Enabling Approximate Storage through Lossy Media Data Compression

Brian David Worek

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

December 13, 2018
Blacksburg, VA

Keywords: approximate storage, lossy compression, data-intensive

Enabling Approximate Storage through Lossy Media Data Compression

Brian David Worek

ABSTRACT

Memory capacity, bandwidth, and energy all continue to present hurdles in the quest for efficient, high-speed computing. Recognition, mining, and synthesis (RMS) applications in particular are limited by the efficiency of the memory subsystem due to their large datasets and need to frequently access memory. RMS applications, such as those in machine learning, deliver intelligent analysis and decision making through their ability to learn, identify, and create complex data models. To meet growing demand for RMS application deployment in battery constrained devices, such as mobile and Internet-of-Things, designers will need novel techniques to improve system energy consumption and performance. Fortunately, many RMS applications demonstrate *inherent error resilience*, a property that allows them to produce acceptable outputs even when data used in computation contain errors. *Approximate storage* techniques across circuits, architectures, and algorithms exploit this property to improve the energy consumption and performance of the memory subsystem through quality-energy scaling. This thesis reviews state of the art techniques in *approximate storage* and presents our own contribution that uses lossy compression to reduce the storage cost of media data.

Enabling Approximate Storage through Lossy Media Data Compression

Brian David Worek

GENERAL AUDIENCE ABSTRACT

Computer memory systems present challenges in the quest for more powerful overall computing systems. Computer applications with the ability to learn from large sets of data in particular are limited because they need to frequently access the memory system. These applications are capable of intelligent analysis and decision making due to their ability to learn, identify, and create complex data models. To meet growing demand for intelligent applications in smartphones and other Internet connected devices, designers will need novel techniques to improve energy consumption and performance. Fortunately, many intelligent applications are naturally resistant to errors, which means they can produce acceptable outputs even when there are errors in inputs or computation. Approximate storage techniques across computer hardware and software exploit this error resistance to improve the energy consumption and performance of computer memory by purposefully reducing data precision. This thesis reviews state of the art techniques in approximate storage and presents our own contribution that uses lossy compression to reduce the storage cost of media data.

Acknowledgements

I would first like to thank my research advisor, Dr. Paul Ampadu, for his guidance and support. He has taught me so much about research and communication and has been instrumental to my success in writing this thesis.

Thank you to Dr. Peter Athanas, Dr. Patrick Schaumont, and Dr. Dong Ha for serving on my advisory committee and for their review of and feedback on my thesis.

Thank you to my teammates on the Embedded Integrated System-On-Chip (EdISon) research group and all members of the Multifunctional Integrated Circuits and Systems (MICS) research group for supporting me in my research endeavors.

Thank you to my classmate and friend, Jinwoo Yom, for helping me gather data during my research.

Finally, thank you to all my family and friends for supporting me during this journey. I could not have done it without you.

Contents

Acknowledgements iv

Contents v

List of Figures vii

List of Tables xi

1 Introduction 1

1.1 Data-Intensive Applications 2

1.2 Approximate Computing and Storage 4

1.2.1 Inherent Error Resilience 5

1.2.2 Design Requirements 9

1.3 Outline 11

2 State of the Art in Approximate Storage 12

2.1 Circuit Level Storage Optimization 13

2.1.1 SRAM Supply Voltage Scaling 13

2.1.2 DRAM Refresh Rate Reduction 17

2.1.3 Multi-Level Cell Storage 20

2.1.4 STT-MRAM Access Reliability 22

2.2 Architecture Level Storage Optimization 25

2.2.1 Architecture and Programming Support 25

2.2.2 Memory Load Skipping and Prediction 27

2.2.3 Approximate Caches 31

2.3 Algorithm Level Storage Optimization 34

2.3.1 Precision Scaling of Data 34

2.3.2 Memoization and Look-Up Tables 37

3 Proposed Work: Improving Compression with Precision Scaling 41

3.1 Introduction 41

3.2 BΔI Lossless Compression 43

3.3 Approximating Data 47

3.4 Evaluation Methodology 51

3.4.1 Compression and Error Metrics 52

3.4.2 Image Classification 53

3.4.3 Hardware 54

3.5 Results 55

3.5.1 Effective Storage Capacity 56

3.5.2 Quantitative Error 58

3.5.3 Image Classification Accuracy 60

3.5.4 Hardware 61

4 Conclusions 65

Bibliography 66

List of Figures

Figure 1.1: Original, high quality image (leftmost) and the same image after lossy compression with an increasing number of least-significant bits approximated. 8

Figure 2.1: Accuracy-aware, dual-voltage SRAM array. The quantization control signals select either V_{high} or V_{low} as the supply voltage for each highlight voltage domain. Attribution: (Cho et al. 2015) [51]... 14

Figure 2.2: Dual-voltage SRAM array. Precision signal controls whether $V_{DD}L$ or $V_{DD}H$ is used as supply voltage for each row's precharge logic, bit cells, and sense amplifiers. Attribution: (Esmaeilzadeh et al. 2012) [36]. 16

Figure 2.3: Flikker partitions DRAM banks into two regions: high refresh for critical data and low refresh for non-critical data. The partition boundary can be assigned at any of the dashed lines. Attribution: (Liu et al. 2011) [8]. 18

Figure 2.4: The range of analog values in (a) precise and (b) approximate MLC. The shaded regions represent target levels and the unshaded areas represent guard bands. Each curve represents the probability of reading a given value from the cell. In the approximate cell (b), the reduced guard bands lead to an overlap in possible read values where erroneous values may be read. Attribution: (Sampson et al. 2013) [15]... 21

Figure 2.5: STT-MRAM bit cell consisting of a magnetic tunnel junction (MTJ) and an access transistor. Attribution: (Ranjan et al. 2017) [56]. 23

Figure 2.6: Load value approximation (LVA) architecture overview (a) and design (b). The highlighted box in (a) represents the design in (b). Upon a load miss X in the cache (a.1), the Load Value Approximator generates an approximate X value (a.2), which the processor will compute with (a.3a). The missed load X is still fetched from main memory (a.3b) for the approximator in order to improve the accuracy (a.4) of future load approximations. The load value approximator (b) uses context from the load instruction address and a global history buffer to select an entry in the approximator table, where it will then select a load value based on a local history buffer. Attribution: (Miguel et al. 2014) [22]. 29

Figure 2.7: Doppelganger cache concept. Doppelganger associates tags of similar blocks with same entry in an approximate data array. This significantly reduces the size of the data array necessary to store approximately similar data. Attribution: (Miguel et al. 2015) [42]................ 32

Figure 2.8: Application output quality due to sweeping similarity strides. Peaks indicate a high degree of similarity between data separated by the similarity stride length. Troughs indicate a low degree of similarity. Attribution: (Miguel et al. 2016) [44]... 33

Figure 2.9: ApproxMA off-chip memory data format. A, B, …, H represent the original data words, which are now separated into 4-bit segments and distributed across 8 memory blocks. The first block stores the 4 MSBs of each word and each subsequent block stores the next most significant 4-bits for each set of words. Attribution: (Tian et al. 2015) [37].. 35

Figure 2.10: Bidirectional precision scaling of storage data. M and L indicate the number of MSBs and LSBs to be truncated from each word. Upon decompression, the truncated MSBs are replaced with the stored PadBit and the LSBs are replaced with zeros. Attribution: (Ranjan et al. 2017) [18]. 36

Figure 2.11: Lossy compression of floating-point (FP) data in hardware achieved by combining truncation with lossless compression. LSBs of FP data words are truncated before applying lossless compression to a whole block of data words. Truncating the LSBs increases the efficiency of the lossless compression. Attribution: (Sathish et al. 2012) [64]. 37

Figure 2.12: Fuzzy memoization architecture. Results of floating-point arithmetic logic unit (FPALU) are stored in a memo, or look-up, table (LUT) for reuse. Later, the two input operands are used to generate an address to search the LUT with. In this architecture, some number, N, of least significant mantissa bits are truncated from each operand before searching the LUT. In a precise LUT, only a precise match for a set of inputs may be returned as the function result, but now similar values can be used when the difference in values lies within those lower N mantissa bits. When a result is found in the LUT, the FPALU is bypassed and the result is directly output. Attribution: (Alveraz et al. 2005) [40]... 39

Figure 3.1: Example of BΔI compression. Using the first word of the cache line as a base, subtraction is performed between every word and the base. The differences are stored after the base as an array of deltas. Each word in the uncompressed cache line is 4 bytes wide, but each delta in the compressed cache line is only 1 byte wide. Attribution: (Pekhimenko et al. 2012) [43]. 45

Figure 3.2: BΔI compression architecture. In the first stage, compressor units (CUs) perform BΔI arithmetic in addition to repeated and zero value checks. In the second stage, the most efficient valid CU output encoding is selected as the compressed block based on the BΔI encoding table. 46

Figure 3.3: Base8-Δ1 compressor unit. First, subtraction is performed between each 8-byte word of the uncompressed cache line and the selected base, V_0. Differences are then checked for sign extension from the least significant byte. If true for all differences, then the differences are stored as 1-byte deltas in the compressed cache line and the valid encoding signal is sent to compression selection logic as shown in Figure 3.2. Attribution: (Pekhimenko et al. 2012) [43]. 46

Figure 3.4: Generic compressor unit (CU) architecture with added right-shifters before subtraction. Specific implementations exist for all BΔ combinations from Table 3.2. The uncompressed block of precise values (PV) is shown at the top. The array of deltas, the base, and the approximation degree, α, all go into the compressed block. α controls the right shift amount for all subtraction unit input operands and the base. Through this precision scaling of operands, α controls the quality-energy tradeoff for our approximate storage technique. 49

Figure 3.5: Maximum and minimum values for a signed 1-byte delta (left) and the impact of right shifting on delta value and sign-bit extension (right). The value 129 cannot be represented by a 1-byte signed word, but after right shifting by 1 bit, it can be stored in 1-byte as 64. Upon decompression, the bits would be left shifted and the approximate value 128 would be returned. 50

Figure 3.6: Generic decompressor unit architecture with added left-shifters after addition. Specific implementations exist for all BΔ combinations from Table 3.2. The compressed block is shown at the top and the decompressed block of approximate values (AV) is shown at the bottom. Approximate degree, α, controls the left shift amount for all addition unit outputs.................... 51

Figure 3.7: Sample images from different categories of the Caltech 101 dataset [67]. Categories from left to right: airplanes, pizza, cougar_body, and electric_guitar. 51

Figure 3.8: Decompressed images for varying degrees of approximation for images from airplanes and pizza categories. 55

Figure 3.9: Average compression ratio with and without accounting for meta data across all 24 image categories for each level of approximation, α. The bar for α=0 without meta data represents the compression ratio of the lossless BΔI cache compression algorithm. 57

Figure 3.10: The average compression ratios (including metadata overhead) for all images from all categories and quality levels are shown. Each bar represents the average compression ratio over 40 images compressed with the specified degree of approximation. α = 0 represents lossless compression and has the lowest (sometimes less than 1.0) compression ratio............................ 58

Figure 3.11: Average compression ratio and average peak signal-to-noise (PSNR) for all images are shown as a function of α. α = 0 is excluded because PSNR is infinite when there is no loss between images.. 59

Figure 3.12: Average compression ratio and average structural similarity index (SSIM) for all images are shown as a function of α.. 59

Figure 3.13: Average compression ratio and average concept prediction probability for all images are shown as a function of α. 60

Figure 3.14: The average improvement in compression ratio and the average degradation in concept prediction probability for all images are shown as a function of α. 61

Figure 3.15: Average DRAM access energy savings are presented with (a) average compression ratio and (b) average concept prediction probability for all images as a function of image version. Baseline represents the original, uncompressed image and all other values represent the value α that the image was approximated with... 64

List of Tables

Table 1.1: Energy (pJ) per operation versus ‘Add’ in 45nm TSMC for compute and memory access. Attribution: (Pedram et al. 2017) [6]. 2

Table 1.2: Overview of recognition, mining, and synthesis applications. 3

Table 1.3: RMS applications where the majority of runtime is spent in error resilient kernels. Attribution: (Chippa et al. 2013) [1]. 6

Table 1.4: Common quality metrics used for applications/kernels in approximate computing and storage. Attribution: (S. Mittal. 2016) [3]. 11

Table 2.1: Taxonomy of approximate storage techniques. 12

Table 2.2: Error expectations, which are the expected deviation from original value for all points stored in LUT, and energy savings (E_s) for trigonometric, exponential, and logarithmic functions and their value domains using ApproxLUT, relative to normal computation. Attribution: (Tian et al. 2017) [49]. 40

Table 3.1: BΔI encoding table for 32-byte blocks. Except for zeros and repeating, Base8-Δ1 is the most space efficient encoding. Except for the case when compression isn’t possible, Base2-Δ1 is the least efficient space encoding. Sizes are in bytes. Attribution: (Pekhimenko et al. 2012) [43]. 47

Table 3.2: Approximated BΔI encoding table for 32-byte blocks. Similar to Table 3.1 except for the addition of the Meta data column. In consideration of main memory architectures where modifications to include meta data bits would be impractical, our work accounts for compression meta data overhead. Sizes are in bytes. 50

Table 3.3: Lossy compression and decompression hardware simulation metrics in 65nm TSMC. 62

Table 3.4: Lossy compression and decompression shifters area (mm^2) overhead in 65nm TSMC. 62

Table 3.5: Lossy compression and decompression shifters power (uW) overhead 65nm TSMC. 62

Table 3.6: Average DRAM access energy cost of storing the average original, BΔI compressed, and approximately stored images. DRAM access energy costs derived from Table 1.1 [6]. 63

Table 3.7: Average compression and decompression energy costs per image and relative to DRAM access energy cost per image. 63

1 Introduction

For a long time, Moore’s Law was fueled by Dennard scaling [27], where smaller, faster, and more energy efficient transistors were introduced with every technology generation. That was until about 2005 when single core processor performance could no longer improve in a cost-effective manner due to on-chip power and thermal limitations caused by increasing leakage and clock frequencies [28]. The microprocessor industry shifted to a multicore design paradigm where cores in parallel, operating at lower clocks speeds and consuming less power, could outperform high-end single core designs; the number of logical cores began increasing while frequency and power consumption stalled. For over a decade, multicore scaling has successfully carried Moore’s Law, increasing performance by parallelizing problems and running applications in parallel. Multicore Systems-on-a-Chip (SoC) with heterogeneous subsystems have enabled battery constrained mobile devices to achieve high performance and many-core architectures have enabled data center processors to meet massive computing demands with low latency requirements.

Multicore scaling performance and energy returns cannot continue forever though because applications cannot be parallelized enough to efficiently use the many available cores [29]. Multicore scaling allows designers to continue leveraging shrinking transistor sizes by using higher transistor counts to build more cores, which in turn improves performance; but if applications can no longer make use of those newly available cores, then performance stalls and adding more transistors no longer improves performance and hence is no longer cost effective. Another factor contributing to the decline of multicore scaling is the burden multicore processors place on the memory subsystem. Off-chip memory traffic generation scales directly with on-chip core count, but memory bandwidth has scaled at a much slower rate than core count, which has created a performance bottleneck in the memory interconnect [9, 86]. Even if multicore scaling could continuously improve on-chip performance, it would have a diminishing impact on total system performance because of the significantly higher latency and energy cost of off-chip memory access operations relative to compute operations, as shown in Table 1.1 [6]. DRAM access costs several more magnitudes of energy than on-chip add operations. Novel techniques will be needed to address this disparity between compute and memory performance.

Table 1.1: Energy (pJ) per operation versus ‘Add’ in 45nm TSMC for compute and memory access. Attribution: (Pedram et al. 2017) [6].

Operation	16 bit (integer)		64 bit (double precision)	
	Energy/Op (pJ)	vs. Add	Energy/Op (pJ)	vs. Add
Add	0.18	-	5	-
Multiply	0.62	3.4x	20	4x
16-word RF	0.12	0.7x	0.34	0.07x
64-word RF	0.23	1.3x	0.42	0.08x
4K-word SRAM	8	44x	26	5.2x
32K-word SRAM	11	61x	47	9.4x
DRAM	640	3556x	2560	512x

1.1 Data-Intensive Applications

The memory bottleneck is an important issue because of the emergence of data-intensive applications, which rely heavily on memory access. Data-intensive applications compute on very large sets of data, typically larger than on-chip cache caches [72], often in a streaming manner, memory accesses are frequent and are their limiting performance factor. Data-intensive applications handle multimedia (picture, video, audio, etc.) or perform complex modeling tasks in the domain of recognition, mining, and synthesis (RMS) [26, 30, 72]. Recognition, mining, and synthesis is the class of applications used to make computers understand data models, which enables them to solve abstract, hard to define problems. A popular area of RMS applications is in machine learning. Table 1.2 lays out a basic overview of RMS applications and some popular examples. Fields that stand to benefit from RMS applications include medicine, security, and finance, where automated analysis on large datasets can accelerate decision making and reduce costs.

To demonstrate the RMS processes, let us consider a simple case; we want an application that can cater to our tastes in music (some music streaming applications already do this to a degree). Using RMS, we could teach a computer what genre of music we enjoy listening through model training (recognition), then it could find music for us by searching for instances of that model (mining), and finally it could compose music for us by trying to build its own model (synthesis).

Table 1.2: Overview of recognition, mining, and synthesis applications.

Process	Question it Asks [26]	Modeling Relationship	Simple Case	Popular Examples
Recognition	"What is …?"	Learn a model	Learn music genre	Computer Vision
Mining	"Is it …?"	Find models	Find songs from genre	Web Search
Synthesis	"What if …?"	Create a model	Compose new songs	Data Analytics

Data-intensive RMS applications require high performance hardware substrates in order to process vast amounts of data. In machine learning, building a model involves training on many examples from dataset, such as 1.3 million examples for ImageNet [74]. Application areas include documents search, facial recognition, object detection, medical diagnosis, investment analysis, shopping recommendations, video game artificial intelligence (AI), and more. Conventionally, RMS applications running on desktops or embedded devices send data over a network to a data center, where the computing resources are plentiful, for processing. This is common because the process of transmitting data back and forth and processing in the data center is faster than processing locally or because the local device cannot meet the processing requirements at all. With the already widespread ubiquity of smartphones and the growing Internet of Things (IoT) [73], there is growing demand to implement RMS applications locally on embedded devices. For some embedded applications using RMS, it may be too costly or impossible to rely on data centers due to connectivity, bandwidth, security, and privacy concerns [5]. Autonomous vehicles for example need to quickly search video data and identify obstacles or pedestrians. The vehicle may have a poor or non-existent Internet connection due to environmental factors, and even if it did, it may take too long to transmit data to and from a data center because of the real time constraints of a moving vehicle.

So what is stopping RMS applications from reducing their reliance on data center processing? The primary design constraint for embedded RMS processing is energy consumption because mobile devices are limited by battery life. Embedded deployment of data-intensive multimedia and RMS applications calls for novel energy consumption reduction techniques. While multicore scaling has effectively improved computing capability under power constraints, it will not last forever. Fortunately, many media and RMS applications demonstrate an inherent resiliency against errors in computation, which can be exploited to improve energy efficiency and performance.

1.2 Approximate Computing and Storage

Since their inception, computers have always had one function; perform a specified set of calculations on a presented set of inputs and output the result. Anything less than an accurate answer was considered an error, which means the computer was either programmed incorrectly or there was a fault in its design, originating from eithers the user at design time or noise during its lifetime. One of the first tasks presented to computers was to calculate artillery firing tables for the military. 100% correct computation by the machine was critical because its results were used to aim weaponry; miscalculations could have devastating repercussions. If the computing machinery was prone to error, then its output results would be suspect and unacceptable. Correct computation and data integrity were essential for producing acceptable outputs.

Today, computers are used to solve a much wider variety of problems for a range of both critical and non-critical applications. Critical application examples include military hardware, banking systems, and airplane controls, which all require precise computation in order to produce acceptable outputs. Non-critical application examples include searching the web, streaming video, and home assistants, which can tolerate errors or imprecision in some of its computation and still produce outputs of acceptable quality. If a list of webpages is returned slightly out of order or the picture quality degrades during a video chat, there are no detrimental consequences, unlike in the artillery firing example. It's possible the user may not notice any application quality loss at all. If the output is acceptable for the user, then one can postulate that the underling computation doesn't always needs to be correct.

Li et al. [14] question whether architectural states (the underlying machine computations) need to be numerically perfect for program execution. If program output at the application level is "good enough" for the user, then does the underlying architectural correctness matter? Why bother computing a precise answer if an *approximate* answer will suffice? If some subset of the computations required to generate a solution don't matter, then system designers have whole new avenue of potential exploitations for improving computing performance and energy efficiency [77, 84]. These questions lead to a new design paradigm, *approximate computing and storage*, where quality, or data integrity, is traded off for improvement in energy consumption or performance. An application that produces acceptable output in the presence of computational errors is an *error resilient application*. If error is acceptable, then the quantity or precision of computations can be reduced to save energy or time. Approximate computing is similar to a predecessor, *soft computing* [31], which also seeks to exploit tolerance for imprecise computation with algorithms to lower solution costs. Approximate computing is a broader paradigm that encompasses circuits and architectures in addition to algorithms. Given the importance of the memory subsystem to performance of RMS applications, this thesis focuses its discussion on approximate storage.

