 10 neurons. We measure the recognition accuracy of each classifier on an unseen test data set relative to the accuracy of the precise classifier (95% for svm and 80% for ann). Unlike the main-memory applications, which consist of a mixture of approximate and precise data, the persistent data sets are entirely approximate.

5.2 MLC Model Parameters

To assess our approximate MLC technique, we use the model described in Section 3.1. The abstract model has a number of parameters that we need to select for the purposes of simulation. To set the parameters, we use values from the literature on MLC PCM configurations. Since our architecture-level model of iterative program-and-verify writes is original, we infer its parameters by calibrating them to match typical write latencies and error rates.

For a baseline (precise) MLC PCM cell, we need a configuration where errors are very improbable but not impossible. We choose a conservative baseline raw bit error rate (RBER) of 10^{-8}, which comports with RBERs observed in Flash memory today [4, 25].

We first select parameters for the read model in Section 3.1, which incorporates the probabilistic effects of read noise and drift. For the parameters μ_r and σ_r, we use typical values from Yeo et al. [46] normalized to our presumed 0.0–1.0 value range. Specifically, for PCM, we choose $\mu_r = 0.0067$ and $\sigma_r = 0.0027$. Since the read model incorporates drift, it is sensitive to the retention time between writes and reads. Retention time can be very short in a main-memory deployment and much longer when PCM is used for persistent storage. As an intermediate value, we consider retention for $t = 10^5$ seconds, or slightly more than one day. Note that this retention time is pessimistic for the main-memory case: in our experiments, every read experiences error as if it occurred 10^5 seconds after the preceding write. In real software, the interval between writes and subsequent reads is typically much lower.

We model a 4-level (2-bit) PCM cell. To calibrate the write model, we start from an average write time of 3 cycles as suggested by Nirschl et al. [27] and a target RBER of 10^{-8}. We need values for the parameters T and P that match these characteristics. We choose our baseline threshold to be 20% of the largest threshold that leads to non-overlapping values (i.e., $T = 0.025$); this leads to about 3 iterations per write. Setting $P = 0.035$ leads to an error probability on the order of 10^{-8} for a retention time of 10^5 seconds.

5.3 Wear-Out Model

To evaluate the effect of using blocks with failed cells for approximate storage, we simulate single-level PCM. In single-level PCM, bits become stuck independently as their underlying cells fail. With multi-level designs, in contrast, a single cell failure can cause multiple bits to become stuck, so bit failures are not independent. Assuming that the memory assigns bits from a given cell to distinct pages [43] and that wear leveling randomly remaps pages, failures nonetheless *appear* independent in multi-level PCM. So a multi-level failure model would closely resemble our single-level model with an accelerated failure rate.

We evaluate PCM with 2-bit error-correcting pointers (ECP) [39]. While precise configurations of the ECP technique typically use 6-bit correction, approximate storage can extend device lifetime without incurring as much overhead as a fully precise configuration. Approximate blocks also use the bit priority assignment mechanism from Section 4.1: where possible, ECP corrections are allocated to higher-order bits within each value in the block.

To understand the occurrence of stuck bits in failed blocks, we need a realistic model for the rate at which cells wear out over time. To this end, we simulate a PCM array for trillions of writes and measure the distribution of cell failures among blocks. The statistical simulator is adapted from Azevedo et al. [11] and assumes

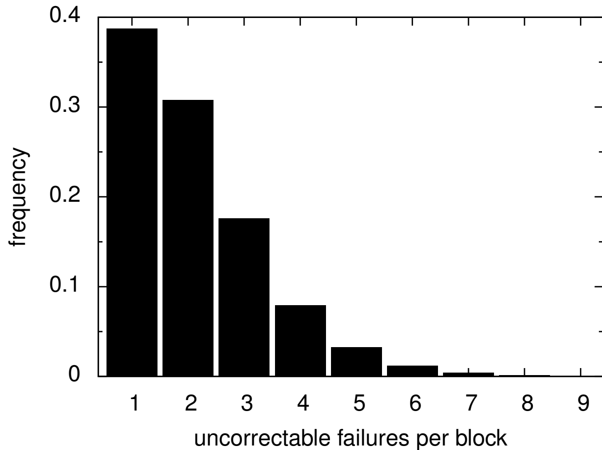


Figure 5: Distribution of uncorrectable cell failures using ECP_2 among 512-bit blocks after the entire memory has been overwritten 3.2×10^7 times under the main-memory wear model. (At this stage, half of the blocks have at least one uncorrectable failure.)

an average PCM cell lifetime of 10^8 writes (although the *first* failure occurs much earlier). We use separate workloads to simulate wear in a main-memory setting and in a persistent-storage setting.