Approximate storage exploits error resilience in circuits, architectures, and algorithms in the memory subsystem to trade off data integrity for energy through *quality-energy scaling* [10, 32, 78, 83, 85]. Applications that demonstrate *inherent error resilience*, such as those in RMS, are suitable for approximate storage because they can expect to see significant improvements in energy consumption or performance while maintaining application-level correctness. Approximate storage differs from fault tolerant design paradigms, which try to maintain architectural correctness by accounting for and responding to faults and errors, because it intentionally introduces error with the knowledge that the target application can still function properly. The defining characteristic of approximate computing and storage is that they purposefully trade off quality for energy. Before exploiting this quality-energy tradeoff though, the sources of inherent error resilience must be identified.

1.2.1 Inherent Error Resilience

Approximate computing and storage work because some applications are still able to produce acceptable or identical outputs in the presence of errors on input data or in computation. These applications demonstrate *inherent error resilience*, which is common in RMS applications [1, 14, 26]. This resilience applies only to non-critical program data, such as raw sensor data or pixels in an image; critical data, such as control path data or important variables, must be maintain perfect integrity or else programs risk failure due to irrecoverable program counter jumps or segmentation faults [10]. Given this restriction, approximate systems require separation of precise and approximate computing and storage elements in order to ensure applications can run without fail [36, 79, 82]. Since only non-critical data may be approximated, one may question if there enough non-critical data in an application to make approximation worth the effort; in Table 1.3, Chippa et al. [1] show that a set of 12 RMS applications spend 62-96% of their run-time in error-resilient kernels. This indicates that RMS applications spend a significant percentage of their time on computations and data that can be approximated.

Table 1.3: RMS applications where the majority of runtime is spent in error resilient kernels. Attribution: (Chippa et al. 2013) [1].

Application	Algorithm	% Runtime in Resilient Kernels	Dominant Kernel (Contribution to runtime)
Document Search	Semantic Search Index	90	Dot Product Computation (86)
Image Search	Feature Extraction	78	Dot Product Computation (71)
Hand Written Digit Classification	Support Vector Machines (SVM): Testing	94	Dot Product Computation (89)
Hand Written Digit Model Generation	Support Vector Machines (SVM): Training	97	Dot Product Computation (93)
Eye Detection	Generalized Learning Vector Quantization (GLVQ): Testing	89	Distance Computation (83)
Eye Model Generation	Generalized Learning Vector Quantization (GLVQ): Training	96	Distance Computation (92)
Image Segmentation	K-means Clustering	74	Distance Computation (66)
Census Data Modeling	Neural Networks: Multi-Layer Back Propagation	62	Matrix Vector Multiplication (42)
Census Data Classification	Neural Networks: Forward Propagation	79	Matrix Vector Multiplication (64)
Nutrition and Health Information Analysis	Logistic Regression	65	Dot Product Computation (48)
Digit Recognition	K-Nearest Neighbors	96	Distance Computation (92)
Online Data Clustering	Stream Cluster	77	Distance Computation (68)

Inherent error resilience can be attributed to three main sources: Noisy or redundant inputs, a range of acceptable outputs or outputs for human perception, and statistical or iterative computation patterns [1, 2, 4, 10, 14, 31, 34, 76]. Armed with this knowledge of inherent error resilience, designers can exploit sources of it at all layers of the design hierarchy to improve energy consumption and performance.

(1) Noisy or redundant inputs: We live in an inherently noisy and imprecise world. Sensors capture real world data as analog signals and most of these signals are converted to a digital representation for processing. Between noise at the sensor front-end and quantization during analog-to-digital conversion, some precision is lost in the data. Applications that process sensor data must have some level of noise tolerance so that slight variations in input data do not cause application failure. Hegde et al. [38] present one such early work that seeks to exploit noise-tolerance in digital signal processing (DSP). Sensor data may also be redundant. Consider a smart home speaker which constantly samples audio data with a microphone to determine if a user is speaking to it. Sampling audio data frequently when no one is speaking creates redundant data. An application can tolerate noise or imprecision in this data because the sample size is so large, that any noise will have minimal effect on the results of any processing done on that input data.

(2) A range of acceptable outputs or outputs for human perception: There lacks a single, perfect solution to a problem. Take a web search for example. A list of web pages for a given search query are returned in the "best" order, but there is no way to quantitatively define what the perfect result is for the user. There may be a best way to return page results based on algorithmic techniques such as page rankings, but this is only based on the provided search query. A search engine cannot presume to know exactly what the user is searching for. It's possible the user themselves were unsure of what they were looking for and they may just be browsing for an answer. Applications like web or document search that cannot output a perfect result can thus tolerate small errors or imprecisions since their output is not expected, nor can it, produce a perfect result. If a web search returns a list of pages in a slightly different order than the highest ranked page order, the user is unlikely to notice any difference. Generally, as long as the page they are searching for is found the first few results, they are satisfied. Another aspect of this source of error resilience is media applications that produce pictures, video, audio, etc. for human consumption. Since the human

senses are limited in their perception abilities, perfect media outputs need not be produced. The human eye may not be able to perceive the smallest details in an image (those represent by the least significant bits of a pixel). This is the motivating factor in many lossy media compression algorithms, such as the JPEG standard [33]. Figure 1.1 below shows a set of images where the three on the right compressed and decompressed with a hardware lossy compression algorithm, which will be discussed in Chapter 3. There is little to no perceivable difference in quality, demonstrating the exploitability of outputs for human perception.

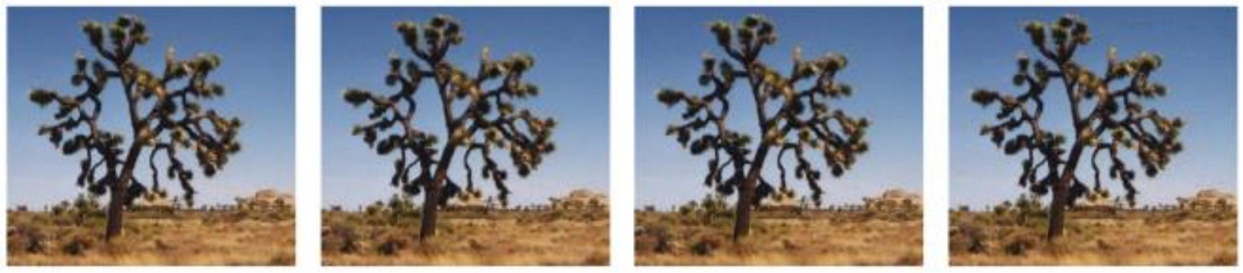


Figure 1.1: Original, high quality image (leftmost) and the same image after lossy compression with an increasing number of least-significant bits approximated.

(3) Statistical or iterative computation patterns: Computation patterns that rely on statistics and probability to find a solution or refine a solution over many iterations of a process to meet some threshold value or find some optimal solution before terminating. These computation patterns are tolerant of imprecise or inexact data because they rely on statistics, which introduces imprecision on its own, or iterations, which can recover over time. Statistical and computation patterns are commonly found in RMS applications because they work to understand, identify, and create abstract data models. Algorithms with this source of resilience include but are not limited to support vector machines (SVM), neural networks, K-means clustering, and Monte Carlo methods [1]. Soft computing [31] relies on these same sources of resilience.

1.2.2 Design Requirements

Once sources of inherent error resilience have been identified, computing and storage systems can be approximated for a specific application. An approximate computing or storage system must do three things: 1) Approximate data within some design abstraction. 2) Expose quality control knobs to the programmer or application designer that allow them to control the quality-energy tradeoff. 3) Monitor quality during runtime to ensure application-level outputs are acceptable, which also requires defining an application specific quality metric.

1) In order to implement approximate storage, data integrity must be lost somewhere. Approximation can be induced through noise in a physical medium or by a deterministic, algorithmic technique. Common approximation techniques include voltage overscaling at the circuit level [24, 25, 38] and precision scaling [18, 37] or loop perforation [35] at the algorithm level. Voltage overscaling focuses on reducing the supply voltage for SRAM or DSP circuits, which increases the probability of error in read or output data. Precision scaling, also known as bit truncation, reduces data precision by disregarding some number of least-significant bits (LSBs) in a data word. This reduces the quantity of data that needs to be transported or processed, saving energy. Loop perforation is a software technique where computational loops used to refine a solution are terminated early so that they only compute a subset of their original set of iterations. This early termination reduces accuracy of the solution, but saves time and resources by cutting out extra computations that are not necessary for reaching an acceptable solution. We will review state of the art approximate storage techniques in Chapter 2.

2) The programmer must be able to control approximation to make it practical to use. Approximation controls allows an application to adjust the quality-energy tradeoff to an appropriate level. Since error resiliency is dependent on program data, the application output quality may change during runtime. If quality loss is too high, then the application needs to reduce the degree of approximation. If quality loss is negligible, then the application could increase the degree of approximation to improve energy consumption. The approximate design must be controllable such that the quality-energy tradeoff can be dynamically tuned to application requirements during runtime. An ad hoc design that is stuck computing at some specific level of

approximation will have poor reusability. This is commonly done through language or ISA extensions to support approximate instructions [16, 17, 36] or hardware/software co-design, where the programmer defines functions to control approximate hardware through memory-mapped registers [18, 24]. There are two approaches to approximation control: binary control and multi-level control. Binary control allows the designer to approximate data or computation at some fixed degree or not approximate it at all (leave it precise). Flikker [8], a technique approximates by reducing DRAM refresh rates, demonstrates binary approximation control; main memory data can either be allocated to precise memory regions with guaranteed data integrity or to imprecise memory regions where faults are likely to occur. Multi-level control allows the designer to approximate data to various degrees (not just one fixed rate). Approximate load value predictors [21, 22], an architectural technique, demonstrate multi-level control by allowing the programmer to specify the rate at which they drop cache misses, which in turn reduces the accuracy of load value predictions and the accuracy of program data.

3) Quality monitoring is a necessary component of approximate systems so that they know when to adjust the quality-energy tradeoff in order to maintain acceptable application-level output quality. If output quality falls below a specified threshold, then computation is no longer useful and all spent energy has been for an unacceptable result. Quality monitoring requires a definition of “good enough” quality. Defining “good enough”, or acceptable, output quality is difficult because it is specific to the application domain. There is no universal metric for evaluating output quality because all applications do not produce some universal output type; the designer must determine the appropriate quality metric for their application and define a quality evaluation function for their system. Common quality metrics used when approximating RMS and media applications are shown in Table 1.4[3].

Table 1.4: Common quality metrics used for applications/kernels in approximate computing and storage. Attribution: (S. Mittal. 2016) [3].

Quality Metric(s)	Corresponding Applications/Kernels
Relative difference/error from standard output	Fluidanimate, blackscholes, swaptions (PARSEC), Barnes, water, Cholesky, LU (Splash2), vpr, parser (SPEC2000), Monte Carlo, sparse matrix multiplication, Jacobi, discrete Fourier transform, MapReduce programs (e.g., page rank, page length, project popularity, and so forth), forward/inverse kinematics for 2-joint arm, Newton-Raphson method for finding roots of a cubic polynomial, n-body simulation, adder, FIR filter, conjugate gradient
PSNR and SSIM	H.264 (SPEC2006), x264 (PARSEC), MPEG, JPEG, rayshade, image resizer, image smoothing, OpenGL games (e.g., Doom 3)
Pixel difference	Bodytrack (PARSEC), eon (SPEC2000), raytracer (Splash2), particle filter (Rodinia), volume rendering, Gaussian smoothing, mean filter, dynamic range compression, edge detection, raster image manipulation
Energy conservation across scenes	Physics-based simulation (e.g., collision detection, constraint solving)
Classification/clustering accuracy	Ferret, streamcluster (PARSEC), k-nearest neighbor, k-means clustering, generalized learning vector quantization (GLVQ), MLP, convolutional neural networks, support vector machines, digit classification
Correct/incorrect decisions	Image binarization, jmeint (triangle intersection detection), ZXing (visual bar code recognizer), finding Julia set fractals, jMonkeyEngine (game engine)
Ratio of error of initial and final guess	3D variable coefficient Helmholtz equation, image compression, 2D Poisson’s equation, preconditioned iterative solver
Ranking accuracy	Bing search, supervised semantic indexing (SSI) document search

1.3 Outline

In Chapter 2, we will survey the state of the art in approximate storage across circuits, architectures, and algorithms. In Chapter 3, we will present our own work on approximate storage, where we enable lossy compression through precision scaling of data. In Chapter 4 we draw conclusions.

2 State of the Art in Approximate Storage

Approximate storage is the process of trading off storage accuracy for improvements in energy consumption or performance; it can be applied algorithmically or physically and at any level of the memory hierarchy. The major research challenges in *approximate storage* are focused developing techniques for approximation, devising methods for controlling the quality-energy tradeoff, and defining and quantifying quality metrics. In this chapter, we review state of the art *approximate storage* techniques. Circuit, architecture, and algorithm level approximate storage techniques will be introduced and discussed. We highlight their approaches to 1) approximating data, 2) controlling approximation, and 3) monitoring approximation. Table 2.1 below is provided as quick reference for the various areas of approximate storage.

Table 2.1: Taxonomy of approximate storage techniques.

Area	**Techniques**	**References**
Circuits		
SRAM	Supply voltage scaling	[24, 25, 36, 51]
DRAM	Refresh rate scaling	[8, 11, 13]
Multi-Level Cells	Reliable access scaling	[15, 19]
STT-MRAM	Reliable access scaling	[53, 54, 56]
Architecture		
Support	ISA extensions, language extensions	[16, 17, 36]
Memory Load	Approximate value prediction	[20-23]
Cache	Spatio-similarity approximation	[42, 44, 56]
Algorithms		
Precision Scaling	Truncation	[18, 37, 56, 64]
Memoization/LUTs	Similar function reuse	[40, 49]

2.1 Circuit Level Storage Optimization

Approximate storage techniques at the circuit level include approximation of memory technologies such as SRAM, DRAM, and solid-state memories. All memory technologies require some type of "guard-banding" against manufacturing variations to ensure 100% data integrity [24]. Guard-band requirements exist in supply voltage for SRAM, refresh rate for DRAM, distance between cell levels in solid-state memories, and read/write effort in STT-MRAM. In this section, we will review approximate storage techniques for each of these technologies.

2.1.1 SRAM Supply Voltage Scaling

Static random access memory (SRAM) circuits are a popular target for approximation since they exist entirely on-chip in the form of register files or caches. SRAM approximation is based on two general techniques: voltage scaling and precision scaling. Voltage scaling involves reducing the supply voltage (V_{DD}) of SRAM arrays and precision scaling involves reducing the number of LSBs stored or supplied with nominal V_{DD}. Typically, voltage scaling can achieve quadratic energy savings, but at the cost of exponentially increasing error rates; and precision scaling can achieve linear energy savings for only linear quality loss [25, 36]. These works focus on exploiting voltage scaling to implement approximation in SRAM; they do not focus on the process of voltage scaling itself, which is left to voltage regulators.

Cho et al. [51] propose a reconfigurable, accuracy-aware dual-voltage SRAM architecture that scales down supply voltage (V_{DD}), bit-line precharge, and write voltage for LSBs and uses nominal values for MSBs. The LSBs will expend less energy during access, but are more susceptible to errors. LSBs are selected because they make the least significant impact on pixel values in image data. In this architecture, the number of voltage scaled LSBs can be reconfigured during runtime to meet application quality requirements. In their accuracy-aware SRAM architecture, shown in Figure 2.1, same significance bit cells of different words, a MUX group (in column multiplexing SRAM terminology), share the same voltage level, which is separate from other MUX groups. The number of bits in a MUX group is referred to as the reconfiguration length (RL), which determines the bit-granularity that the quality-energy tradeoff can be controlled at. A MUX group's power supply rail is either connected to a nominal, V_{high}, or scaled, V_{low}, supply voltage through two PMOS transistors, controlled by a select bit from a quantization control word. The power rail supplies voltage for the SRAM bit cells, precharge logic, and a reconfiguration inverter that drives the local word line for all bit-cells in a MUX group. The reconfiguration inverter is driven by a global word line that is powered by nominal V_{DD} comes from decoding logic. They note that the reconfiguration inverters incur an area overhead of 6% for the core SRAM array because one is needed for every MUX group. By appropriately setting the quantization word that controls the power supply PMOS transistors, the voltage can be set to V_{high} or V_{low} respectively for a set of MUX groups. This dynamic control allows a system to specify any number of LSBs for a given address in SRAM to be approximated.

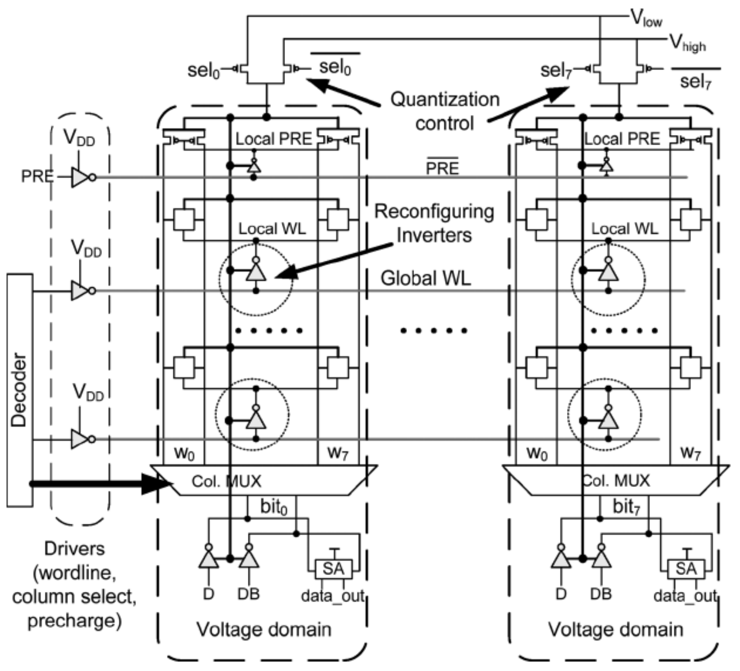


Figure 2.1: Accuracy-aware, dual-voltage SRAM array. The quantization control signals select either V_{high} or V_{low} as the supply voltage for each highlight voltage domain. Attribution: (Cho et al. 2015) [51].

Cho et al. evaluate this work in simulation with 8-bit grayscale image data and use Mean Structural Similarity (MSSIM) as their quality metric. Compared to a normal SRAM array, they found that by setting the number of voltage scaled LSBs = 4 and Vlow = 0.4V, they could achieve mean power savings of 45% for 10% reduction in MSSIM quality. While keeping quality loss to 10%, voltage scaling all bits in a word actually achieves 20% less power savings than only 4 LSBs, which shows the efficacy of selectively voltage scaling only a subset of all bits.

Esmaeilzadeh et al. [36] propose a dual-voltage SRAM for approximate and precise data storage (using voltage scaling) as part of their Truffle microarchitecture and set of ISA extensions for approximate computing and storage. Their ISA extension requires that every instruction operating on register data has to include an extra precision control bit, which dictates whether the data should be treated as precise or approximate. The Truffle microarchitecture's 6-transistor SRAM (used for register files and caches) uses two voltage lines: $V_{DD}H$ (high) for reliable operation on precise data and $V_{DD}L$ (low) for energy-efficient operation on approximate data. Their SRAM array architecture is illustrated in Figure 2.2. $V_{DD}H$ is the typical V_{DD} used for a given SRAM (1.5V in this work) and $V_{DD}L$ can either be set statically at design or dynamically changed during runtime using on-chip voltage regulators [50]. The SRAM instruction control structures use only $V_{DD}H$ to ensure, whereas memory access structures (i.e. precharge logic, bit cells, sense amplifiers) can switch between $V_{DD}H$ or $V_{DD}L$, depending on an instruction's precision control bit. The SRAM has a precision column which store the voltage state ($V_{DD}H$ or $V_{DD}L$) for every row, which drives the power lines for all of the SRAM access structures. The voltage state is controlled by the precision control bit and is set prior to address decoding to ensure that the correct voltage level is set for the SRAM rows prior to accessing a cell. Reducing V_{DD} for the access structures reduces energy consumption, but at the cost of read/write reliability. In their evaluations of the Truffle microarchitecture, they set $V_{DD}H = 1.5V$ and swept $V_{DD}L$ across 0.75V, 0.94V, 1.13V, and 1.31V. Since their whole microarchitecture introduces approximation in more constructs than just SRAM, they do not report specific energy savings or quality impact attributed to it. They do however report the percentage of total core energy consumed by different components, including the register file and data cache, for Truffle. The register file and data cache are shown to consume less than 5% and about 10% of total core energy for an in-order execution version of Truffle. Evaluation of the in-order Truffle microarchitecture for the same set of benchmarks used in EnerJ [16], show average energy savings ranging from about 8% to about 25%, increasing with lower $V_{DD}L$. To evaluate quality impact, they model an error distribution for approximate components and inject errors into each benchmark. They find that overall application output error is largely a function of registers, cache, and floating-point units (FPUs).

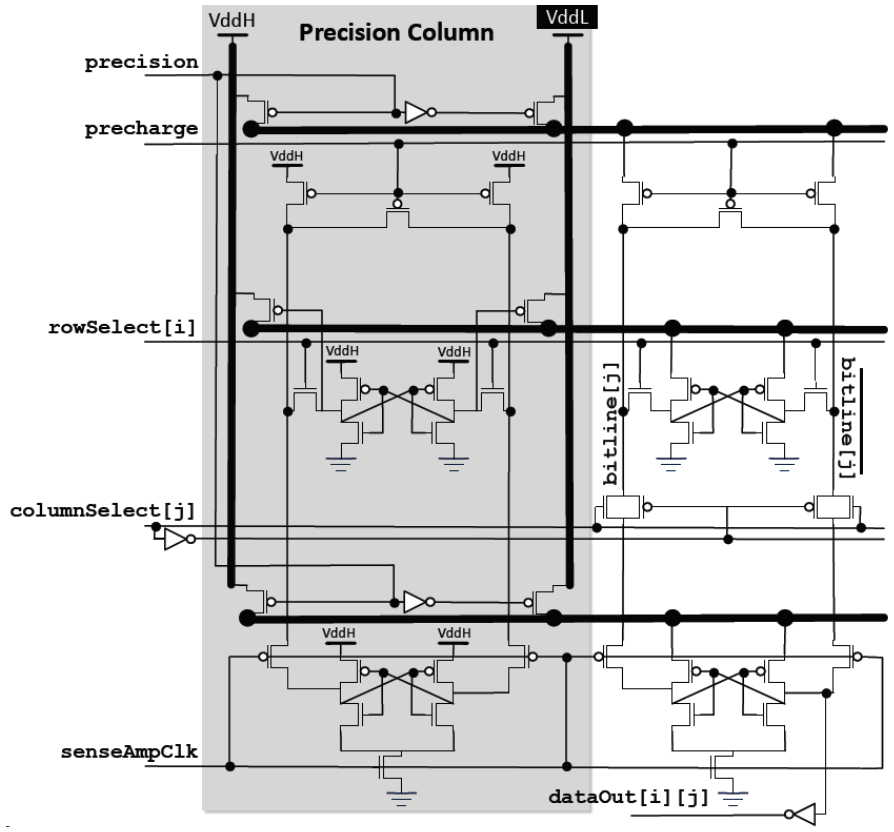


Figure 2.2: Dual-voltage SRAM array. Precision signal controls whether $V_{DD}L$ or $V_{DD}H$ is used as supply voltage for each row's precharge logic, bit cells, and sense amplifiers. Attribution: (Esmaeilzadeh et al. 2012) [36].

In order to improve the energy efficiency and speed of memory accesses and the lifetime of memory modules, Shoushtari et al. [24] propose a partially-forgetful SRAM cache which scales down supply voltage (V_{DD}) to reduce leakage energy consumption and can be dynamically controlled by the programmer through a run-time controller. The programmer can steer the degree of approximation using two control knobs for the cache: V_{DD} scaling and the number of acceptable faulty bits (AFB) per cache block. V_{DD} scaling reduces the leakage of SRAM, but also increases fault probability. Using built-in self-test (BIST), the cache can detect the number of faults in each cache block. The programmer can then use AFB to control which blocks are allowed to be use for approximate storage; this way the programmer has more fine-grained control over approximation degree (instead of just storing in blocks with any number of faults). The drawback to AFB is that blocks can no longer can be used which shrinks the effective cache capacity. The cache controller contains a non-criticality table which tracks the approximate memory address regions. The cache still contains a number of protected blocks, for which V_{DD} is not scaled. Data which does not fall into the approximate memory regions will be stored in these protected blocks. They evaluate their partially-forgetful cache in a CPU system simulator with a set of multimedia benchmarks and use PSNR and fraction of true positive detections as quality metrics for image and edge detection benchmarks respectively. They define 28dB as an acceptable SNR for their image benchmark and 32dB for video, which are in the same range as the defined acceptable SNR in [40]. Results show their cache can decrease leakage energy by up to 74% while maintain acceptable quality for image resizing and recognition benchmarks. They however do not discuss the system performance penalties due to disabling cache blocks based on the AFB value.

2.1.2 DRAM Refresh Rate Reduction

One popular approximate storage technique targets the dynamic random access memory (DRAM) subsystem for power savings. DRAM banks require periodic refreshes when in idle mode to retain data. DRAM memory cells consist of a transistor gate and a capacitor which holds the memory value. Overtime these capacitors discharge and require a periodic refresh to maintain their state. Since capacitor discharge time is dependent on manufacturing variations, the DRAM system refresh rate must be set high enough to prevent the fastest discharging cell from losing its value. Refresh rate periods in smartphone DRAMs are typically 64ms or 32ms. Since smartphones spend a lot of time “off”, or siting in a user’s pocket, this idle mode can last for long periods of time. Later when a user turns their smartphone back “on”, they expect all data and applications to still be there from before, which is why DRAM isn’t shut off completely.

Liu et al. [8] was the first to observe that by partitioning program data into critical and non-critical regions, power could be saved by allocating those data to normal DRAM banks with typical idle mode refresh rates and banks where the refresh rate has been lowered respectively. The normal banks guarantee data integrity, whereas the lowered refresh rate banks introduce the possibility of faults in data. They propose a software method, Flikker, for reducing DRAM power consumption that reduces refresh rate for specified banks. These banks are illustrated in Figure 2.3. They identify critical data as data structures and other variables or structures essential to control flow and non-critical data as media or other data that is being processed and output by the program. They contend that annotating program data as non-critical is not an overly time consuming effort for programmers. During runtime, Flikker allocates data to the appropriate set of DRAM banks based on specified criticality. They run experiments with smartphone applications, such as *mpeg2* for video and *rayshade* for 3D apps, in a cycle accurate architectural simulator. During active DRAM mode (when the smartphone is "on"), they observed negligible changes in performance and power consumption; then during idle mode (the targeted domain of this idea), they observed up to 25% power savings in DRAM, while maintaining less than 1% performance degradation and zero application failure.

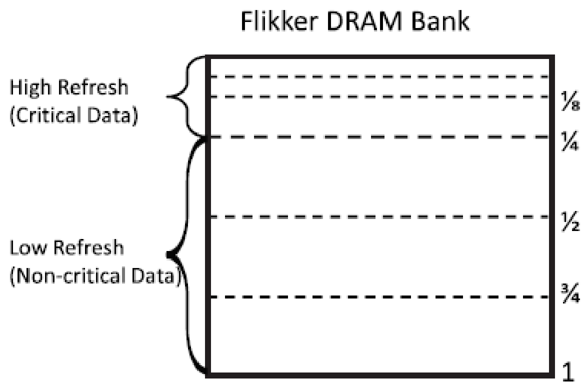


Figure 2.3: Flikker partitions DRAM banks into two regions: high refresh for critical data and low refresh for non-critical data. The partition boundary can be assigned at any of the dashed lines. Attribution: (Liu et al. 2011) [8].

Lucas et al. [11] propose Sparkk, an extension of Flikker that leverages observation about varying minimum required refresh rates from RAIDR [12]. While Flikker performs a binary allocation of data into approximate or non-approximate partition, Sparkk takes it a step further by breaking the approximate partition into ranks of decreasing refresh rates, hence introducing greater possibility of error due to bit cell discharging. Through permutation of data bytes across DRAM chips, higher order data bits are stored in higher quality bit cells (where refresh rate is higher) and lower order data bits are stored in lower quality bit cells (where refresh rate is lower). Since lower order bits will have less impact on data quality, they observe that these bits can be store with greater chance of error to save power. Through an approximate storage simulation on image data, they show that Sparkk is able to maintain higher image quality that Flikker when breaking up the approximate DRAM partition into at least two ranks with decreasing refresh rates. They observe that Sparkk can achieve the same image quality with less than half the average refresh rate of Flikker. The authors do not perform any real world evaluation of Sparkk and do not provide power measurements from real world or simulation. Since Sparkk requires multiple refresh rates for DRAM, significant modifications must be made to hardware to implement it. Flikker's implementation is much simpler (software modifications and an extension of the DRAM refresh counter). The strength of this work lies not in its experimental evaluation, but the novelty of breaking up the approximate memory region into varying levels of reliability.

Raha et al. [13] propose a quality-aware and configurable approximate DRAM, which takes into account DRAM error characteristics before allocating memory pages to a number of *quality bins*. DRAM is characterized by writing the whole DRAM and reading the values back for different refresh rate values. They record parameters such as frequency, significance, and type of bit-flips to build a profile of every DRAM page. They then rank the DRAM pages by any of the following strategies: (1) word errors per page, (2) bit errors per page, (3) weighted bit errors per page, or (4) strategy 2 with preference for either 0->1 or 1->0 bit flips. They characterize 8 different commercial DRAMs in their experiments. After characterization, a single refresh rate for the whole DRAM is set at a rate to ensure the first quality bin is large enough to store all critical application data in error free pages. Non-critical data pages are then allocated to the remaining quality bins in ascending order of error rate. By setting the refresh rate high enough to protect critical pages and allocating non-critical data to bins of ascending error rate, they use the minimum necessary refresh rate to run an application, which minimizes power consumption, and they are able to reduce quality degradation. The performance and energy overhead of allocating pages into quality bins is negligible. In experimental evaluation, the authors show 73% refresh power reduction for when half of DRAM is assumed to contain critical data, which is a 2.4x improvement in refresh power reduction over Flikker. They also setup a smart camera system as a case study and show that from a refresh rate power reduction of 73%, total system (compute, memory, and sensor) energy is reduced by up 33% with demonstrated minimal quality loss. Since this work requires only one

DRAM refresh rate, implementation is simple relative to Sparkk, which requires significant hardware changes to support multiple refresh rates.

2.1.3 Multi-Level Cell Storage

Another area of approximate storage is in the area of solid-state drives (SSDs), which includes flash memories and emerging phase-change memories (PCM). SSD memory can be implemented as single-level cells (SLC) or multi-level cells (MLC), which store 1 or 2+ bits per memory cell respectively. Data is written into a cell as an analog value (charge or resistance) and then is quantized upon read to get a digital value. MLCs require guard bands between levels to ensure accurate analog access operations. This results in shorter target ranges for storage values, which makes writing values in tedious and expensive. SSDs are subject to soft (transient) errors, over time due to non-ideal charge and resistance characteristics and typically require error correction codes (ECC) to maintain data integrity. In this section we will discuss approximate storage techniques for SSDs that target memory cell access operations and ECC requirements.

Sampson et al. [15] proposes two approximate storage techniques for MLC memories: more efficient writes and lifetime extension. This work focuses on phase-change memory, but the concepts also apply to flash memory. In the first technique they propose a model for an approximate MLC write, which improves the performance and energy consumption of MLC writes relative to precise MLC by trading off write accuracy. They exploit imprecise nature of writing to an analog MLC by relaxing the guard band requirements used for defining distinct levels within a cell. Normal and relaxed guard bands for an MLC are illustrated in Figure 2.4. This reduces the number of program-and-verify (P&V) iterations needed to write an MLC. P&V is a typical technique used for ensure an MLC has been written correctly within the target (analog) value range. Since approximate MLC relaxes the guard band requirements for a cell, it statistically takes fewer attempts to correctly write a cell, which improves performance and saves energy. The drawback is that relaxed guard bands leads to greater probability of soft errors occurring.

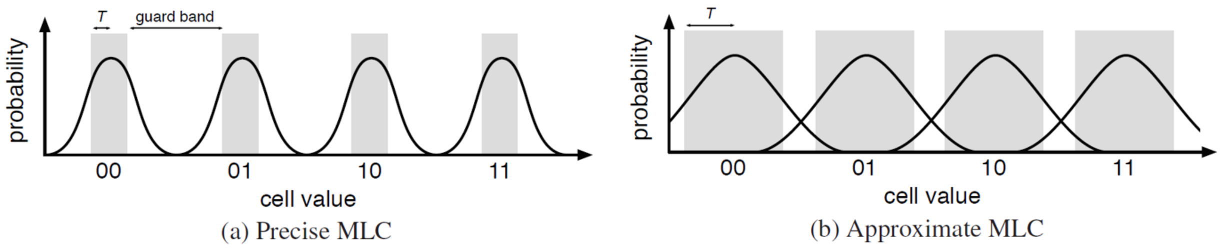


Figure 2.4: The range of analog values in (a) precise and (b) approximate MLC. The shaded regions represent target levels and the unshaded areas represent guard bands. Each curve represents the probability of reading a given value from the cell. In the approximate cell (b), the reduced guard bands lead to an overlap in possible read values where erroneous values may be read. Attribution: (Sampson et al. 2013) [15].

Their second technique proposes extending SSD lifetime by using failed memory blocks to store approximate data. Overtime MLCs wear out and are subject to hard (permanent) errors. If the number of MLC failures in a block exceed ECC capabilities, then the block is abandoned by the controlling system. If enough blocks fail in an SSD, then the drive must be replaced. They propose extending the life of failed blocks by allocating correction resources, namely error-correcting pointers (ECP), to more significant bits, while leaving less significant bits uncorrected. By modifying an operating system's (OS) memory allocator, they can specify that these failed blocks are now for approximate data storage only. The authors simulate their techniques for main-memory applications and persistent storage for a variety of applications, such as image renderer and scientific kernels, and datasets, such as sensor logs and image bitmaps, respectively. Their main-memory evaluation uses annotated benchmarks from EnerJ [16]. In the most error-tolerant application, they observe a 1.77x on average write speedup of approximate MLC over precise MLC with less than 2% quality loss. For the least error-tolerant application, they observe a 1.24x average write speedup with 4% quality loss. For the persistent datasets they observe a 1.7x write speedup with less than 10% quality loss. Regarding SSD lifetime extension, they observe a mean extension of 18% for main-memory applications and 36% for persistent storage with quality loss limited to 10%.

Xu et al. [19] propose SoftFlash for removing strong ECC SSD requirements for applications with inherent error resilience in order to reduce non-trivial ECC overhead at the cost of application-level quality output. As mentioned prior, SSDs are prone to variety of soft errors and use ECC to ensure 100% reliable operation but at the cost of significant energy, performance, and area overhead. In an evaluation of four common ECCs (Hamming, Reed-Solomon (RS), Bose-Chaudhuri-Hocquenghem (BCH), and Low-Density Parity-Check (LDPC)), they find that they all add 20% overhead to memory read performance and an energy overhead of at least 200% for a set of benchmarks with different memory access patterns. SoftFlash works by estimating the error rate of the SSD, identifying and providing application quality requirements for the SSD, and storing data with the appropriate ECC strength (potentially none). They evaluate a set of data-intensive applications on flash storage with faults injected at rate representative of real flash memory bit error rates (BER). SoftFlash has a low implementation cost because it only requires changes to the OS and SSD firmware.

2.1.4 STT-MRAM Access Reliability

One area of emerging memories is Spin Transfer Torque Magnetic Random Access Memory (STT-MRAM), also known as *spintronic memory*, which offer high capacity, high performance, non-volatility, and low cost [52]. The primary drawbacks to STT-MRAM though are the high energy cost of performing reads and writes and retention failures due to thermal scaling limitations [53, 54, 57]. STT-MRAM represents logic values as spin orientation in a ferromagnetic material. A typical STT-MRAM bit cell consists of an access transistor and a magnetic tunnel junction (MTJ), as shown in Figure 2.5. Within the MTJ, the relative spin orientation of its layers determines stored logic value. To read a cell, a voltage is induced across the access transistor and MTJ and the resulting MTJ current is compared against a reference current to determine the value. To write a cell, a current large enough to switch the MTJ's relative orientation is applied to the MTJ through one direction or the other, depending on the value being written in.

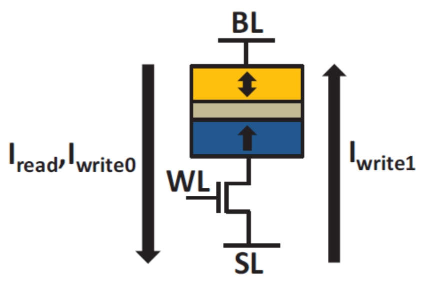


Figure 2.5: STT-MRAM bit cell consisting of a magnetic tunnel junction (MTJ) and an access transistor. Attribution: (Ranjan et al. 2017) [56].

Ranjan et al. [53] seek to improve the energy efficiency of STT-MRAM by approximating the read and write access mechanisms and propose a QCMem, a quality configurable memory array, that serves as a scratchpad memory. They approximate reads by lowering the read voltage that is applied across the MTJ, which lowers the current and increases the probability of a read error. They approximate writes by lowering the write pulse duration (length of time write current is applied) to below the average time it takes for an MTJ bit cell to switch, which increases the probability the bit cell is not properly switched. These approximation techniques reduce energy consumption by decreasing the amount of current necessary to read and the length of time a write current needs to be applied. These approximate STT-MRAM access techniques are applied within QCMem, which includes a quality decoder module for controlling the quality level of individual RAM columns. They integrate QCMem within a vector processor architecture [55] that includes precise and approximate processing elements and add quality fields to the vector load/store instructions so quality level may be specified. These load/store instructions communicate to the QCMem quality decoder the level of quality required for the STT-MRAM array. This cross-layer interaction demonstrates how hardware quality control knobs can be exposed to software controls. In their evaluation they use SPICE-compatible MTJ models to simulate and characterize the QCMem energy and performance metrics. They then simulate the vector processor with a 1MB sized QCMem in a cycle-accurate simulator and run a set of RMS benchmark applications from [1] and use classification accuracy as the quality metric. Compared to a precise STT-MRAM array, they find that QCMem reduces application energy consumption by 19.5% and 28% for under 0.5% and 7.5% quality loss respectively. Within the memory subsystem they find a 40% reduction in memory access energy consumption with write and ready energy efficiency improving by 1.76x and 1.48x respectively.

In further work Ranjan et al. [56] propose STAxCache, an L2 approximate spintronic cache with a mix of algorithmic and error-tolerance approaches to approximating STT-MRAM. These approaches include: 1) LSB read skipping, 2) lowering read current, 3) write skipping, 4) lowering write duration, and 5) bit-cell refresh skipping. The STAxCache is broken up into ways of varying quality guarantees that are managed by a quality table and quality controller, similar to the approach in QuARK [54], a concurrent work on STT-MRAM caches. To summarize each method: 1) Read energy can be saved by gating the bit lines to LSB cells whose values are insignificant to computation and return zeros instead, which is an intentional quality reduction. 2) The bit-cell read current can be lowered to save energy at the cost of increased probability of read failure, which relies on the MTJ medium to induce error. 3) Skipping writes to cache blocks when the difference between the old and new block falls within a specified quality constraint to save write energy, which is an intentional quality reduction. 4) Decrease bit cell write duration, which saves energy by applying current for a shorter period of time, but increases the probability that the MTJ fails to switch, which relies on the MTJ medium to induce error. 5) Reduce the rate of nominally required refreshes for low-quality ways, which relies on the MTJ medium to induce error. In a similar manner to many other works [8, 18, 24], STAxCache exposes hardware quality control knobs to the programmer to specify safe-to-approximate data by adding a new instruction to the ISA which specifies approximate memory regions and their quality requirements. This new instruction updates STAxCache's quality table, which the quality controller uses to regulate cache access quality. In their evaluation of STAxCache, the authors characterize a SPICE model of an STT-MRAM bit-cell, which they use for a model of STAxCache that is implemented as the L2 cache in the gem5 architectural simulator [58]. They evaluate STAxCache with several RMS benchmark applications and quality metrics from [1]. They find that STAxCache compared to an accurate STT-MRAM cache improves average energy consumption by 1.44x and 1.93x for 0.5% and 3.5% maximum quality degradation respectively. They also find that STAxCache improves average write and read energy efficiency by 1.56x and 1.03x for 0.5% maximum quality loss respectively.

2.2 Architecture Level Storage Optimization

At the architectural level, approximate storage ignores the common norms for maintaining data and program integrity. Approximation techniques typically focus on mitigating performance degradation from long memory access latencies. We will review ISA support techniques, approximations that exploit load value prediction, and approaches that improve the effective storage capacity of the cache.