Main-memory wear.

To model wear in main-memory PCM deployments, we simulate the above suite of main-memory applications and gather statistics about their memory access patterns, including the relative size of each program's approximate vs. precise data and the frequency of writes to each type of memory. We then take the harmonic mean of these statistics to create an aggregate workload consisting of the entire suite. We run a statistical PCM simulation based on these application characteristics, during which all blocks start out precise. When a block experiences its first uncorrectable cell failure, it is moved to the approximate pool. Failed blocks continue to be written and experience additional bit failures because they store approximate data. Periodically, we record the amount of memory that remains precise along with the distribution of failures among the approximate blocks. We simulate each application under these measured failure conditions.

As an example, Figure 5 depicts the error rate distribution for the wear stage at which 50% of the memory's blocks have at least one failure that is uncorrectable using ECP_2—i.e., half the blocks are approximate. In this stage, most the of the blocks have only a few uncorrectable failures: 39% of the approximate blocks have exactly one such failure and only 1.7% have six or more.

Persistent-storage wear.

For our persistent-storage data sets, all data is approximate. So we simulate writes uniformly across all of memory, both failed and fully-precise. This corresponds to a usage scenario in which the PCM array is entirely dedicated to persistent storage—no hybrid transient/persistent storage is assumed. As with the main-memory wear model, we periodically snapshot the distribution of errors among all blocks and use these to inject bit errors into stored data.

6. RESULTS

We evaluate two sets of benchmarks under each of our two approximate storage techniques.

6.1 Approximate MLC Memory

In our approximate MLC experiments, we map all approximate data to simulated arrays of two-bit PCM cells. We run each benchmark multiple times with differing threshold (T) parameters. We use T values between 20% and 90% of the maximum threshold (i.e., the threshold that eliminates guard bands altogether). For each threshold, we measure the average number of iterations required to write a random value. This yields an application-independent metric that is directly proportional to write latency (i.e., inversely proportional to performance). Configurations with fewer iterations per write are faster but cause more errors. So, for each application, the optimal configuration is the one that decreases write iterations the most while sacrificing as little output quality as possible. Faster writes help close PCM's performance gap with DRAM in the main-memory case and improve write bandwidth in the persistent-data case [21, 23].

Approximate main memory.

Figure 6a relates write performance to application output quality loss. For configurations with fewer write iterations—to the right-hand side of the plot—performance improves and quality declines. The leftmost point in the plot is the nominal configuration, in which writes take 3.03 iterations on average and errors are rare. Reducing the number of iterations has a direct impact on performance: a 50% reduction in iterations leads to 2× improvement in write speed.

The error for each application stays low for several configurations and then increases sharply when errors become too frequent. The raytracer benchmark exhibits quality loss below 2% up to the configuration with 1.71 iterations per write on average, a 1.77× speedup over the baseline. Even the least tolerant application, fft, sees only 4% quality loss when using an average of 2.44 iterations per write (or 1.24× faster than the baseline). This variance in tolerance suggests that different applications have different optimal MLC configurations. Approximate memories can accommodate these differences by exposing the threshold parameter T for tuning by the developer or runtime system.

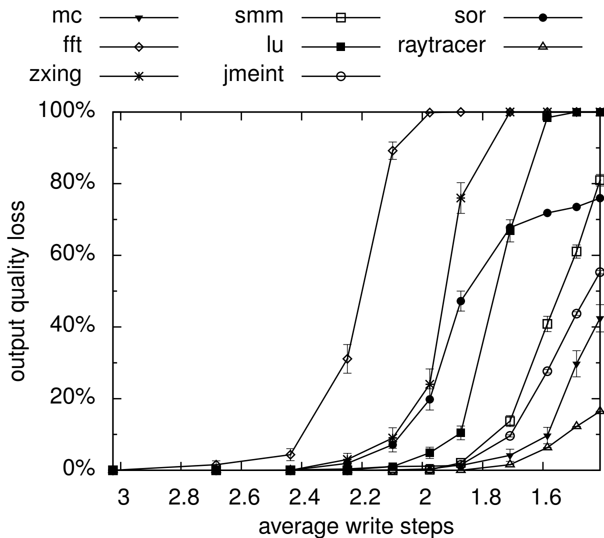
To put these speedups in the context of the whole application, we show the fraction of dynamic writes that are to approximate data in Figure 7. Most applications use approximate writes for more than half of their stores; jmeint in particular has 98% approximate writes. One application, zxing, has a large amount of "cold" approximate data and benefits less from accelerating approximate writes.