2.2.1 Architecture and Programming Support

Many works in approximate storage focus on how to exploit accuracy at a particular level of the design hierarchy to improve energy consumption and performance, but do not address the issue of how programmers can effectively utilize these approximation techniques. Challenges in effectively implementing approximate storage deal with quality assurance and crossing layers of design abstraction. Application designers need ways to ensure an acceptable level of application output quality, or quality of service (QoS) when operating on approximate hardware and need to be able to program approximate systems without wasting time navigating design abstraction boundaries themselves [16, 17]. Fortunately, researchers have presented several architectural support techniques for programmers to implement approximate storage, which will be discussed in this section.

Baek et al. [17] observe that many approximation techniques used by programmers are ad-hoc and do not provide QoS guarantees. To address this issue, they propose Green, a framework that provides programmers seeking energy/performance improvements support for approximate programming and meeting QoS goals. Green allows programmers to approximate functions by substituting precise functions with user-defined approximate versions and loops through early termination. Green generates QoS model using a calibration version of a target program and compiles the target program that was annotated with Green extension to create an approximate program. Green monitors the program during runtime and is able to recalibrate the approximate functions and loops in case QoS targets are not being met. This recalibration scheme also ensure that the errors introduced by individual approximations do not compound into errors that are detrimental to QoS. The authors evaluated Green for performance, energy consumption, and QoS on a desktop and server machines with a back-end web search engine and other applications and benchmarks. In cases where quality loss was negligible, they observed over 20% energy consumption and performance improvement, and in cases where quality loss was less than 10% they saw up to 50% improvements in energy consumption and performance (metrics relative to precise implementations).

Sampson et al. [16] provide instruction set architecture (ISA) support for approximate computing and storage substrates through EnerJ, a Java language extension that introduces type qualifiers to distinguish *approximate* and *precise* program data. EnerJ does not define or implement any approximation technique itself, but provides support for programmers developing applications on approximate systems or hardware. EnerJ approximations won't save energy or improve performance if there is no underlying approximate system or hardware to accompany it. EnerJ uses type annotations, endorsements, contextual data types and other language extensions to ensure approximate and precise data are insulated from each other unless explicitly connected by the programmer. For instance, it is illegal to assign an approximate value to a precise value unless the programmer explicitly endorses the assignment. This allows programmers to easily protect and isolate precise control-flow and critical application data from approximate data. Explicit control over the influence of approximate data on precise data allows programmers to write approximate programs that are guaranteed to not crash unless they have failed to safely control their approximate data. Additionally, values are precise by default, which ensures that no approximate values exist without the programmer's explicit declaration. The authors evaluate EnerJ on an approximation-aware hardware model simulation which includes precise and approximate registers, functional units (FUs), SRAM cache, and DRAM for several approximate compute and storage strategies: FU voltage scaling, DRAM refresh [8], floating-point (FP) mantissa truncation, and low SRAM supply voltage. Recall that the focus of EnerJ is providing ISA support for approximate and precise data types; these approximation techniques are used to demonstrate the capabilities of EnerJ to work with any approximate hardware. They annotate a variety of benchmark kernels with EnerJ extensions to enable the approximate compute and storage techniques and run them through the simulator. They observe energy savings between 9% and

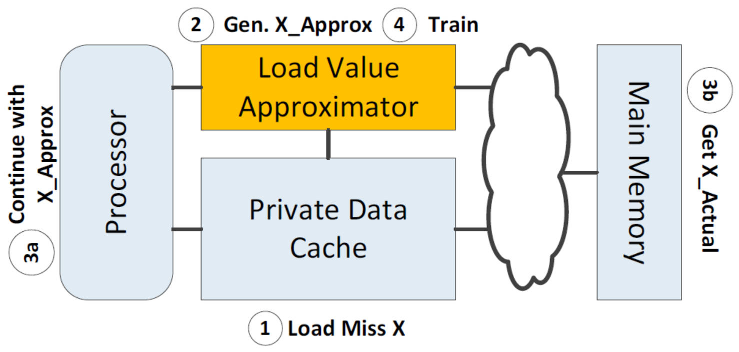
48% relative to precise only benchmark simulations. They also make notes about the effect of each approximation technique on application output, which they had to define on their own due to the difficulty of finding a universal “acceptable quality” metric [14]. Errors from the DRAM refresh and FP mantissa truncation techniques had near negligible impacts on. Errors from the SRAM supply voltage and FU voltage scaling techniques had significant impacts on output quality. Finally, the authors note that making EnerJ annotations to programs was not labor intensive and they were able to do it with unfamiliar benchmark code bases, which shows that EnerJ can easily be used to support any approximate system or hardware. EnerJ has proven to be a very useful tool as many of the works discussed in this review have made use of it.

2.2.2 Memory Load Skipping and Prediction

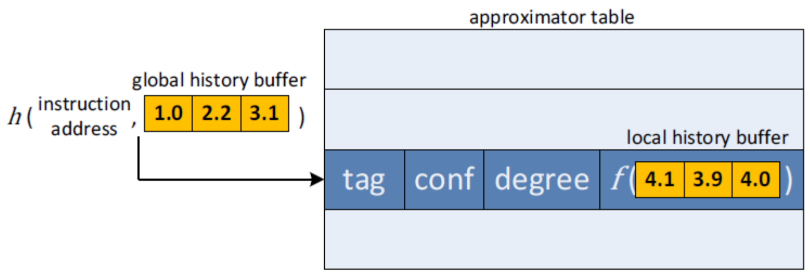
When a program’s load instruction inevitably misses in the L1 cache, then data must be fetched from a higher level cache or main memory. Compared to L1 cache access, higher level on-chip caches take longer to access and off-chip main memory takes orders of magnitude longer to access. These long access latencies can stall processors and limit performance, which can be detrimental to data-intensive applications that frequently access main memory. Load value prediction is an architectural technique that attempts to predict the value of a load when a cache misses. It immediately supplies a predicted value to the processor, then checks if the prediction was correct when the load operation returns data to the predictor. If a loaded value arrives from memory and it does not match the predicted value, then a hazard has occurred and the processor must rollback to that load instruction to correct the value. Thwaites et al. [20] and Miguel et al. [22] both observed that load value prediction can be approximated to improve performance and energy consumption because error-resilient applications can tolerate small errors in predicted load values. This technique implements approximation at single instruction granularity.

Thwaites et al. [20-21] proposed Rollback-Free Value Prediction (RFVP) to reduce the number of processor roll-backs and to free up memory bandwidth on cache misses for error-resilient applications. RFVP accomplishes this by predicting when loads are safe-to-approximate, then intercepting and dropping a percentage of cache misses bound for memory. Note that a cache miss event is what triggers RFVP; the purpose is to avoid long latency memory accesses when a cache misses. Approximate loads can tolerate differences between prediction and actual value, thus eliminating the need for processor rollbacks when prediction is wrong. Dropping blocks (or cache-lines) on cache miss reduces the number of memory requests, thus freeing up memory bandwidth. The rate at which RFVP drops blocks serves as an approximation control knob for the quality-energy tradeoff; dropping more blocks saves energy by reducing data traffic, but also increases error because the load predictor does not receive load data to train off of for improving the accuracy of future predictions. Their approximate value predictor is based off a (precise) two-delta predictor design. They extend an ISA to support an approximate load instruction and an instruction for controlling the drop rate for blocks. They also use programmer annotations, similar to the work in EnerJ [16], to mark safe-to-approximate data because some application data is too critical to approximate [10]. They evaluate RFVP in CPU [20] and GPU [21] simulators. For CPU, they evaluate a set of SPEC benchmarks and observe a mean performance improvement of 8.1% with an average quality loss of 0.8%, where quality loss is defined by the root mean square error (RMSE) between precise and approximate program output. For GPU, they evaluate image and other GPU benchmarks and observe an average performance improvement of 36% and energy reduction of 27% for 10% quality loss, where quality loss is defined by RMSE, average displacement, and mismatch rate depending on the benchmark. Additionally, for only 5% quality loss, they observe an average performance improvement of 16% and energy reduction of 14%.

Miguel et al. [22] proposed Load Value Approximation (LVA), a concurrently developed technique similar to RFVP, which estimates memory load values on cache miss to reduce load latency and energy consumption for error-resilient applications. On cache miss, LVA predicts the value that will be loaded based on recent global and local load data, then supplies that value to the processor so it can continue executing instructions without waiting for the value to load all the way from memory. LVA also eliminates the need for rollbacks since a mismatch between the approximately predicted value and the actual load value is acceptable. They note that floating-point values are an ideal candidate for LVA because very small differences in the bits of the mantissa would cause a conventional load predictor to trigger a rollback, whereas error-resilient applications can tolerate those small differences. LVA predicts values using global and local load instruction contexts. Its design is illustrated in Figure 2.6. A global history buffer tracks all recent loads and serves as an index for an approximator table of local history buffers. Each local history buffer tracks loads for a given tag, which is created from a hash of global history buffer values and load instruction address. An approximate load value is generated from some function of the values in the local history buffer (i.e. average).



(a)



(b)

Figure 2.6: Load value approximation (LVA) architecture overview (a) and design (b). The highlighted box in (a) represents the design in (b). Upon a load miss X in the cache (a.1), the Load Value Approximator generates an approximate X value (a.2), which the processor will compute with (a.3a). The missed load X is still fetched from main memory (a.3b) for the approximator in order to improve the accuracy (a.4) of future load approximations. The load value approximator (b) uses context from the load instruction address and a global history buffer to select an entry in the approximator table, where it will then select a load value based on a local history buffer. Attribution: (Miguel et al. 2014) [22].

LVA implements an approximation degree control knob for quality-energy similar to RFVP's; the approximation degree controls how many times LVA drop blocks (or cache-lines) before loading one to further train the load predictor. Every block that does not make a memory request saves energy, but also skips an opportunity to train the load predictor, which potentially increases load value error. The load predictor relies on real load values to adjust its global and local history buffers in order to provide more accurate value predictions. In their evaluations, they use EnerJ [16] to annotate data that is safe to approximate for LVA. In their simulations, they evaluate a set of PARSEC 3.0 applications on a multi-processor system. They find that LVA improves application performance by 8.5% on average for approximation degrees of 0, 4, and 16, while maintaining around 10% application output error. They also find energy reductions of 7.2% and 12.6% for degrees 4 and 16. There are nearly zero energy saving for approximation degree 0 because it does not drop any blocks, meaning it makes a typical amount of memory requests.

In another approach, Kislal et al. [23] propose exploiting the inherent error resilience of decision tree learning algorithms (DTL) by skipping some main memory accesses on cache miss with a data access skipping module (DASM). Their work focuses on DTL algorithms that are common in data-intensive mining applications. By reducing the number of loads from main memory (long latency) performed after L2 cache misses, they can significantly improve the performance of DTL based applications for low quality loss. In a small experiment, they show that randomly skipping 1-5% of DTL data points results in only worst case 8% accuracy loss, which indicates that a small percentage of application data can be skipped without detrimental effects on accuracy and that more skips results in lower accuracy. They also note that data access latency depends on where the data resides in the memory hierarchy, which gives it a variable latency. Costly accesses miss in the LLC and cheap accesses hit at on-chip caches. Since accuracy depends on the quantity of data skips and performance depends on the cost of data access, they posit that they can achieve improved significantly improved performance for low quality loss by selectively skipping a few costly memory accesses. Using a skip ratio, their framework will randomly skip memory load requests to make sure data points are not repeatedly skipped across iterations of the DTL algorithm. The DASM is hardware add-on to the cache controller which can intercept memory access requests at a rate determined by the skip ratio. When a request is not skipped, the load request is sent to memory as normal then returned to the processor. When a request is skipped, the DASM will return Not-a-Number (NaN) to the processor (as opposed to the load value) and leave it to the programmer to replace the missing load value. The authors propose a DTL last-point heuristic prediction technique, which essentially uses the last similar data value within a DTL context. In their evaluations, they note that their heuristic has a higher average accuracy and lower deviation than random value replacement. They evaluate the DASM based framework by running a data mining benchmark on two system simulations: one with a uniform cache architecture and one with a Network-On-Chip (NoC) based architecture. For the uniform architecture with a 3% skip rate, they observed a 45-50% reduction in memory access time and a 15% improvement in average execution time for only 5% loss in accuracy. Data for the NoC based architecture with a 3% skip rate was extremely similar.

2.2.3 Approximate Caches

Another approximate storage approach exploits data similarity across blocks in caches in order to improve effective cache capacity and performance. Miguel et al. [42, 44] introduce two approaches to approximate caches: Doppelganger, which de-duplicates similar blocks; and the Bunker Cache, which exploits spatio-value similarity to map similar blocks to the same cache location.

Miguel et al. [42], introduce Doppelganger, a last-level cache (LLC) design that de-duplicates similar blocks by using a single data entry to store them. The authors hypothesize that by approximately storing similar values in the same block entry, LLC area and energy consumption can be significantly reduced. Cache blocks are considered approximately similar if the differences between all elements across blocks is within some specified threshold, which is determined at design time in order to meet application output quality needs. When blocks are similar, their cache tags are associated with a single block entry in the Doppelganger cache that approximately represents all similar blocks. A simplified diagram illustrating this concept is shown in Figure 2.7 below. In practice, a normal LLC would be split into part normal cache for precise data and part Doppelganger for approximate data. Upon insertions, Doppelganger compares blocks values with an approximate similarity map consisting of a hash function and mapping step.

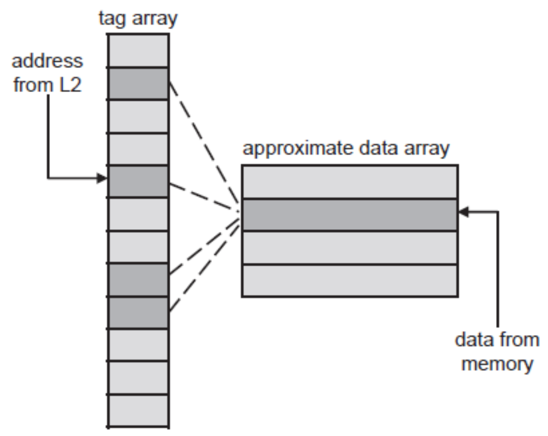


Figure 2.7: Doppelganger cache concept. Doppelganger associates tags of similar blocks with same entry in an approximate data array. This significantly reduces the size of the data array necessary to store approximately similar data. Attribution: (Miguel et al. 2015) [42].

They implement Doppelganger in a cycle-accurate x86 architecture simulator, use EnerJ [16] to annotate benchmark program data that is safe to approximately store, and assume the ISA support presented in [36] to identify approximate data to the hardware. Doppelganger's approximation of data results in 10% or lower quality reduction. Relative to a baseline 2MB LLC, they achieve LLC dynamic and leakage energy reductions of 2.55x and 1.41x, which correspond to 1.19x and 1.28x dynamic and leakage reductions for the total on-chip cache hierarchy. For area, they achieve 1.55x and 1.36x reductions for LLC and total on-chip cache hierarchy respectively. With regards to performance overhead due to an increase in LLC misses, Doppelganger incurs a 3.4% increase in off-chip memory traffic; they do not an incurred energy overhead for this increased off-chip traffic. In addition to these results, they compare Doppelganger with a well-known, on-chip cache compression algorithm, Base-Delta-Immediate (BΔI) [43], to evaluate its effectiveness in saving storage space; they find for the same set of benchmarks that Doppelganger saves 37.9% of storage on average compared to 20.9% for BΔI. Doppelganger's implementation cost is high due to the architectural changes it makes to LLC.

In another value similarity approach to caches, Miguel et al. [44] present the Bunker Cache, which exploits the *spatio-value similarity* of data stored in memory, in order to save energy and improve memory access performance. They define spatio-value similarity as "similarity in data values at regular intervals" and state this is a common occurrence when storing multi-dimensional structures in one-dimensional memory. The authors first explore spatio-value similarity by comparing values from approximate computing applications from the PERFECT and AxBench suites [45, 46] across memory address based off arbitrary sweeping strides. As shown in Figure 2.8, they observe periodic points where application quality peaks or degrades depending on stride length, leading them to believe spatio-value locality can be effectively exploited. The Bunker Cache works by applying a mapping function to a conventional cache, which defines the similarity stride, which must be defined by the programmer, and a radix, which acts as the approximation control knob for similarity mapping. This function maps blocks to *bunkaddresses* based off their physical addresses. Since Bunker Cache only needs address data to map data, this approach has low overhead. Additionally, this means they can improve cache hit rates by proactively approximating data since they can read from the *bunkaddress*. The Bunker Cache dedicates a bit in the *bunkaddress* to differentiating between approximate and precise data. They evaluate their work using a full-system cycle simulator with Bunker Cache as the LLC and using benchmarks from the aforementioned sources [45, 46] with support from an approximate code annotator [16, 17, 47] and ISA extensions [36]. For a y-dimension similarity radix of 2 (conservative), 8 (moderate), and 64 (extreme), they observe an average root-mean-square error of 3.7%, 7.7%, and 13.4%. For radix 2 and 64, they observe average energy savings of 1.18x and 1.39x and average performance improvement of 1.08x and 1.19x respectively.

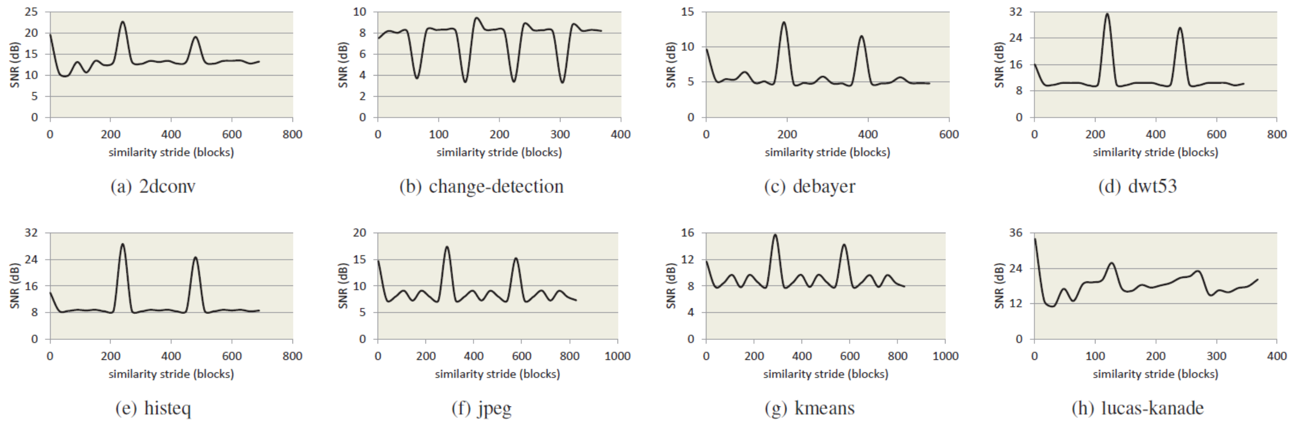


Figure 2.8: Application output quality due to sweeping similarity strides. Peaks indicate a high degree of similarity between data separated by the similarity stride length. Troughs indicate a low degree of similarity. Attribution: (Miguel et al. 2016) [44].

2.3 Algorithm Level Storage Optimization

In this section we review algorithmic techniques for approximate storage including precision scaling and look-up tables (LUTs). “Algorithmic” is broad term that can label techniques in circuits and architectures too, but we will use it to refer to processes and methods for approximation that are independent of specific hardware or system constructs. Whereas circuit level techniques rely on noise and defects in hardware to approximate data, algorithmic techniques deterministically approximate data.

2.3.1 Precision Scaling of Data

Precision scaling, the process of disregarding some number of least-significant bits (LSBs) in a data word, is a common approximate technique that is easily applied to computation, communication, and storage [39, 41, 80, 81]. The motivation for precision scaling is that LSBs have the smallest impact on quality, but cost the same as more significant bits. Precision scaling can save energy by reducing the amount of data that needs to be transported in a system, the amount of memory needed to store a word, and the amount of switching in an arithmetic unit. In this section we will discuss approximate storage techniques based off precision scaling.

In consideration of RMS applications, Tian et al. [37] present ApproxMA, a dynamic precision scaling framework for off-chip memory accesses that reorganizes data words into words containing subsets based on the bit-significance of each word. This reorganization of words is illustrated in Figure 2.9. For example, a stored 32-bit word would contain the 4 most-significant bits (MSBs) of 8 normal 32-bit words. The next 7 stored words would contain following 4-bit subsets of the remaining 28 bits per word, with the last stored word containing the 4 LSBs of each normal word. Using a memory access controller and runtime precision controller, both implemented in software, they are able to control the precision that they retrieve data from memory with. The runtime precision controller monitors error bounds on modeling data during computation and subsequently updates a precision constraint table, which is used to determine the precision scaling allowable for a memory access. They evaluate ApproxMA on various datasets with two clustering algorithms: Gaussian mixture model-based (GMM) and k-means. They define model deviation and testing data error rate as their application quality metric, which represent the distance between an approximate and accurate model and the percentage of misassigned clustering data respectively. They observe an average memory access energy savings of 51% for GMM and 52% for k-means, with near-zero quality loss under both metrics. They note that ApproxMA has some energy consumption overhead due to its data reorganizing process, but state that is negligible and do not present any data for it. We believe this is a reasonable assumption given the orders of magnitude greater energy consumption of memory access operations compared to compute operations [6]. Since this framework can be implemented in only software, it has broad applicability, but can likely be made more efficient if implemented within memory controller hardware.