Persistent storage.

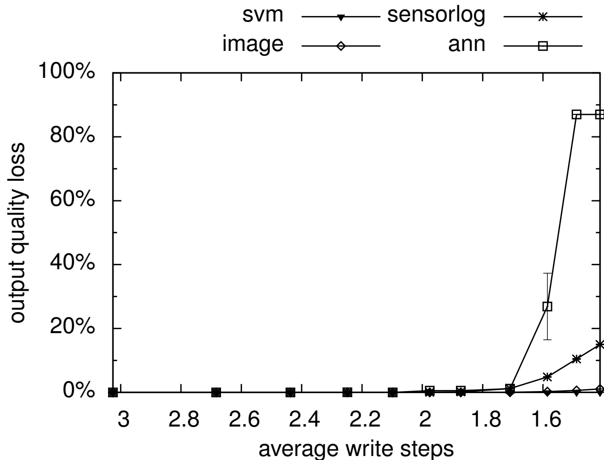
Figure 6b shows the quality degradation for each persistent data set when running on approximate MLC memory. The persistent data sets we examine are more tolerant than the main-memory benchmarks. The sensor logging application, for instance, exhibits only 5% quality degradation in the configuration with 1.59 iterations per write (1.91× faster than the baseline) while the bitmap image has only 1% quality degradation even in the most aggressive configuration we examined, in which writes take 1.41 cycles (2.14× faster than the baseline). The neural network classifier, ann, experiences less than 10% recognition accuracy loss when using 1.77× faster writes; svm, in contrast, saw negligible accuracy loss in every configuration we measured.

Overall, in the configurations with less than 10% quality loss, the benchmarks see 1.7× faster writes to approximate cells over precise cells on average.

This write latency reduction benefits application performance and memory system power efficiency. Since write latency improvements



(a) Main-memory applications with approximate MLC.



(b) Persistent data sets with approximate MLC.

Figure 6: Output degradation for each benchmark using the approximate MLC technique. The horizontal axis shows the average number of iterations per write. The vertical axis is the output quality loss as defined by each application's quality metric. Quality loss is averaged over 100 executions in (a) and 10 in (b); the error bars show the standard error of the mean.

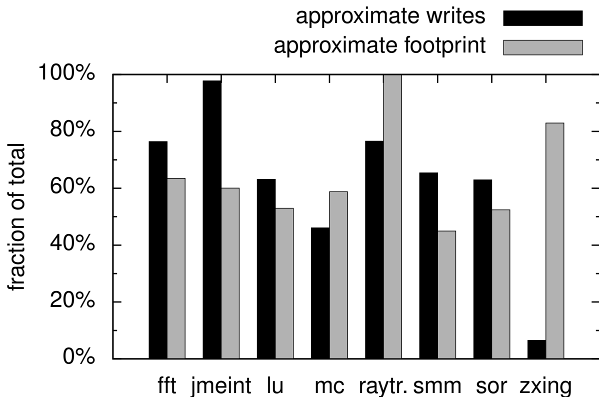


Figure 7: Proportions of approximate writes and approximate data in each main-memory benchmark.

reduce contention and therefore also impact read latency, prior evaluations have found that they can lead to large IPC increases [16, 19]. Since fewer programming pulses are used per write and write pulses make up a large portion of PCM energy, the overall energy efficiency of the memory array is improved.

Impact of encoding.

Section 3.2 examines two different strategies for encoding numeric values for storage on approximate MLCs. In the first, the bits from multiple cells are concatenated to form whole words; in the second, each value is "striped" across constituent cells so that the highest bits of the value map to the highest bits of the cells. The results given above use the latter encoding, but we also evaluated the simpler code for comparison.

The striped code leads to better output quality on average. For three intermediate write speeds, using that code reduces the mean output error across all applications from 1.1% to 0.4%, from 3.6% to 3.0%, and from 11.0% to 9.0% with respect to the naive code.

We also performed two-sample *t*-tests to assess the difference in output quality between the two coding strategies for each of 13 write speed configurations. For nearly every application, the striped code had a statistically significant positive effect on quality more often than a negative one. The only exception is mc, a Monte Carlo simulation, in which the effect of the striped code was inconsistent (positive at some write speeds and negative for others).

While the striped code is imperfect, as discussed in Section 3.2, it fares better than the naive code in practice since it lowers the probability of errors in the high-order bits of words.

Density increase.

We experimented with adding more levels to an approximate MLC. In a precise MLC, increasing cell density requires more precise writes, but approximate MLCs can keep average write time constant. Our experiments show acceptable error rates when six levels are used (and no other parameters are changed). A non-power-of-two MLC requires additional hardware, similar to binary-coded decimal (BCD) circuitry, to implement even the naive code from Section 3.2 but can still yield density benefits. For example, a 512-bit block can be stored in $\lceil \frac{512}{\log 6} \rceil = 199$ six-level cells (compared to 256 four-level cells). With the same average number of write iterations (3.03), many of our benchmarks see very little error: jmeint, mc, raytracer, smm, and the four persistent-storage benchmarks see error rates between 0.09% and 4.19%. The other benchmarks, fft, lu, sor, and zxing, see high error rates, suggesting that density increase should only be used with certain tolerant applications.

Impact of drift.

Previous work has suggested that straightforward MLC storage in PCM can be untenable over long periods of time [46]. Approximate storage provides an opportunity to reduce the frequency of scrubbing necessary by tolerating occasional retention errors. To study the resilience of approximate MLC storage to drift, we varied the modeled retention time (the interval between write and read) and examined the resulting application-level quality loss. Recall that the results above assume a retention time of 10^5 seconds, or about one day, for every read operation; we examined retention times between 10^1 and 10^9 seconds (about 80 years) for an intermediate approximate MLC configuration using an average of 2.1 cycles per write. For the main-memory applications, in which typical retention times are likely to be far less than one day, we see very little quality loss (1% or less) for retention times of 10^4 seconds or shorter. For the persistent-storage benchmarks, quality loss remains under 10% for at least 10^6 seconds and, in the case of image, up to 10^9 seconds.

A longer retention time means scrubbing can be done less frequently. The above results report the quality impact of one retention cycle: the persistent-storage benchmarks, for example, lose less than 10% of their quality when 10^6 seconds, or about 11 days, elapse after they are first written to memory assuming no scrubbing occurs in that time. Eleven more days of drift will compound additional error. While the above results suggest that the more error-tolerant applications can tolerate longer scrubbing cycles, we do not measure how error compounds over longer-term storage periods with infrequent scrubbing.

Bit error rate.

To add context to the output quality results above, we also measured the effective bit error rate (BER) of approximate MLC storage. The BER is the probability that a bit read from approximate memory is different from the corresponding last bit written. Across the write speeds we examined, error rates range from 3.7×10^{-7} to 8.4% in the most aggressive configuration. To put these rates in perspective, if the bit error rate is p, then a 64-bit block will have at least 2 errors with probability $\sum_{i=2}^{64} B(i, 64, p)$ where B is the binomial distribution. At a moderately aggressive write speed configuration with an average of 1.9 steps, approximate MLC storage has an error rate of 7.2×10^{-4}, so 0.1% of 64-bit words have 2 or more errors. This high error rate demonstrates the need for application-level error tolerance: even strong ECC with two-bit correction will not suffice to provide precise storage under such frequent errors.

6.2 Using Failed Blocks

We evaluate the failed-block recycling technique by simulating benchmarks on PCM arrays in varying stages of wear-out. As the memory device ages and cells fail, some blocks exhaust their error-correction budget. Approximate data is then mapped onto these blocks. Over the array's lifetime, bit errors in approximate memory become more common. Eventually, these errors impact the application to such a degree that the computation quality is no longer acceptable, at which point the memory array must be replaced. We quantify the lifetime extension afforded by this technique, beginning with the main-memory applications.