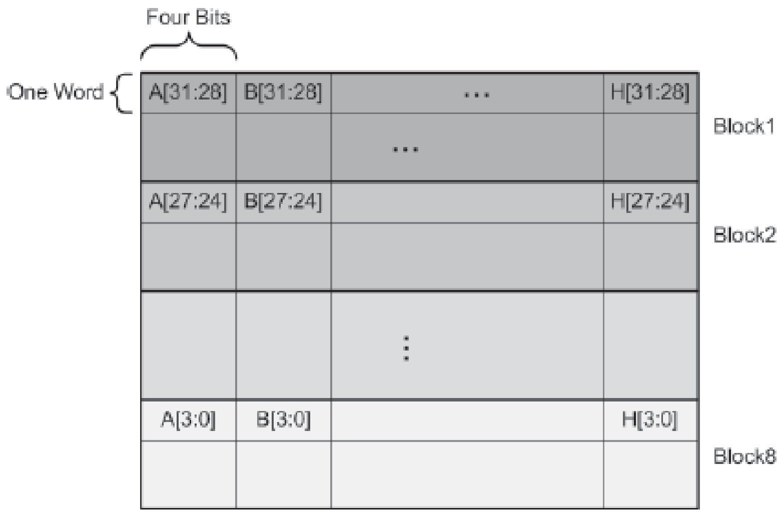


Figure 2.9: ApproxMA off-chip memory data format. A, B, …, H represent the original data words, which are now separated into 4-bit segments and distributed across 8 memory blocks. The first block stores the 4 MSBs of each word and each subsequent block stores the next most significant 4-bits for each set of words. Attribution: (Tian et al. 2015) [37].

Ranjan et al. [18] propose approximately compressing non-critical data before writing it to off-chip memory (and subsequently decompressing it when reading it back) in order to reduce memory traffic and use a runtime quality controller to ensure application quality targets are met. By reducing the overall quantity of data being transported, they reduce DRAM write/read energy costs. They approximately compress data through *bidirectional* precision scaling, where they truncate some number of MSBs and LSBs, as shown in Figure 2.10. These truncations typically introduce insignificant errors because the LSBs are of little significance and the MSBs, which are zero or sign-extended, are recovered with a padding bit. Their work allows the programmer to identify non-critical data for approximation by specifying memory regions for it, set a quality target for the runtime controller, and define a quality evaluation function. During runtime, their quality control framework compares quality output with the target quality and makes adjustments to approximation as necessary. They evaluated their method on FPGA with recognition and detection benchmark applications from [1]. They observed an average performance improvement of 11.5% and average energy improvements of 1.28x, all while maintaining less than 1.5% application quality loss. They note that their work has broader applicability over works that focus on approximating at the memory technology/circuit level because their work can be applied to any off-chip memory subsystem since their modification is in the memory controller.

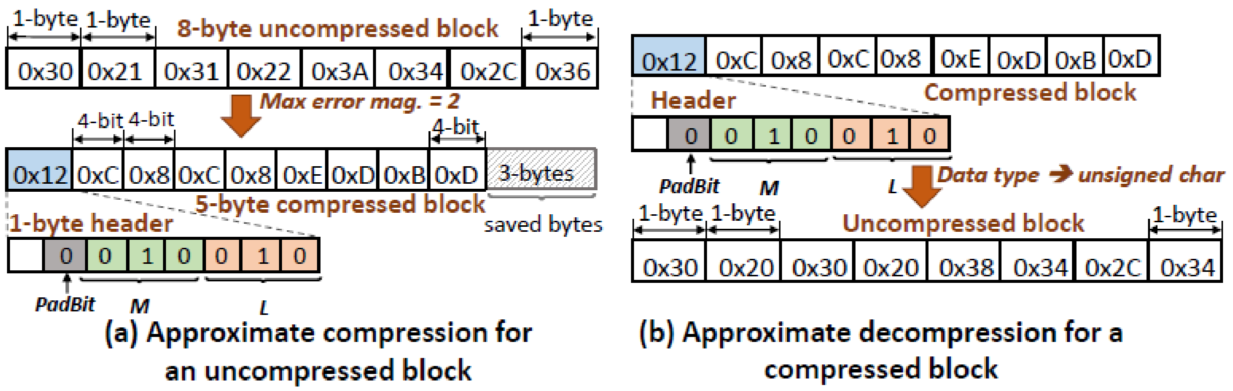

1-byte
1-byte
8-byte uncompressed block
1-byte
0x30 0x21 0x31 0x22 0x3A 0x34 0x2C 0x36
Max error mag. = 2
4-bit 4-bit
4-bit
0x12 0xC 0x8 0xC 0x8 0xE 0xD 0xB 0xD
3-bytes
5-byte compressed block
saved bytes
1-byte header
0 0 1 0 0 1 0
PadBit
M
L
(a) Approximate compression for an uncompressed block
0x12 0xC 0x8 0xC 0x8 0xE 0xD 0xB 0xD
Header
Compressed block
0 0 1 0 0 1 0
PadBit
M
L
Data type → unsigned char
1-byte 1-byte
Uncompressed block
1-byte
0x30 0x20 0x30 0x20 0x38 0x34 0x2C 0x34
(b) Approximate decompression for a compressed block

Figure 2.10: Bidirectional precision scaling of storage data. M and L indicate the number of MSBs and LSBs to be truncated from each word. Upon decompression, the truncated MSBs are replaced with the stored PadBit and the LSBs are replaced with zeros. Attribution: (Ranjan et al. 2017) [18].

Observing that many general purpose graphics processing unit (GPGPU) workloads are bounded by memory bandwidth, Sathish et al. [64] propose lossless and lossy compression techniques for transferring data between GPU and off-chip memory to improve the efficiency of available memory channels. They use C-Pack, developed by Chen et al. [62], for lossless compression of blocks transferred to DRAM. For lossy compression, they apply precision scaling to floating-point (FP) data by truncating some number of LSBs, then compress it with C-Pack. Their lossy compression scheme is shown in Figure 2.11. They note that reducing the precision of FP data has very little impact on the accuracy of results because the LSBs of the mantissa contribute very little to the overall value represented by a FP number. They point out that truncating the 8 LSBs of a 32-bit FP number for all words in a block would effectively reduce bandwidth usage by 25%. Through modification of a GPU memory function, the programmer can specify the degree of truncation (none, 8-bit, 12-bit, or 16-bit) to apply to FP data. This is the quality-energy control knob for this approximate storage scheme. They evaluate their lossless and lossy link compression schemes in a GPU architecture simulator with a set of memory-bound and compute-bound benchmark workloads. They report for memory-bound workloads that lossy compression with 8, 12, or 16 bits of truncation improves workload performance by 15%-41% compared to lossless compression and error is limited to a normalized root-mean-square error of 10^{-5} to 10^{-3}.

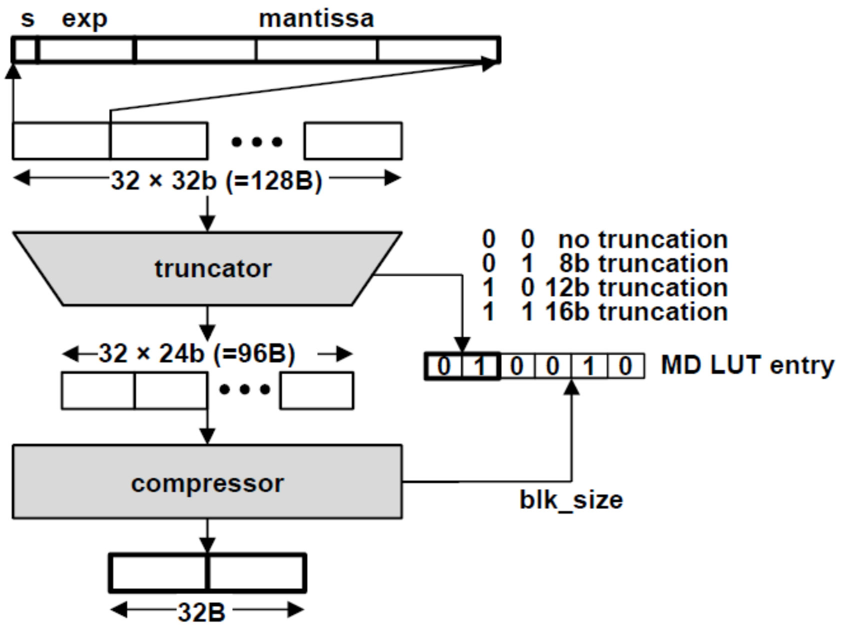


Figure 2.11: Lossy compression of floating-point (FP) data in hardware achieved by combining truncation with lossless compression. LSBs of FP data words are truncated before applying lossless compression to a whole block of data words. Truncating the LSBs increases the efficiency of the lossless compression. Attribution: (Sathish et al. 2012) [64].

2.3.2 Memoization and Look-Up Tables

One general technique to reduce the cost of complex computations (i.e. trigonometric functions) is to store the outputs to a corresponding set of inputs in a memo or look-up table (LUT) in hardware for future use [48]. This can be done by either predefining LUT values or by storing computed values during runtime. This saves future instructions' energy and possibly time since they can use a pre-stored value or reuse a previously calculated value instead of performing long and/or complex arithmetic. However, the set of input combinations, and therefore computations that can be saved, is directly limited by the size of the LUT. Since it would be infeasible to store all the outputs for all possible function inputs, LUTs typically store the outputs for frequent input patterns. One class of approximate storage algorithms seeks to approximate LUTs in order to increase their effectiveness by improving the probability of output reuse.

Early on, Alveraz et al. [40] proposed a fuzzy memoization technique using a LUT to exploit the inherent error resilience of multimedia applications for mobile systems. They use the term "fuzzy" in this work, but in current literature the term is superseded by "approximate." Their goal is to improve function reuse for floating-point (FP) operations, which are difficult to achieve a high degree of reuse with due to the extremely precise nature of FP values. They take a conventional double precision (64-bit) FP ALU and LUT functional unit and compare it with one that has a fuzzy LUT. This microarchitecture is shown in Figure 2.12. Typically, the input FP operands' less significant mantissa bits are XOR'd together to form a read address that is input to the LUT, and if it successfully finds a reuse value, then that is use instead of performing FP multiplication or division. With their fuzzy LUT, they approximate input operands before LUT read by removing some number of LSBs from the FP mantissa. They define N as the tolerance level, which they use in quality evaluation of their technique. This precision reduction, a form of precision scaling, allows for inputs similar to some previously input set to reuse that previous set's stored computation result. Whereas in a precise LUT, a set of inputs may only use an LUT entry with a precise match, they may now use any similar value where the difference in values lies within those lower N mantissa bits. To summarize, their fuzzy LUT will increase computation reuse rates at the cost of quality. The authors evaluate their fuzzy LUT against a precise LUT on an embedded processor system simulator with a benchmark suite of multimedia applications including audio, image, 3D rendering, and speech recognition and use signal-to-noise ratio (SNR) as their quality metric, noting that any SNR above 30dB has nearly imperceptible error to humans. They find for a tolerance level, N, of up to 30, there is no change in SNR, but over the range 31 to 46, reuse rates steadily increase while SNR drops. For the FP unit, they observe average energy savings of 35%

and 9% for the fuzzy and precise LUT respectively over a baseline FP unit without a LUT. This work has low implementation overhead because it only requires small modification to existing FP-LUT functional units.

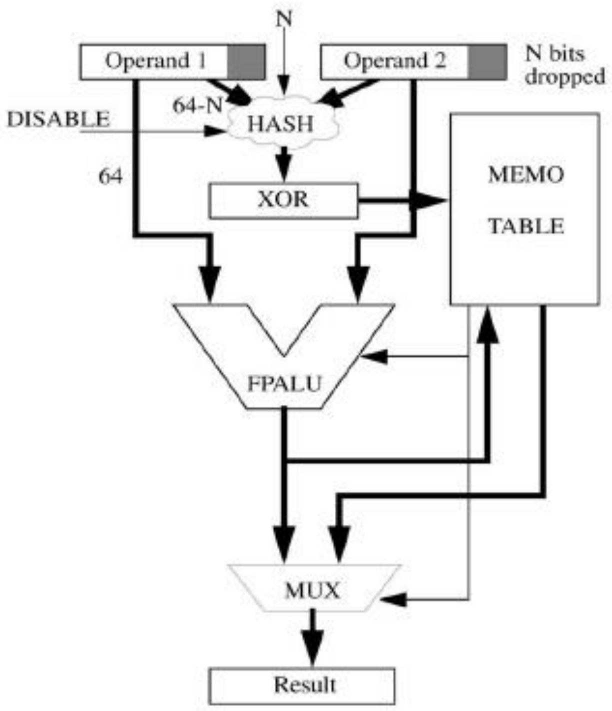


Figure 2.12: Fuzzy memoization architecture. Results of floating-point arithmetic logic unit (FPALU) are stored in a memo, or look-up, table (LUT) for reuse. Later, the two input operands are used to generate an address to search the LUT with. In this architecture, some number, N, of least significant mantissa bits are truncated from each operand before searching the LUT. In a precise LUT, only a precise match for a set of inputs may be returned as the function result, but now similar values can be used when the difference in values lies within those lower N mantissa bits. When a result is found in the LUT, the FPALU is bypassed and the result is directly output. Attribution: (Alveraz et al. 2005) [40].

Tian et al. [49] present ApproxLUT, an approximate LUT that enables variable degrees of approximation by computing the output based on an error bound in addition to the input operand. They accomplish this by designing a LUT with two levels and with lightweight error compensation. The second level holds a set of sub-tables which each stores finer granularity of results than the first table. The error compensation is an optional portion of arithmetic that can be used to reduce the error induced by approximation. They designed ApproxLUT to approximate functions for error-resilient kernels with compute-intensive functions, including trigonometric, exponential, and logarithmic functions, because they are the most amenable to and worthy of approximation, which is consistent with the observations in [1]. For each of these functions they specify a number of points across the continuous output value range and then select a number of those based on their error-bounds to store in the LUT. ApproxLUT has 4 approximation modes, which are defined by which level table is accessed and if compensation is used. 2nd level lookup with compensation is the most accurate approximation and 1st level without compensation is the least accurate. The approximation mode for a given function lookup is determined by the input error bound, which is compared against 4 threshold error bounds corresponding to the 4 approximation modes. ApproxLUT is evaluated on a memristor RAM model (although it can be applied to any memory technology) with benchmarks simulated on CPU system simulator. Energy savings and error expectation of ApproxLUT compared to normal computation for various functions are presented in Table 2.2. The error expectation is the expected deviation from original value for all points stored in the LUT. Its shown that trigonometric functions can achieve energy savings of at least 50% with very low error expectation using approximation mode 1. Exponential and logarithmic functions actually incur energy savings at approximation mode 1, but can achieve high energy savings by removing the error compensation.

Table 2.2: Error expectations, which are the expected deviation from original value for all points stored in LUT, and energy savings (E_s) for trigonometric, exponential, and logarithmic functions and their value domains using ApproxLUT, relative to normal computation. Attribution: (Tian et al. 2017) [49].

Approximation Mode		cos x	tan x	e^x	ln x
		[0,π/2], [0,1]	[0,2π/5], [0,3.08]	[0,3], [0,20.09]	[1,10], [0,1.61]
1 (2nd level, w/comp.)	error E_s	1.03e-05 49.85%	1.65e-05 59.55%	0.0008 -2.76%	7.13e-06 -9.98%
2 (1st level, w/comp.)	error E_s	0.0015 55.41%	0.0033 64.03%	0.1534 8.64%	0.0016 2.21%
3 (2nd level, w/o comp.)	error E_s	0.0043 88.87%	0.0045 91.03%	0.0556 77.20%	0.0022 75.60%
4 (1st level, w/o comp.)	error E_s	0.0414 94.44%	0.0559 95.51%	0.7912 88.60%	0.0333 87.80%

3 Proposed Work: Improving Compression with Precision Scaling

As compute capabilities continue to scale, memory capacity and bandwidth continue to lag behind. Data compression is an effective approach to reducing both memory and interconnect bandwidth requirements; but prior works have focused primarily on lossless compression and retrieval for data-critical applications. While data integrity is critical to many applications, a growing number of memory-bound RMS applications contain non-critical program data; this property can be exploited to improve memory performance and capacity. In this chapter, we enable approximate storage to exploit the inherent error resilience of multimedia and other applications, through lossy compression, to improve storage without adversely affecting application functionality. We realize an incidence of approximate storage through modification of the Base-Delta-Immediate algorithm [43] to create a lossy memory compression algorithm that can be applied to cache, link, and main memory architectures. We conclude that approximate storage is an effective and promising technique for improving effective memory capacity and decreasing the energy cost of memory access; results show that memory capacity can be increased by 15% and memory access energy costs can be reduced by 15% with only 3% loss in application quality.

3.1 Introduction

As discussed in Chapter 1, the performance bottleneck of data-intensive media and RMS applications are lies in the memory hierarchy, where memory accesses come with high latency and energy costs. Data-intensive application performance will suffer if novel solutions aren't presented to overcome off-chip bandwidth limitations. In Chapter 2, we observed many approximate storage techniques that address challenges in the memory hierarchy in order to improve application performance and decrease storage costs.

Another general solution to these memory performance issues is lossless data compression. This research direction focuses on increasing effective memory capacity to improve application performance. There already exists a substantial body of knowledge on compression at the cache, link, and main memory levels. Works on cache compression seek to improve effective cache capacity to reduce the cache miss rate and subsequently reduce the number of main memory accesses [9, 43, 61-63]. Cache compression/decompression requires extremely low latency operation to prevent caches themselves from becoming a performance bottleneck. Link compression focuses on compressing data for interconnect transport before storage in main memory [9, 18, 64]. Many published works on improving memory performance focus on increasing effective storage capacity and bandwidth through compression [63, 65, 66], but nearly all of them are limited by their lossless nature.

Lossy compression achieves much higher compression ratios than lossless compression by trading off data integrity (typically in media data). The inherent error resilience of some applications makes lossy compression feasible for program data. Inherent error resilience (IER) is the property of an application to produce acceptable outputs despite underlying computation errors or approximations [1]. Sources of IER include: noisy or redundant inputs, lack of perfect answer or outputs for human perception, and statistical or iterative computational patterns. Most prior work in memory compression does not consider a target application domain when compressing data, and therefore only consider lossless compression, in order to restore fidelity perfectly for all program data. In consideration of error-resilient applications, such as image search, hand written digit classification, and eye detection, approximating data through lossy compression can increase effective bandwidth and memory capacity more than lossless memory compression could. Works by Ranjan et al. [18] and Sathish et al. [64] are some of the first to explore lossy memory compression for data-intensive applications.

In this chapter, we propose a general purpose approximate storage technique to improve effective storage capacity and the energy consumption of memory accesses. We realize approximate storage through a lossy modification of the Base-Delta-Immediate (BΔI) lossless compression scheme [43]. Our work differs from lossless compression schemes works because it exploits the inherent error resilience (IER) of media and RMS applications. Our work also differs from the discussed circuit level approximate storage techniques because ours deterministically approximates data while those allow errors to be introduced passively. Our work trades off data quality for improvements in memory subsystem capacity and energy consumption. We make the following contributions:

- We extend BΔI with *dynamic precision scaling* before compression arithmetic. The low dynamic range of words across a block suggests that approximation of deltas will improve compression ratios without significantly affecting data integrity.
- We evaluate the impact of our approximate storage technique on effective storage capacity by measuring compression ratio improvement and on media quality by measuring peak signal-to-noise ratio (PSNR), structural similarity index (SSIM), and image classification quality metrics.
- We estimate the improvement in memory access energy consumption as a result of our lossy compression and evaluate the hardware overhead of implementing our approximate storage technique.

Section 3.2 summarizes the lossless BΔI compression algorithm that we extend. Section 3.3 presents our approximate storage algorithm. In Section 3.4, we discuss our evaluation methodology for our algorithm. We present our results in Section 3.5.

3.2 BΔI Lossless Compression

When exploring lossless memory compression techniques to exploit for approximate storage, we sought a technique which primarily has extremely low latency. Other traits such as complexity and area were secondary. Low latency is important because memory access operations already have long latency relative to compute operations. The latency penalty incurred by a memory compression/decompression unit should not critically degrade memory access latency. Pekhimenko et. al [43] proposed Base-Delta-Immediate (BΔI), a value-based lossless cache compression scheme, which is extremely low latency. Compression and decompression take about 1-2 and 1 cycles respectively. BΔI is implemented as a simple two-stage scheme (arithmetic and encoding selection) and does not need a dictionary, which would incur storage and maintenance penalties, as shown in [62]. The major drawback to BΔI is its high area overhead due to the use of adders in parallel.

BΔI encodes a block, or cache line, of words as a base value followed by an array of delta values. A block is typically the size of a cache line, 32 or 64 bytes. In this paper, we consider the 32-byte block size. Compression is achieved by storing the block as a base and an array of deltas with an encoding symbol, as opposed to an uncompressed array of raw values. A given uncompressed block and its corresponding compressed block are respectively defined by

$$uncompressed\ block = \{V_0, V_1, \dots, V_{n-1}\}$$
$$compressed\ block = \{base, \Delta_0, \Delta_1, \dots, \Delta_{n-1}\}$$
$$\text{where } \Delta_i = V_i - base$$
$$\text{for } i = \{0, 1, \dots, n-1\}$$

The words can be any size (i.e. 16, 32, and 64 bit) and represent any common data type (i.e. int, float, and double). Representing a block of words as a base word value and an array of consecutive deltas (relative to the base) corresponding to consecutive words in the block may require fewer bytes to store than the original block of words does. This is dependent on the existence of *low dynamic range* among neighboring words, meaning they hold similar values. For example, low dynamic range can be found in neighboring pixels in the low frequency regions of an image, where there is little to no difference in value. If consecutive words in a block have similar values, then the differences between those values will be small relative to the word size. This concept is illustrated in Figure 3.1 [43] where the words in the block exhibit low dynamic range. This is shown by the fact that the difference between each 4-byte word and the base value can be represented with only 1 byte.

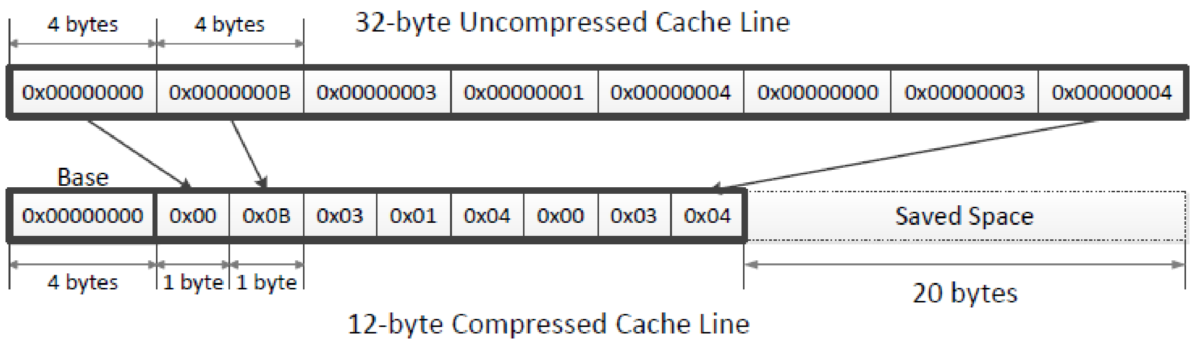


Figure 3.1: Example of BΔI compression. Using the first word of the cache line as a base, subtraction is performed between every word and the base. The differences are stored after the base as an array of deltas. Each word in the uncompressed cache line is 4 bytes wide, but each delta in the compressed cache line is only 1 byte wide. Attribution: (Pekhimenko et al. 2012) [43].

BΔI compression takes place over two major steps: BΔI arithmetic and encoding selection. In BΔI arithmetic, the uncompressed cache line feeds into several compressor units (CU) in parallel. Each CU operates on a specific base-delta (B, Δ) byte width combination from the set of pairs

$$(B, \Delta) = \{(8,1), (8,2), (8,4), (4,1), (4,2), (2,1)\}$$

The BΔI architecture is shown in Figure 3.2 and a Base8-Δ1 CU is shown in Figure 3.3 [43]. Within each CU, a word in the cache-line is selected as a base of size B, and the differences between every word and the base are calculated. Differences are checked for a sign-extension beginning from its lower Δ bytes to determine if the they are small enough to be stored in the specified delta size. A given CU's output encoding is only valid if all of its differences are sign-extended from their lower Δ bytes. To increase compressibility, they add a second base, which is always set to an implicit immediate zero (hence the "immediate" in Base-Delta-Immediate). This is to take advantage of datasets where values may be close to zero and increase the coverage of every (B, Δ) pair. This implicit base requires less storage than an explicit base (delta minus zero equals delta). If compression isn't possible with the zero base, then the first base whose value lies outside the given delta range is selected as the base specified at the beginning of this paragraph. In addition to these base-delta CUs, there are units which check and compress for zero or repeated values, a common value-based compression technique [61, 65]. These zero and repeated value checkers can compress with much greater efficiency than BΔI arithmetic for a block of zeros or repeating values, but are limited in their scope to those types of values. In total, the first stage of BΔI consists of 6 CUs and the zero and repeating value checkers in parallel.

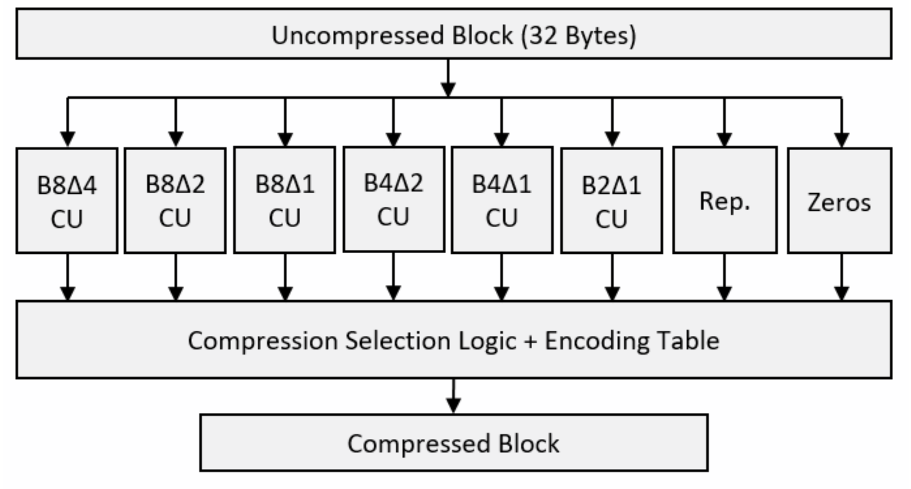

Uncompressed Block (32 Bytes)
B8Δ4 CU
B8Δ2 CU
B8Δ1 CU
B4Δ2 CU
B4Δ1 CU
B2Δ1 CU
Rep.
Zeros
Compression Selection Logic + Encoding Table
Compressed Block

Figure 3.2: BΔI compression architecture. In the first stage, compressor units (CUs) perform BΔI arithmetic in addition to repeated and zero value checks. In the second stage, the most efficient valid CU output encoding is selected as the compressed block based on the BΔI encoding table.

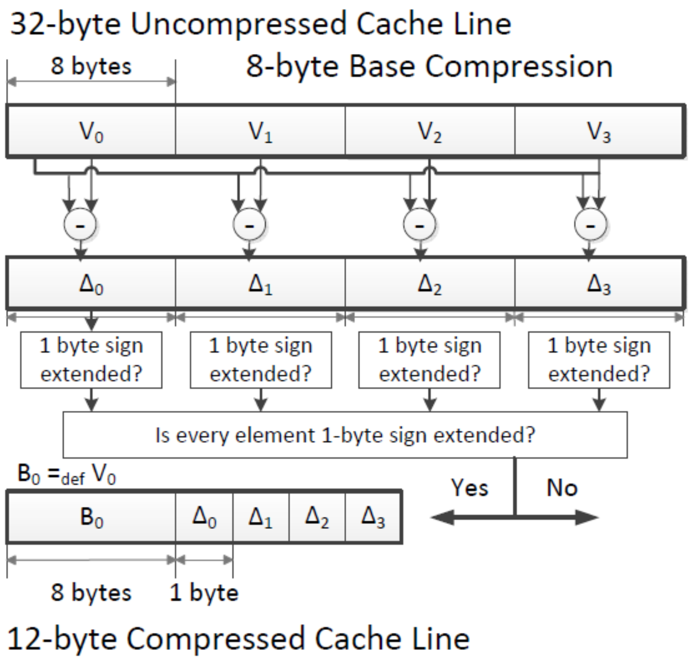


Figure 3.3: Base8-Δ1 compressor unit. First, subtraction is performed between each 8-byte word of the uncompressed cache line and the selected base, V_0. Differences are then checked for sign extension from the least significant byte. If true for all differences, then the differences are stored as 1-byte deltas in the compressed cache line and the valid encoding signal is sent to compression selection logic as shown in Figure 3.2. Attribution: (Pekhimenko et al. 2012) [43].

The second major step in BΔI compression is selecting the best compression encoding. Once all CUs have computed BΔI arithmetic, encoding selection is made based on a "compressible or not" signal output from every CU and a static encoding table, shown in Table 3.1. It is possible for multiple encodings to be valid. Of the valid CU encodings, the most space efficient one is selected from the encoding table and is output as the compressed block. The selected compressed block is output with the appropriate encoding symbol from the table as meta data. In BΔI, meta data bits are added to the cache architecture and are not included in compressed data size. Note that our work includes the meta data in compressed block size. In Section 3.5 we present comparisons between compression with and without meta data overhead.

Table 3.1: BΔI encoding table for 32-byte blocks. Except for zeros and repeating, Base8-Δ1 is the most space efficient encoding. Except for the case when compression isn't possible, Base2-Δ1 is the least efficient space encoding. Sizes are in bytes. Attribution: (Pekhimenko et al. 2012) [43].

Encoding Name	Base Size	Δ Size	Total Size
Zeros	1	0	1
Repeating	8	0	8
Base8-Δ1	8	1	12
Base8-Δ2	8	2	16
Base8-Δ4	8	4	24
Base4-Δ1	4	1	12
Base4-Δ2	4	2	20
Base2-Δ1	2	1	18
No Compression	N/A	N/A	32

BΔI decompression simply consists of checking the BΔI encoding symbol from the compressed block and performing addition with the appropriate (B, Δ) widths to build the uncompressed block. The latency of decompression is about the latency of 1 adder and is typically a single cycle, which is ideal for latency critical memory read operations.

3.3 Approximating Data

We enable approximate storage through a lossy extension of the BΔI compression algorithm. We make BΔI lossy by applying *dynamic precision scaling* to the subtraction operands of each CU.

This improves data compressibility by increasing the probability of a more efficient compression encoding being valid for a given block at the cost of data integrity. This dynamic precision scaling controls the quality-energy tradeoff in this approximate storage realization. This approximation does not decrease the size of each (B, Δ) pair's encoding size, but rather the probability that a smaller sized encoding will be valid. Our approximation reduces the magnitude of the deltas, thus increasing compressibility. We approximate the operands by right-shifting them so that their least significant bits (LSB) are shifted out and shift the value of the sign bit into the MSBs. This causes their sign bits extend further, which causes the calculated differences' sign bits to further extend too. A given base-delta's compression encoding is only valid if all of its differences are sign-extended from its lower Δ bytes, which means extending the differences' sign bits increases the probability of that compression encoding being valid. If a more efficient compression encoding is valid, then more space can be saved and the overall compression ratio improves. BΔI compression can be significantly improved through precision scaling at the cost of integrity loss in the LSBs.

Figure 3.4 shows our approximate CU. Following the precision scaling approach, we shift out the lower α bits of the subtraction operands by inserting right-shifters before the subtraction units' inputs. The shift amount is controlled by the *approximation degree*, α, which reduces the precision of the deltas by α bits each. Rather than producing a difference accurate over the bit range [width-1 : 0], each subtraction unit produces a difference accurate only over the range [width-1 : α]. This quantizes the value of each delta by a factor of $1/2^{\alpha}$. The lower α bits are implicitly set to the value of the sign bit to skew positive and negative values towards zero, so data are not biased towards positive or negative infinity. These implicit values are later shifted back in during decompression. Since α is necessary for decompression, it is stored in the compressed block's meta data. The meta data overhead is shown in the approximate BΔI encoding table in Table 3.2. The size of α stored in meta data varies from 3-5 bits depending on the base-delta compression encoding. This enables a max scaling of 7, 15, and 31 bits for bases 2, 4, and 8 respectively. α can be set to zero to enable lossless compression for critical data.

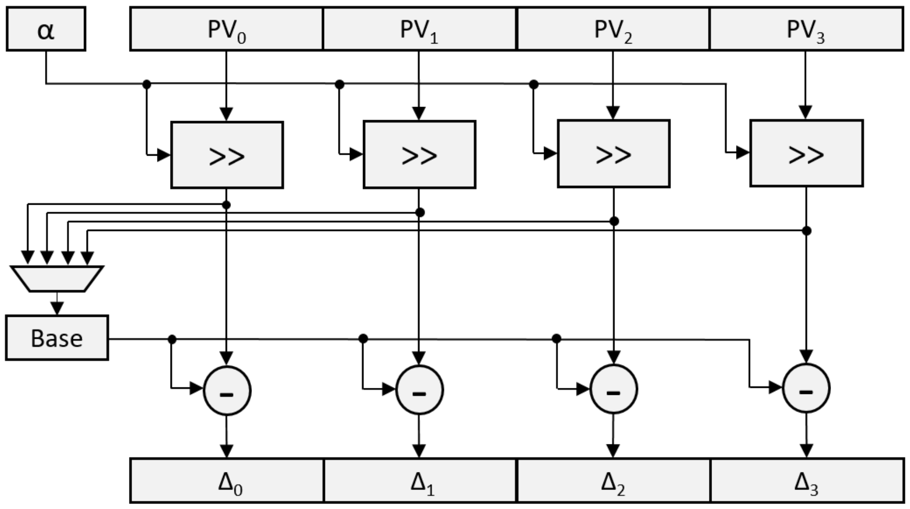


Figure 3.4: Generic compressor unit (CU) architecture with added right-shifters before subtraction. Specific implementations exist for all BΔ combinations from Table 3.2. The uncompressed block of precise values (PV) is shown at the top. The array of deltas, the base, and the approximation degree, α, all go into the compressed block. α controls the right shift amount for all subtraction unit input operands and the base. Through this precision scaling of operands, α controls the quality-energy tradeoff for our approximate storage technique.

One may question why not apply precision scaling to the differences instead of the CU input operands if the final output is what matters. We apply precision scaling to the inputs for 2 reasons. First, shifting the LSBs out of the outputs does not save any shifter units because there is one output for every input value. Applying shifting to the input values uses the same number of shifters as the output values. Second, shifting the LSBs out of the inputs to the subtraction units reduces the amount of switching that occurs within the subtraction blocks because input values are skewed closer to zero.

This shifting is advantageous because each difference's sign is now extended by an extra α bits. Take for example a 1-byte delta, which holds a signed value range of -128 to 127, as shown in Figure 3.5. In lossless BΔI, a 1-byte delta wouldn't be wide enough to hold a subtraction result of 129, which is also shown in Figure 3.5. This means a base-delta pair with a 2-byte wide delta (less efficient encoding) is needed to hold it. If the subtraction operands were approximated with α=1, then each difference produced would be right-shifted by one bit. The difference value of 129 is now represented as 64 (not 65 because the LSB was dropped). A 1-byte delta is now wide enough to hold the difference value, but at the cost of the LSB of precision. A base-delta encoding with a

1-byte wide delta can now be selected over 2-byte wide delta encoding. According to Table 3.2, this saves 4 extra bytes in the case of a 32-byte block with an 8-byte base. For any given block, the number of bytes saved by approximation is dependent on the word values in the block.

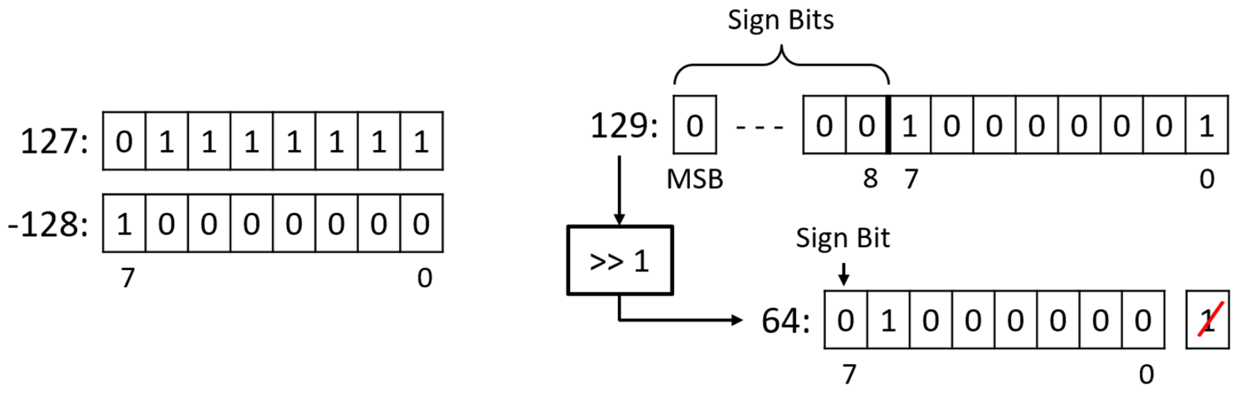


Figure 3.5: Maximum and minimum values for a signed 1-byte delta (left) and the impact of right shifting on delta value and sign-bit extension (right). The value 129 cannot be represented by a 1-byte signed word, but after right shifting by 1 bit, it can be stored in 1-byte as 64. Upon decompression, the bits would be left shifted and the approximate value 128 would be returned.

Table 3.2: Approximated BΔI encoding table for 32-byte blocks. Similar to Table 3.1 except for the addition of the Meta data column. In consideration of main memory architectures where modifications to include meta data bits would be impractical, our work accounts for compression meta data overhead. Sizes are in bytes.

Encoding Name	Base Size	Δ Size	Meta Size	Total Size
Zeros	1	0	0	1
Repeated Values	8	0	1	9
B8D1	8	1	2	14
B8D2	8	2	2	18
B8D4	8	4	2	26
B4D1	4	1	2	15
B4D2	4	2	2	23
B2D1	2	1	1	21
No Compression	N/A	N/A	1	33

Upon decompression, a compressed block’s deltas are shifted to the left by α bits and then added to the base value. The decompressor unit architecture is shown in Figure 3.6. For every left-shift, we replace the LSB with the sign-bit value. Decompressed values are thus skewed towards zero. Other than this left shifting policy, decompression is the same as in BΔI.

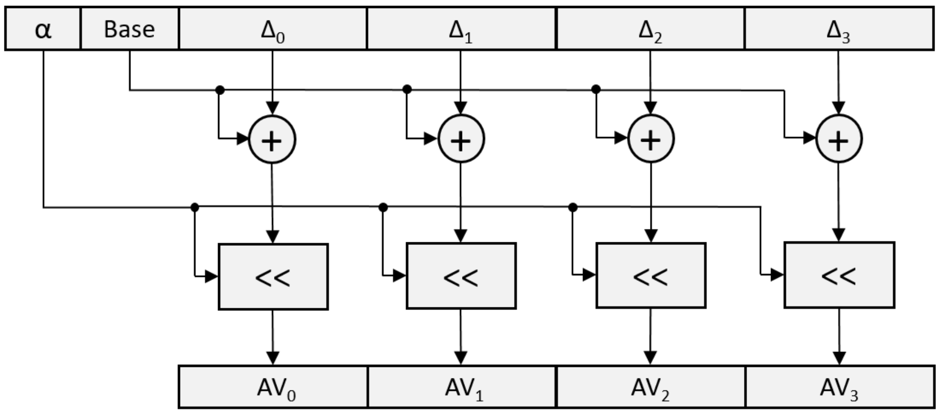


Figure 3.6: Generic decompressor unit architecture with added left-shifters after addition. Specific implementations exist for all BΔ combinations from Table 3.2. The compressed block is shown at the top and the decompressed block of approximate values (AV) is shown at the bottom. Approximate degree, α, controls the left shift amount for all addition unit outputs.

3.4 Evaluation Methodology

We evaluate how approximate storage affects data compressibility and quality by testing our method on a set of color images from the Caltech 101 dataset [67]. This image dataset has categories of different objects which are used for recognition algorithm training and testing (i.e. RMS applications). Sample images from different categories are shown in Figure 3.7. The images are in RGB24 format. We evaluate the compression and error metrics of our lossy compression technique in MATLAB, evaluate the impact of quality degradation on image classification with the Clarifai application programming interface (API) [68], and evaluate the power, area, and timing overheads of our design in Cadence [69].

Figure 3.7: Sample images from different categories of the Caltech 101 dataset [67]. Categories from left to right: airplanes, pizza, cougar_body, and electric_guitar.