To quantify lifetime extension, we assume a memory module with a 10% "space margin": 10% of the memory is reserved to allow for some block failures before the array must be replaced. In the baseline precise configuration, the array fails when the fraction of blocks that remain precise (having only correctable failures) drops below 90%. In the approximate configuration, programs continue to run until there is not enough space for their precise data or quality drops below a threshold.

Approximate main memory.

Figure 8 depicts the lifetime extension afforded by using failed blocks as approximate storage. For each application, we determine the point in the memory's lifetime (under the wear model described in Section 5.3) at which the program can no longer run. We consider two termination conditions: when the amount of precise memory becomes insufficient (i.e., the proportion of approximate memory exceeds the application's proportion of approximate data) and when the application's output quality degrades more than 10%. Each bar in the figure shows the normalized number of writes to the memory when application failure occurs.

With quality degradation limited to 10%, the benchmarks see lifetime extensions ranging from 2% (zxing) to 39% (raytracer) with a harmonic mean of 18%. With quality unconstrained, the mean lifetime extension is 34%, reflecting the fact that this technique leads to gradually decreasing quality as the memory array ages.

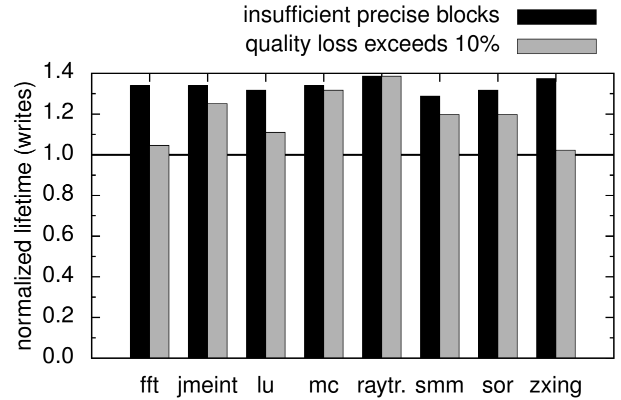


Figure 8: Lifetime extension for each application. Each bar represents the number of writes to the entire array at which the application can no longer run, normalized to the point of array failure in fully-precise mode. The black bar indicates when there is not enough precise memory available. The gray bar shows when the application's output quality degrades more than 10%.

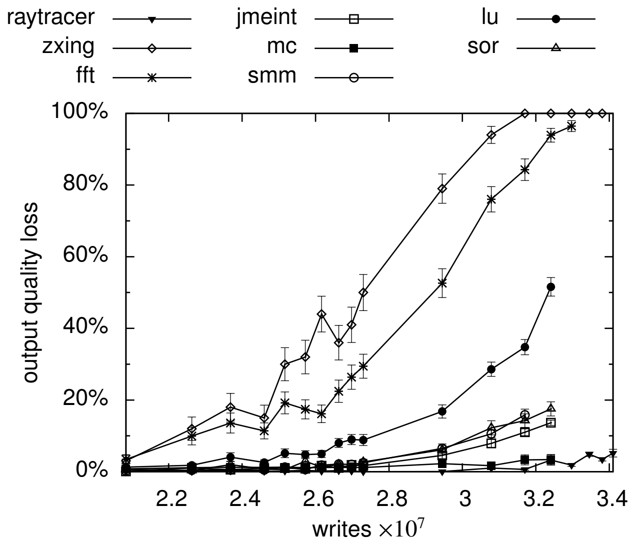
To help explain these results, Figure 9a shows the quality degradation for each application at various points during the memory array's wear-out. The most error-tolerant application, raytracer, sees very little quality degradation under all measured wear stages. Some applications are limited by the amount of approximate data they use. Figure 7 shows the proportion of bytes in each application's memory that is approximate (averaged over the execution). Some applications, such as mc, are tolerant to error but only have around 50% approximate data. In other cases, such as zxing and fft, bit errors have a large effect on the computation quality. In fft in particular, we find that a single floating-point intermediate value that becomes NaN can contaminate the Fourier transform's entire output. This suggests that the application's precision annotations, which determine which data is stored approximately, may be too aggressive.

Persistent storage.