3.4.1 Compression and Error Metrics

First, we verify the functionality of and explore the effectiveness of our approximate storage algorithm through MATLAB simulations. We compress and decompress 24 sets of 40 images with our approximate storage technique and with the original BΔI for comparison. For precision scaling, we use degrees of α = {0, 2, 4, 6, 8, 12, 16} to simulate approximation control at a moderate granularity. After these compressions/decompressions, we are left with 7 quality levels for every image. This gives us a total of 6,720 approximate storage simulations. For all 6,720 images, we gather data on compression ratios and two error estimators: peak signal-to-noise (PSNR) and the structural similarity index (SSIM) [59].

The compression ratio is defined as:

$$compression\,ratio = \frac{uncompressed\ size}{compressed\ size}$$

This ratio is the inverse of the compressed data's size as a fraction of the original data's size.

PSNR represents the maximum error between two images and is measured on a logarithmic decibel scale. The higher the PNSR value, the greater the image quality. The MATLAB PSNR function [71] defines PSNR as:

$$PSNR = 10 \times \log_{10}(\frac{R^2}{MSE})$$

where *R* is the maximum value for the image data type and MSE is the mean squared error of the image. MSE is defined as:

$$MSE = \frac{\sum_{M,N}[I_1(m,n) - I_2(m,n)]^2}{M \times N}$$

where *M* and *N* are the number of rows and columns in the images.

SSIM is used to evaluate the perceptual degradation of structural information in an image and compares luminance, contrast, and structure across two images [59]. A maximum value of 1.0 is output when the images are identical. The MATLAB SSIM function [70] defines SSIM as:

$$SSIM(x,y) = [l(x,y)]^{\alpha} \times [c(x,y)]^{\beta} \times [s(x,y)]^{\gamma}$$

where

$$l(x,y) = \frac{2\mu_x 2\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1},$$
$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2},$$
$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

where $\mu_x, \mu_y, \sigma_x, \sigma_y,$ and σ_{xy} are the local means, standard deviations, and cross-covariance for images x and y, and $\alpha, \beta,$ and γ are used for weighting each component. Note that this α for weighting is unrelated to the approximation degree, α.

3.4.2 Image Classification

Next, we evaluate the impact of our approximate storage technique on image quality using a recognition application service provided by Clarifai [68] through an API. Clarifai uses deep convolutional neural networks (CNNs) to recognize concepts in images and videos. CNNs are common algorithmic technique used in RMS applications [5]. By evaluating Clarifai's computer vision accuracy for every image in our dataset across 7 quality levels, we can draw conclusions about our technique's impact on RMS application quality. We use Clarifai's general recognition model, which analyzes images for common objects, such as those in the Caltech 101 dataset, and for each image, returns the top 20 results for the concepts it predicts are in the image and the probability that those predictions are correct. Each result contains a predicted concept and accurate prediction probability value. Prediction probability is the probability that the CNN has correctly predicted the presence of a concept in an image. The top result has the highest prediction probability and the last result has the lowest prediction probability. Note that the goal of this experiment is not to evaluate the correctness of the Clarifai model, but rather the impact of quality degradation on the Clarifai model's ability to predict the same concepts in a given image.

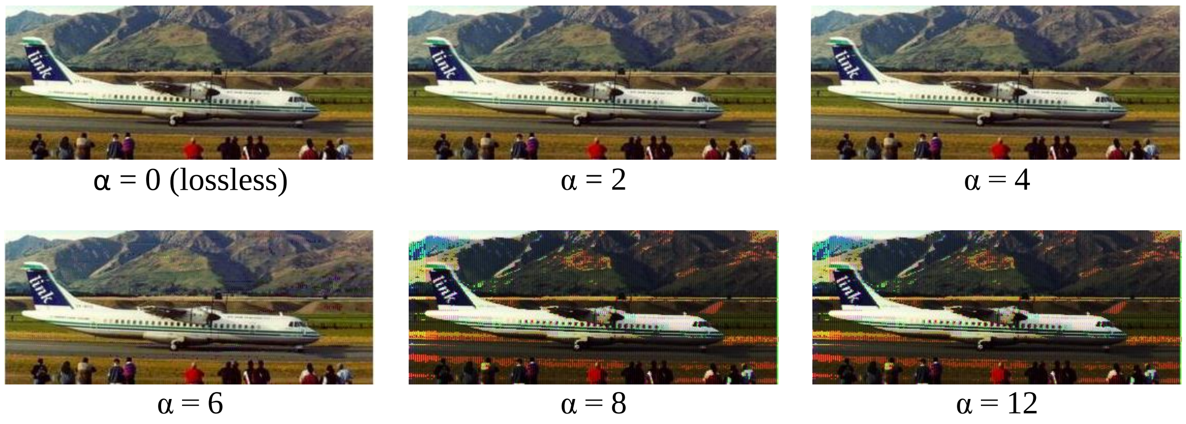
Using a Python script, we send the approximated images from our MATLAB simulations through a Clarifai API call, which later returns image recognition results. We store the top 4 predicted concepts (top 20% of returned results) for each of the highest quality images ($\alpha = 0$) as a baseline for evaluating the prediction probability on lower quality images. For each concept, we record its prediction probability in every lower quality image ($\alpha = \{2, 4, 6, 8, 12, 16\}$) processed by Clarifai. If one of the baseline concepts was not among the top 20 results for an image, then we assign it a prediction probability of 0% (a conservative approach, since it is likely still higher than 0%). After running all images through this process, we take the average of the top 4 prediction probabilities for every image for every quality level. We chose to use the top 4 predicted concepts since the Clarifai service may still incorrectly predict some concepts, even with high image quality level. Once we have the prediction probability for every quality level for every image, we calculate the loss in prediction probability for every image as a function of the quality-energy control knob, α.

3.4.3 Hardware

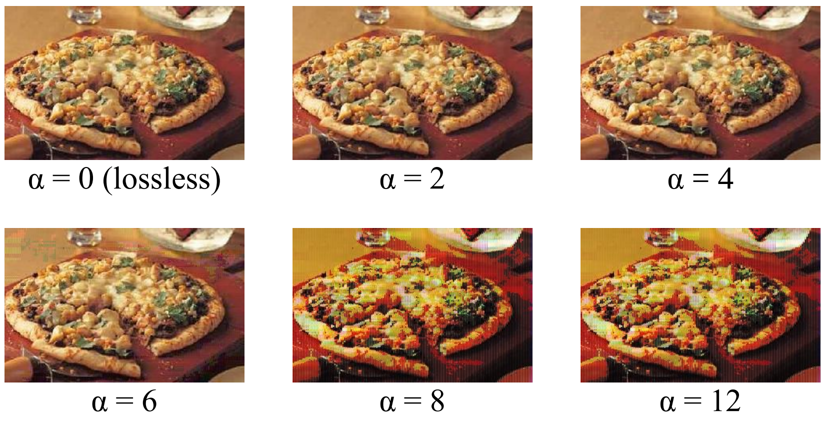
To measure the implementation cost of our approximate storage technique, we develop an RTL model in Verilog hardware description language (HDL). We synthesize the design into a gate level netlist using the Cadence Encounter RTL compiler [69] with a TSMC 65nm standard cell library to gather power, area, and timing simulation measurements. We then use memory access energy data from Pedram et al. [6] to estimate the memory access energy savings delivered by our technique.

3.5 Results

In this section we discuss the results of the aforementioned experiments. We present data on compression ratios, error metrics, image prediction, energy consumption, and hardware overhead. Our results show approximate storage of media data can significantly improve effective storage capacity and energy consumption for a small drop in media quality. In Figure 3.8, we present the decompressed approximate versions of two of the images from Figure 3.7 for α = {2, 4, 6, 8, 12, 16}. The quality loss in these images is similar to that for all approximated images in the dataset for the same value of α.



(a)



(b)

Figure 3.8: Decompressed images for varying degrees of approximation for images from airplanes and pizza categories.

3.5.1 Effective Storage Capacity

We first analyze the compression results from our MATLAB simulations on the Caltech 101 images. We show that our technique can increase effective storage capacity through its ability to significantly compress media data. This effective storage capacity increase is one of the benefits our technique receives for trading off data integrity for. Results show that our lossy extension of BΔI can achieve significantly higher compression ratios than lossless BΔI [43]. Our work assumes that internal memory fragmentation is avoided through the work of a compressed memory management scheme. We find this assumption reasonable because the same assumption is used in BΔI [43] and other lossless memory compression work [75].

In Figure 3.9, we present the average compression ratio with and without meta data overhead across all image categories for each of the 7 tested quality levels (induced by varying degrees of α. Compression with and without meta data accounting represent off-chip memory and cache compression respectively. Compression without accounting for meta data overhead represents implementation at the cache level where metadata can be stored as extra tag bits in the cache architecture. Pekhimenko et al [43] do not include this meta data in their compression ratio evaluations of BΔI. Compression with accounting for meta data overhead represents implementation at the off-chip memory level where it is impractical to add extra bits to the DRAM or SSD architecture for the purpose of storing compression meta data. Figure 3.9 shows that compression without meta data consistently produces modestly higher compression ratios than with meta data. It also shows that all compression ratios for $\alpha \geq 4$ are greater than the compression ratio for lossless BΔI. Throughout the remainder of our results, we focus on application of our technique to off-chip memory compression; therefore, *all compression ratios we present from here onwards do account for meta data*, unless otherwise specified.

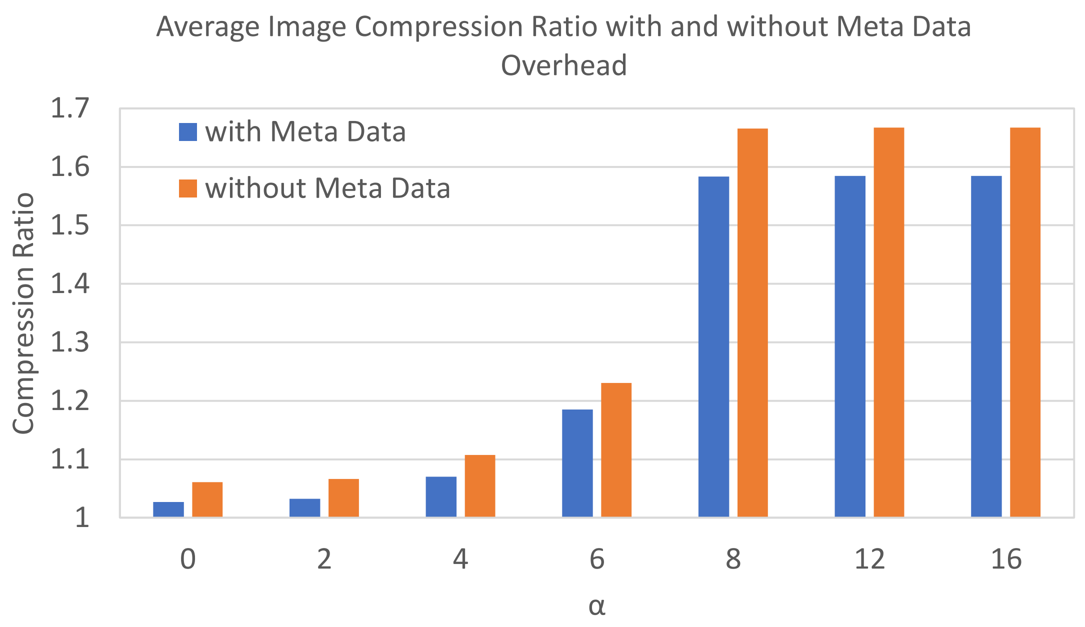


Figure 3.9: Average compression ratio with and without accounting for meta data across all 24 image categories for each level of approximation, α. The bar for α=0 without meta data represents the compression ratio of the lossless BΔI cache compression algorithm.

In Figure 3.10, we present the average compression ratio of all 40 images for each of the 24 image categories across all 7 quality levels (α = {0, 2, 4, 6, 8, 12, 16}). In the best cases, we observe image compression ratios over 1.6 for α ≥ 8. In the worst cases, we observe image compression ratios below 1.0 for α ≤ 2, which indicates very little of the data was compressible at all by BΔI. Uncompressible blocks must store 1 extra byte of meta data overhead, which decreases total compressibility for an image. We observe across all image categories that there is little to no improvements for α=2 approximation over lossless compression, α=0. α=4 shows modest improvement over loss, and α=6 shows significant improvement. The compression ratio increases significantly again at α=8, but levels off here as shown by ratios for α=12 and α=16. This leveling off is a result of the RGB24 data format. Colors red, green and blue are each represented with 8 bits (1 byte) each. Since BΔI compresses data in bases of 2, 4, and 8 bytes, consecutive RGB24 data words do not align evenly with consecutive bases. These leads to alternating cases of the red, green, and blue bytes being the least-significant byte in a block. This leads to dissimilar values in the words across a BΔI input block, which decreases compressibility. When α > 8, the approximation carries over to the next byte (color component), which is not similar to the next byte in neighboring BΔI words, so there is no improvement in encoding efficiency, as discussed in Section 3.3.

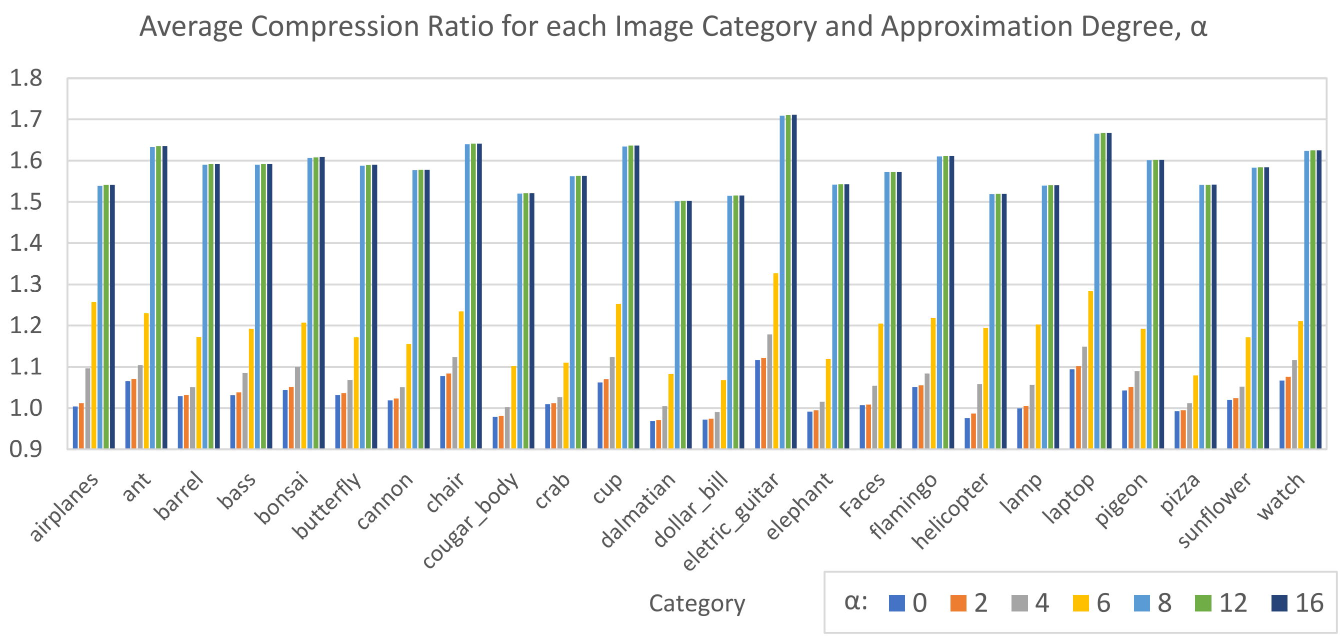


Figure 3.10: The average compression ratios (including metadata overhead) for all images from all categories and quality levels are shown. Each bar represents the average compression ratio over 40 images compressed with the specified degree of approximation. α = 0 represents lossless compression and has the lowest (sometimes less than 1.0) compression ratio.

3.5.2 Quantitative Error

While compression ratios highlight the potential storage gains from approximating media data, they do not reflect the errors induced by approximation or the impact those errors will have on application quality. Evaluating the error of the images in Figure 3.8 using only an eye test is a neither reliable nor quantitative way to do so. To accurately measure error induced by approximation, we use two mathematical evaluation techniques: PSNR and SSIM. These techniques allow us to quantitatively define the error induced by approximation, as opposed to relying solely on a qualitative eye test. Figure 3.11 and Figure 3.12 show the average compression ratio of all images versus average PSNR and SSIM respectively. According to Alvarez et al. [40], 28dB is an acceptable level of quality for PSNR for lossy compressed images. The PSNR levels in Figure 3.11 show that acceptable image quality is maintained for α ≤ 6.

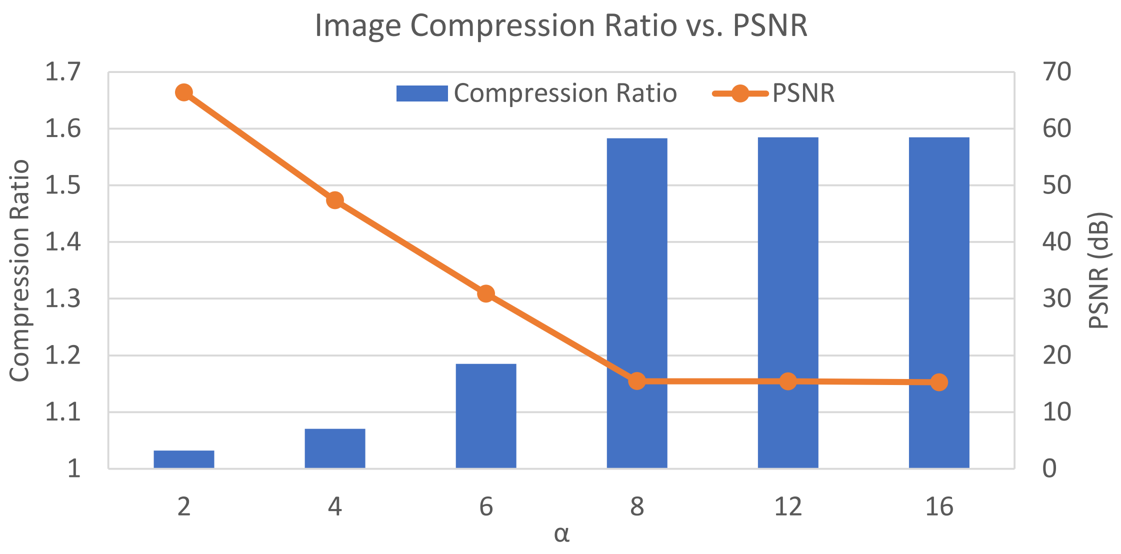


Figure 3.11: Average compression ratio and average peak signal-to-noise (PSNR) for all images are shown as a function of α. α = 0 is excluded because PSNR is infinite when there is no loss between images.

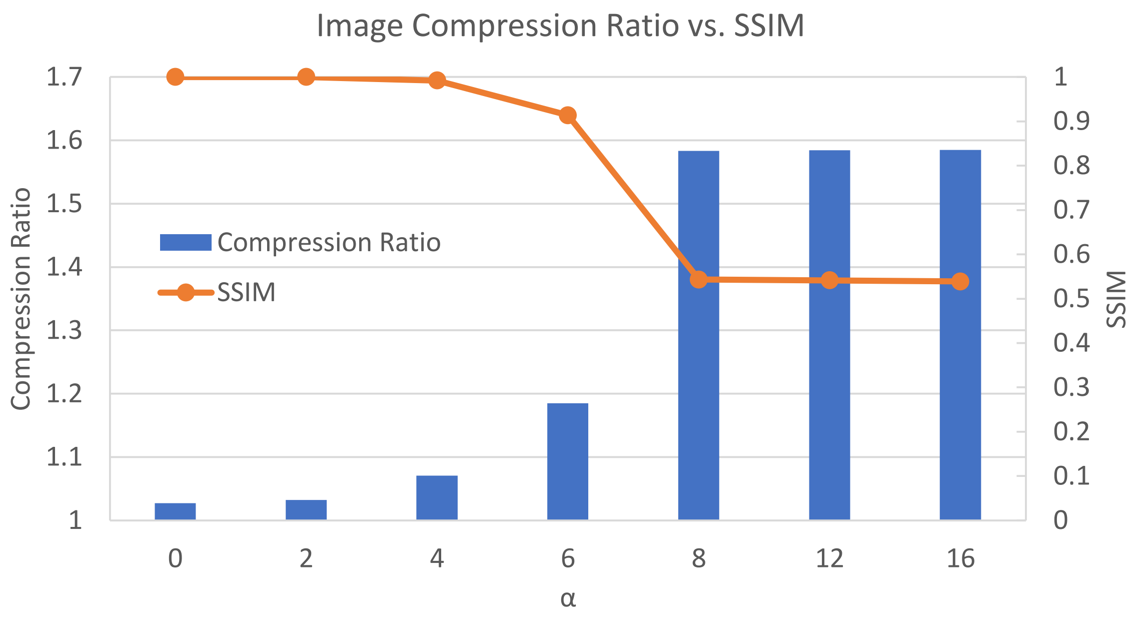


Figure 3.12: Average compression ratio and average structural similarity index (SSIM) for all images are shown as a function of α.