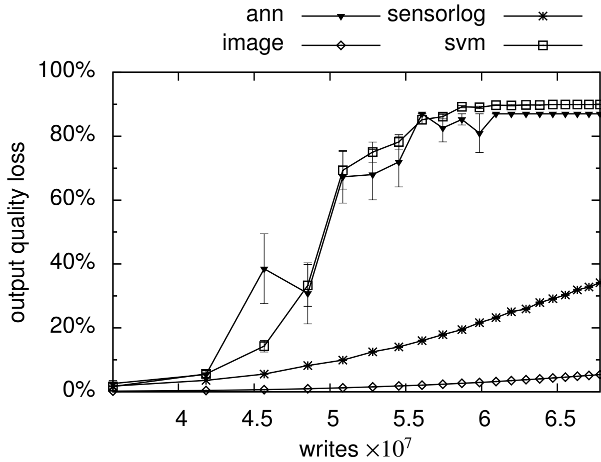
Figure 9b shows the quality degradation for each data set at different points during the lifetime of the memory. The memory's intermediate wear-out conditions come from the persistent-storage wear model described in Section 5.3. In a fully-precise configuration, the memory fails (exceeds 10% failed blocks) at about 3.4×10^7 overwrites, or at the left-hand side of the plot. Recall that, in these persistent-storage benchmarks, the data is stored 100% approximately; no precise storage is used.

As with the main-memory storage setting, quality decreases over time as errors become more frequent. But these benchmarks are more tolerant to stuck bits than the main-memory applications. For image, quality loss is below 10% in all wear stages; for sensorlog, it remains below 10% until the array experiences 5.0×10^7 writes, or 42% later than precise array failure. The two machine learning classifiers, ann and svm, each see lifetime extensions of 17%. This tolerance to stuck bits makes the failed-block recycling technique particularly attractive for persistent storage scenarios with large amounts of numeric data.

Overall, across both categories of benchmarks, we see a harmonic mean lifetime extension of 23% (18% for the main-memory benchmarks and 36% for the persistent-storage data sets) when quality loss is limited to 10%. Recent work has demonstrated PCM arrays that sustain a random write bandwidth of 1.5 GB/s [7]; for a 10 GB



(a) Main-memory applications using failed blocks.



(b) Persistent data sets using failed blocks.

Figure 9: Output quality degradation for each benchmark when using the failed-block recycling technique. The horizontal axis is the number of complete overwrites the array has experienced, indicating the stage of wear-out. The vertical axis is an application-specific error metric.

memory constantly written at this rate, these savings translate to extending the array's lifetime from 5.2 years to 6.5 years.

Impact of bit priority.

The above results use our type-aware prioritized correction mechanism (Section 4.1). To evaluate the impact of bit prioritization, we ran a separate set of experiments with this mechanism disabled to model a system that just corrects the errors that occur earliest. We examine the difference in output quality at each wear stage and perform a two-sample *t*-test to determine whether the difference is statistically significant ($P < 0.01$).

Bit prioritization had a statistically significant positive impact on output quality for all benchmarks except mc. In sensorlog, for example, bit prioritization decreases quality loss from 2.3% to 1.7% in an early stage of wear (the leftmost point in Figure 9b). In fft, the impact is larger: bit prioritization reduces 7.3% quality loss to 3.3% quality loss. As with encoding for approximate MLCs, the exception is mc, whose quality was (statistically significantly) improved in only 4 of the 45 wear stages we measured while it was *negatively* impacted in 6 wear stages. This benchmark is a simple Monte Carlo method and hence may sometimes *benefit* from the entropy added by failed bits. Overall, however, we conclude that bit prioritization has a generally positive effect on storage quality.

Impact of ECP budget.

The above experiments use a PCM configuration with error-correcting pointers (ECP) [39] configured to correct two stuck bits per 512-bit block at an overhead of 21 extra bits per block. More aggressive error correction improves the endurance of both fully-precise and approximate memory and amplifies the opportunity for priority-aware correction in intermediate wear stages. To quantify the effect of increasing error correction budgets, we also evaluated an ECP_6 configuration (61 extra bits per block).

Moving from ECP_2 to ECP_6 extends the lifetime of a precise memory array by 45% under main-memory wear or 17% under persistent-storage wear. Our results for approximate main-memory storage with ECP_2 provide a portion of these benefits (18% lifetime extension) while incurring no additional error-correction overhead. In the persistent-storage case, the lifetime extension for approximate storage (36%) is greater than for increasing the ECP budget.

7. RELATED WORK

Approximate storage builds on three broad categories of related work: approximate computing, optimizing accesses to storage cells, and tolerating failures in solid-state memories.

Approximate computing is an area of research that seeks to optimize the execution of error-tolerant programs using both hardware [8, 12, 14, 15, 24, 26] and software [2, 17, 41] techniques. Programmers control the impact of approximate execution using language features, analyses, or program logics [5,6,12,38]. While much of the work on approximate computing has focused on computation itself—optimizing algorithms or processor logic—some recent work has proposed to lower the refresh rate of DRAM [24] or the supply voltage of SRAM [9, 14, 20]. This paper, in contrast, explores techniques that take advantage of the unique properties of *non-volatile* solid-state storage technologies like PCM and Flash: wear-out and multi-level configurations. Unlike prior work on SRAM and DRAM, we evaluate approximate storage for both transient (main memory) and persistent (filesystem or database) data.

Prior work has also explored techniques for optimizing accesses to PCM and Flash [19, 33] or architecting systems to efficiently use PCM as main memory [18,21,36,39,47]. In particular, half-wits [37] and power fade [44] use low-voltage, error-prone Flash operations to reduce power and energy. Similarly, retention relaxation uses less-precise program-and-verify parameters to speed up writes to MLC Flash cells when data has a short lifetime [23]. Approximate MLCs complement these techniques by allowing even faster writes when bit errors are tolerable. A system can, for example, combine retention relaxation and approximate storage by "underestimating" the necessary retention time of approximate data. Previous work has also proposed adapting the density of SLC and MLC cells in main memory [35] and persistent storage [13] deployments. While these systems trade off density for performance, energy, and endurance, data is always stored precisely. This paper proposes MLC configurations that permit storage errors but improve density or access efficiency beyond what is possible in precise configurations.

Prior work has also considered low-overhead techniques for hiding failures in memories that experience wear-out [11, 34, 39, 40]. Our failed-block recycling technique extends this prior work by selectively relaxing error correction on memory that contains approximate data.

Approximate storage resembles techniques for lossy compression: both improve resource usage at the cost of some lost information. But by exploiting the characteristics of the underlying storage technology, approximate storage differs from traditional compression in two important ways. First, whereas lossy compression deterministically discards data, approximate storage techniques exhibit probabilistic data retention. This difference makes approximate storage more suited to applications where random errors are tolerable, such as in neural networks. Second, failed block recycling unlocks more usable bytes of memory that are not available to traditional compression. For example, a system that compresses data in precise blocks but stores uncompressed data in blocks with failures can conserve more space than either technique allows independently. Composing approximation with compression on the same data, however, can lead to poor results because each bit represents a larger amount of data. (Encryption has a similar effect.) Although an evaluation of lossy compression techniques is out of this paper's scope, a quantitative comparison with approximate storage would be illuminating future work.

8. CONCLUSION

Solid-state, non-volatile storage technologies such as PCM and Flash are becoming increasingly important components of the modern computing landscape. As DRAM scaling begins to falter, PCM and other resistive memories will become crucial to satisfying increasing memory needs in mobile and server settings alike. But these technologies present new challenges due to wear-out and slow writes, especially in multi-level cell configurations. For many applications, however, perfect data retention is not always necessary—the time and space spent on ensuring correct operation is wasted for some data. *Approximate storage*, via approximate MLCs and failed-block recycling, represents an opportunity to exploit this error tolerance to improve performance, energy, and capacity. Our results for write acceleration and lifetime extension suggest that approximate storage can help mitigate the drawbacks of these solid-state, non-volatile memories while compromising only a small amount of storage quality.

Acknowledgments

We would like to thank our anonymous reviewers for their invaluable comments. Our thanks also to Doug Burger for early discussions on the project, to Ben Ransford for comments on the text, and to Mattan Erez for suggestions on improving the paper. This work was supported in part by NSF award #1216611, gifts from Microsoft Research, and the Facebook Graduate Fellowship.

9. REFERENCES

[1] K. Bache and M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[3] S. Braga, A. Sanasi, A. Cabrini, and G. Torelli, "Voltage-driven partial-RESET multilevel programming in phase-change memories," *IEEE Transactions on Electron Devices*, vol. 57, no. 10, pp. 2556–2563, 2010.

[4] Y. Cai, E. Haratsch, O. Mutlu, and K. Mai, "Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis," in *DATE*, 2012.

[5] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Reasoning about relaxed programs," in *PLDI*, 2012.

[6] M. Carbin, S. Misailovic, , and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO*, 2010.

[8] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006.

[9] I. J. Chang, D. Mohapatra, and K. Roy, "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.

[11] R. J. de Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie: Extending memory lifetime by reviving dead blocks," in *ISCA*, 2013.

[12] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[13] X. Dong and Y. Xie, "AdaMS: Adaptive MLC/SLC phase-change memory design for file storage," in *Asia and South Pacific Design Automation Conference*, 2011.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[15] ——, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[16] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger, "Preventing PCM banks from seizing too much power," in *MICRO*, 2011.

[17] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

[18] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: Building reliable systems from nanoscale resistive memories," in *ASPLOS*, 2010.

[19] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in MLC phase change memory," in *HPCA*, 2012.

[20] A. Kumar, J. Rabaey, and K. Ramchandran, "SRAM supply voltage scaling: A reliability perspective," in *International Symposium on Quality of Electronic Design*, 2009.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.

[22] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

[23] R.-S. Liu, C.-L. Yang, and W. Wu, "Optimizing NAND flash-based SSDs via retention relaxation," in *FAST*, 2012.

[24] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.

[25] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill, “Bit error rate in NAND flash memories,” in *International Reliability Physics Symposium*, 2008.

[26] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, “Scalable stochastic processors,” in *DATE*, 2010.

[27] T. Nirschl, J. Phipp, T. Happ, G. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H.-L. Lung, and C. Lam, “Write strategies for 2 and 4-bit multi-level phase-change memory,” in *IEDM*, 2007.

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

[29] N. Papandreou, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, H. Pozidis, and E. Eleftheriou, “Multilevel phase-change memory,” in *IEEE International Conference on Electronics, Circuits, and Systems*, 2010.

[30] N. Papandreou, H. Pozidis, T. Mittelholzer, G. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, “Drift-tolerant multilevel phase-change memory,” in *IEEE International Memory Workshop*, 2011.

[31] N. Papandreou, H. Pozidis, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, and E. Eleftheriou, “Programming algorithms for multilevel phase-change memory,” in *ISCAS*, 2011, pp. 329–332.

[32] H. Pozidis, N. Papandreou, A. Sebastian, T. Mittelholzer, M. BrightSky, C. Lam, and E. Eleftheriou, “A framework for reliability assessment in multilevel phase-change memory,” in *IEEE International Memory Workshop*, 2012.

[33] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, “Improving read performance of phase change memories via write cancellation and write pausing,” in *HPCA*, 2010.

[34] M. K. Qureshi, “Pay-as-you-go: low-overhead hard-error correction for phase change memories,” in *MICRO*, 2011.

[35] 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 *ISCA*, 2010.

[36] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *ISCA*, 2009.

[37] M. Salajegheh, Y. Wang, K. Fu, A. Jiang, and E. Learned-Miller, “Exploiting half-wits: Smarter storage for low-power devices,” in *FAST*, 2011.

[38] 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 *PLDI*, 2011.

[39] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ECP, not ECC, for hard failures in resistive memories,” in *ISCA*, 2010.

[40] N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers, and H.-H. Lee, “SAFER: Stuck-at-fault error recovery for memories,” in *MICRO*, 2010.

[41] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” in *FSE*, 2011.

[42] K.-D. Suh, B.-H. Suh, Y.-H. Lim, J.-K. Kim, Y.-J. Choi, Y.-N. Koh, S.-S. Lee, S.-C. Kwon, B.-S. Choi, J.-S. Yum, J.-H. Choi, J.-R. Kim, and H.-K. Lim, “A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme,” *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1149–1156, 1995.

[43] K. Takeuchi, T. Tanaka, and T. Tanzawa, “A multipage cell architecture for high-speed programming multilevel NAND flash memories,” *IEEE Journal of Solid-State Circuits*, vol. 33, no. 8, pp. 1228 –1238, 1998.

[44] H.-W. Tseng, L. M. Grupp, and S. Swanson, “Underpowering NAND flash: Profits and perils,” in *DAC*, 2013.

[45] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ASPLOS*, 2011.

[46] 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 *Workshop on Duplicating, Deconstructing and Debunking*, 2012.

[47] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” in *ISCA*, 2009.