3.5.3 Image Classification Accuracy

Error metrics are useful for quantitatively evaluating error for lossy compressed images, but they do not show the potential impact on application quality. Here we evaluate the impact of image quality loss on the accuracy of the Clarifai general object recognition application based on deep CNNs [68]. In Figure 3.13 and Figure 3.14 we present the results of our experiment described in Section 3.4.2. Figure 3.13 shows average image compression ratio and concept prediction probability for every simulated level of α. Figure 3.14 shows the relative improvement and degradation of compression ratio and concept prediction probability respectively.

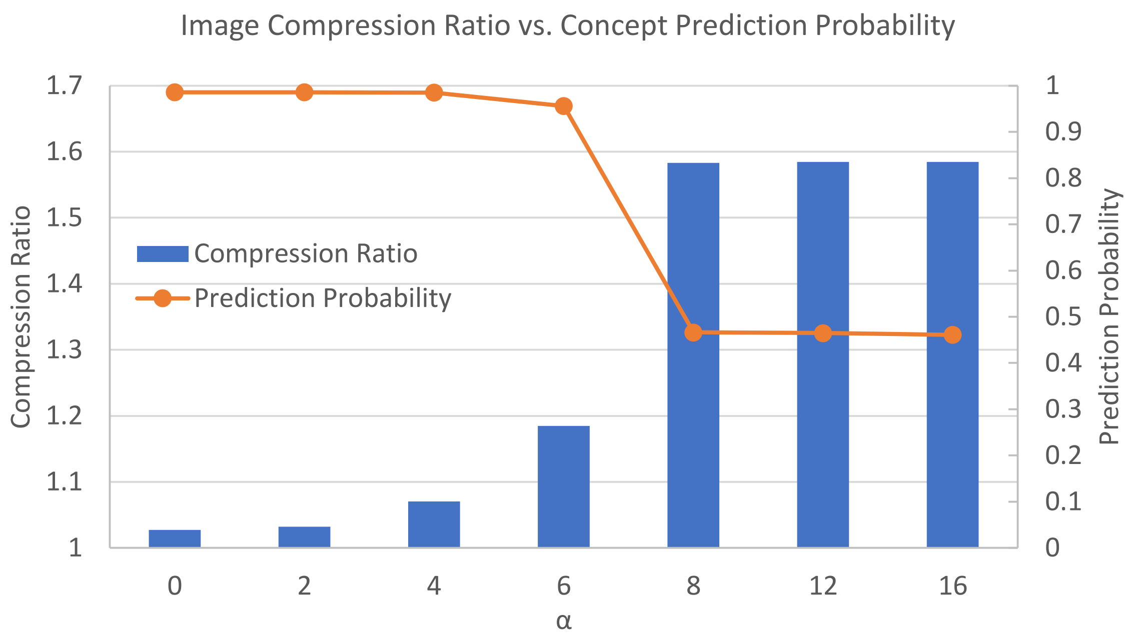


Figure 3.13: Average compression ratio and average concept prediction probability for all images are shown as a function of α.

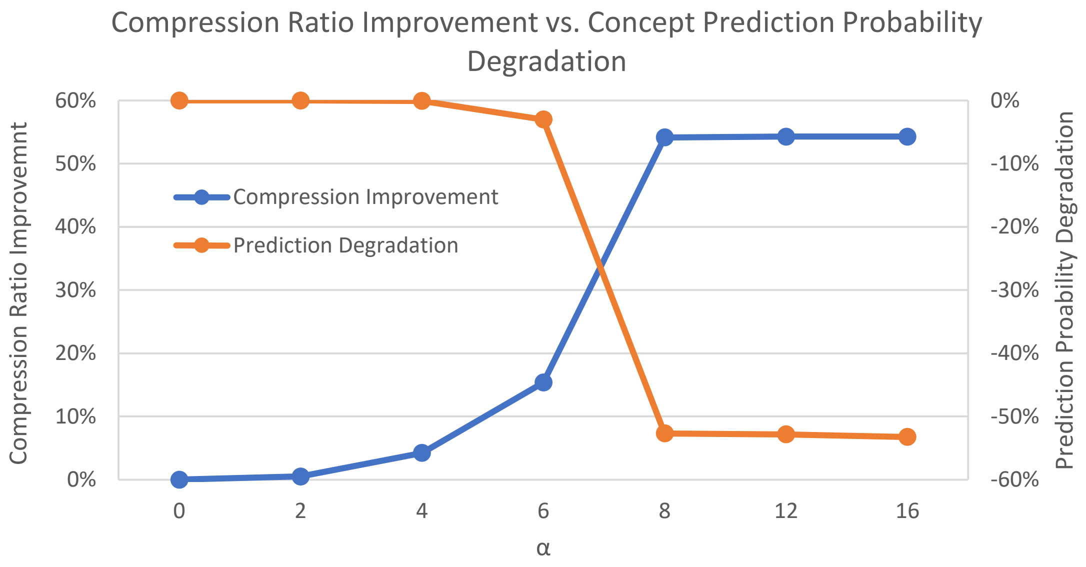


Figure 3.14: The average improvement in compression ratio and the average degradation in concept prediction probability for all images are shown as a function of α.

For α ≤ 4, there is negligible loss in prediction probability, but compression ratios are low (< 1.1). This indicates that low levels of approximation on image data will not affect application output quality, but offers no practical increase in effective storage capacity. For α ≥ 8, compression ratio increases to nearly 1.6, but the prediction probability falls to less than 50%. This is an unacceptable amount of quality loss, which makes α ≥ 8 impractical to use. When α = 6, there is a slight drop in prediction probability and the compression ratio improves to nearly 1.2. Relative to the baseline lossless compression (α = 0), prediction probability degrades by 3% and the compression ratio improves by over 15%. Most of the literature in Section 2 considers quality loss below 5% (sometimes 10%) to be acceptable. This means that α = 6 provides an optimal quality-energy tradeoff because it can significantly improve effective storage capacity for an acceptable level of quality loss.

3.5.4 Hardware

In this subsection we show that our approximate storage technique can significantly reduce DRAM access energy consumption by reducing the size of the data that needs to be transported. To evaluate the energy savings, we must first consider the overhead of implementing our technique.

The simulated hardware metrics of our design are presented in Table 3.3. Our RTL model synthesized with 65nm tech shows low power overhead, low latency overhead, and moderate area overhead. Note that this model is capable of implementing α for all values in the range [0, 31], which is much more complex than implementing just the presented α values. Therefore, these results represent an upper bound on area and power overhead. In Table 3.4 and Table 3.5 we present the area and power overheads respectively of the shifters added to the BΔI compression technique. These tables show the major drawback to our technique is the significant increase in area and power relative to lossless BΔI.

Table 3.3: Lossy compression and decompression hardware simulation metrics in 65nm TSMC.

Module	Compression	Decompression	Total
Avg. Power (mW)	1.506	0.467	1.974
Latency (Cycles)	1	1	2
F_{max} (MHz)	210	360	360
Area (mm^2)	0.06245	0.03478	0.09723
Area (kGE)	53.39	29.74	83.13
Throughput (Bytes/Cycle)	32	32	64
Avg. Energy per Operation (pJ)	7.165	1.298	8.463

Table 3.4: Lossy compression and decompression shifters area (mm^2) overhead in 65nm TSMC.

Module	BΔI Area (mm^2)	Shifters Area (mm^2)	Total Area (mm^2)	Shifters Overhead
Compression	0.0422	0.0203	0.0624	48.0%
Decompression	0.0199	0.0149	0.0348	74.7%
Total	0.0621	0.0351	0.0972	56.6%

Table 3.5: Lossy compression and decompression shifters power (uW) overhead 65nm TSMC.

Module	BΔI Power (uW)	Shifters Power (uW)	Total Power (uW)	Shifters Overhead
Compression	0.892	0.614	1.506	68.8%
Decompression	0.334	0.133	0.467	39.7%
Total	1.227	0.747	1.974	60.9%

To evaluate energy savings, we consider the average energy cost of performing all the DRAM accesses necessary to store a whole image using access operation data from [6]. Table 3.6 shows the average energy cost of performing enough DRAM accesses to store an image from our test dataset for the uncompressed size, BΔI compressed size, and approximated sizes. We compare this data with the average energy cost of performing compression or decompression for a whole image, shown in Table 3.7, and find that total overhead energy cost is less than 0.1%. This great disparity in total energy cost is indicative of the performance bottleneck in the memory hierarchy, where on-chip operations are much more efficient than off-chip ones. It also indicates that while the added shifters significantly increase power and area relative to lossless BΔI, the total energy overhead is still negligible.

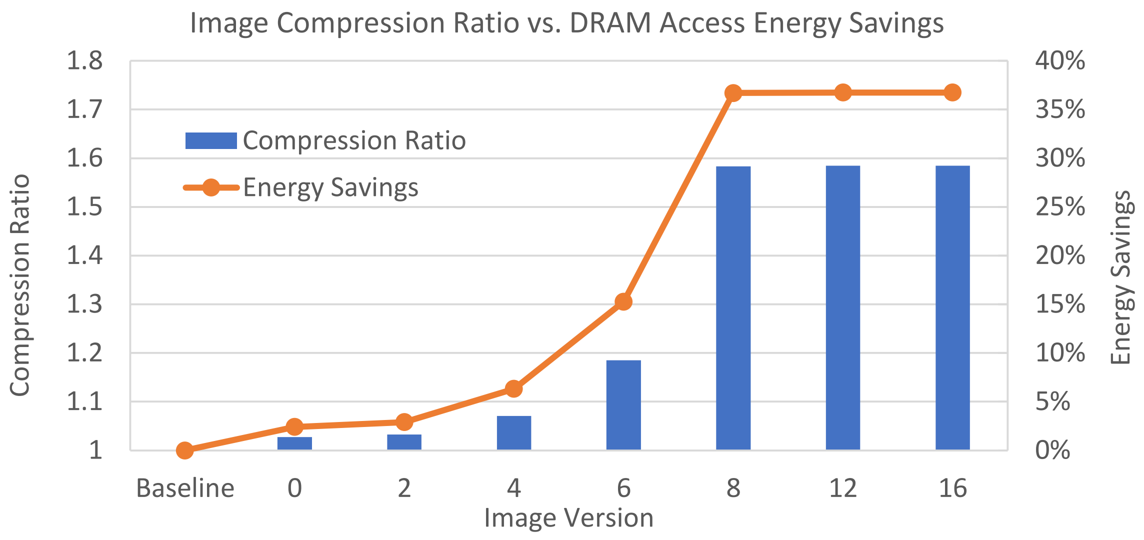
Table 3.6: Average DRAM access energy cost of storing the average original, BΔI compressed, and approximately stored images. DRAM access energy costs derived from Table 1.1 [6].

Image Version	Original	α = 0	α = 2	α = 4	α = 6	α = 8	α = 12	α = 16
Avg. Energy Cost (uJ)	67.26	65.63	65.32	63.00	56.86	42.53	42.50	42.50

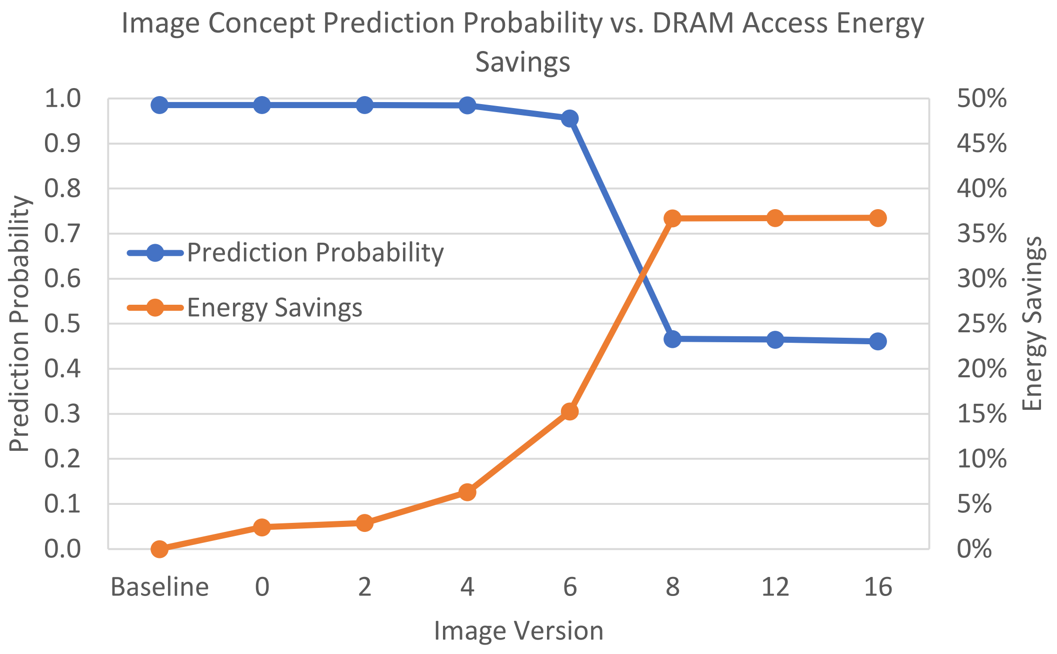
Table 3.7: Average compression and decompression energy costs per image and relative to DRAM access energy cost per image.

Operation	Compression	Decompression	Total
Avg. Energy Cost per Image (uJ)	0.0471	0.00852	0.0556
Energy Cost_{Op} / Energy $\text{Cost}_{\text{Access}}$	0.070%	0.013%	0.083%

In Figure 3.15 we present the average DRAM access energy savings for different versions of an image compared with (a) average compression ratio and (b) average concept prediction probability. For both figures, baseline is the original, uncompressed image size and the remaining values represent the approximation degree, α. As expected, there is a direct correlation between compression ratio and energy savings. As image size decreases, less data must be transported and the total off-chip energy cost can be mitigated. The figures show that energy savings are about 15% for α = 6, the approximation degree where concept prediction probability for an image is still at an acceptable level.



(a)



(b)

Figure 3.15: Average DRAM access energy savings are presented with (a) average compression ratio and (b) average concept prediction probability for all images as a function of image version. Baseline represents the original, uncompressed image and all other values represent the value α that the image was approximated with.

4 Conclusions

In this thesis, we discussed and presented *approximate storage* techniques for improving the energy consumption and performance of the memory subsystem for data-intensive RMS applications.

In Chapter 2 we reviewed the state of the art in approximate storage at the circuit, architecture, and algorithm levels. Circuit level techniques approximate data passively by allowing errors to occur in the physical storage medium. Reducing error guard-bands through SRAM voltage scaling, DRAM refresh rate reduction, and reliable access of MLC and STT-MRAM creates significant energy and power savings. Unlike circuit level techniques, architecture and algorithm level techniques deterministically approximate data. At the architecture level, programming support and ISA extensions enable designers to control and monitor approximate storage systems. Works in approximate load value prediction and approximate caches reduce the number of off-chip memory accesses by allowing processors to compute on approximately similar data, which significantly improves application performance. Algorithmic techniques such as precision scaling and look-up tables can be easily implemented in existing architectures and can be applied to data at any level of the memory hierarchy.

In Chapter 3, we presented a novel lossy compression technique based on *dynamic precision scaling* to enable approximate storage for media data. We observe that whereas small degrees of approximation bring marginal performance benefits and high degrees lead to unacceptable quality degradation, there exists an optimal point ($\alpha = 6$) that can yield significant performance improvements with low quality loss and low overhead costs. We find an increase in effective storage capacity of 15% and subsequently a reduction in DRAM access energy cost of 15% for only 3% loss in image classification accuracy and less than 0.1% total energy overhead. We conclude that approximate storage is an effective technique for decreasing off-chip memory access costs for media data.

Bibliography

1. V. K. Chippa, S. T. Chakradhar, K. Roy and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2013, pp. 1-9.
2. S. T. Chakradhar and A. Raghunathan, "Best-effort computing: Re-thinking parallel software and hardware," *Design Automation Conference*, Anaheim, CA, 2010, pp. 865-870.
3. Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages. DOI: https://doi.org/10.1145/2893356
4. J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," *2013 18th IEEE European Test Symposium (ETS)*, Avignon, 2013, pp. 1-6.
5. V. Sze, "Designing Hardware for Machine Learning: The Important Role Played by Circuit Designers," in *IEEE Solid-State Circuits Magazine*, vol. 9, no. 4, pp. 46-54, Fall 2017.
6. A. Pedram, S. Richardson, M. Horowitz, S. Galal and S. Kvatinsky, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," in *IEEE Design & Test*, vol. 34, no. 2, pp. 39-50, April 2017.
7. A. Raha and V. Raghunathan, "Synergistic Approximation of Computation and Memory Subsystems for Error-Resilient Applications," in *IEEE Embedded Systems Letters*, vol. 9, no. 1, pp. 21-24, March 2017.
8. Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: saving DRAM refresh-power through critical data partitioning. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 213-224.
9. Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X. and Solihin, Y. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. ISCA, 2009.

10. M. Alioto, "Energy-quality scalable adaptive VLSI circuits and systems beyond approximate computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017,* Lausanne, 2017, pp. 127-132.
11. J. Lucas, M. Alvarez-Mesa, M. Andersch, and B. Juurlink, “Sparkk: Quality-scalable approximate storage in DRAM,” in The Memory Forum, 2014, pp. 1–9.
12. J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, “RAIDR: Retention-aware intelligent DRAM refresh,” in Proc. 39th Annu. Int. Symp. Comput. Archit., 2012, pp. 1–12.
13. A. Raha, S. Sutar, H. Jayakumar and V. Raghunathan, "Quality Configurable Approximate DRAM," in *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1172-1187, 1 July 2017.
14. X. Li and D. Yeung, "Application-Level Correctness and its Impact on Fault Tolerance," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, 2007, pp. 181-192.
15. A. Sampson, J. Nelson, K. Strauss and L. Ceze, "Approximate storage in solid-state memories," *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Davis, CA, 2013, pp. 25-36.
16. Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). ACM, New York, NY, USA, 164-174.
17. Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '10). ACM, New York, NY, USA, 198-209.
18. A. Ranjan, A. Raha, V. Raghunathan and A. Raghunathan, "Approximate memory compression for energy-efficiency," *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED),* Taipei, 2017, pp. 1-6.
19. X. Xu and H. H. Huang, "Exploring Data-Level Error Tolerance in High-Performance Solid-State Drives," in *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 15-30, March 2015.

20. B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Edmonton, AB, 2014, pp. 493-494.
21. A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu and T. C. Mowry, "Mitigating the Memory Bottleneck With Approximate Load Value Prediction," in *IEEE Design & Test*, vol. 33, no. 1, pp. 32-42, Feb. 2016.
22. J. S. Miguel, M. Badr and N. E. Jerger, "Load Value Approximation," *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, 2014, pp. 127-139.
23. O. Kislal, M. T. Kandemir and J. Kotra, "Cache-Aware Approximate Computing for Decision Tree Learning," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, 2016, pp. 1413-1422.
24. M. Shoushtari, A. BanaiyanMofrad and N. Dutt, "Exploiting Partially-Forgetful Memories for Approximate Computing," in *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 19-22, March 2015.
25. F. Frustaci, D. Blaauw, D. Sylvester and M. Alioto, "Approximate SRAMs With Dynamic Energy-Quality Management," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 6, pp. 2128-2141, June 2016.
26. P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. Technology@Intel Magazine, February 2005.
27. R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," in *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256-268, Oct. 1974.
28. D. Geer, "Chip makers turn to multicore processors," in *Computer*, vol. 38, no. 5, pp. 11-13, May 2005.
29. H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling," *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, San Jose, CA, 2011, pp. 365-376.
30. Daniel A. Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (June 2015), 56-68.

31. Lotfi A. Zadeh. 1994. Fuzzy logic, neural networks, and soft computing. *Commun. ACM* 37, 3 (March 1994), 77-84.
32. Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. 2018. A Taxonomy of General Purpose Approximate Computing Techniques. *IEEE Embed. Syst. Lett.* 10, 1 (March 2018), 2-5.
33. Peng H. Ang, Peter A. Ruetz, and David Auld. 1991. Video compression makes big gains. *IEEE Spectr.* 28, 10 (October 1991), 16-19.
34. M. Breuer, "Hardware that produces bounded rather than exact results," *Design Automation Conference*, Anaheim, CA, 2010, pp. 871-876.
35. Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (ESEC/FSE '11). ACM, New York, NY, USA, 124-134. DOI=http://dx.doi.org/10.1145/2025113.2025133
36. Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XVII). ACM, New York, NY, USA, 301-312.
37. Ye Tian, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. 2015. ApproxMA: Approximate Memory Access for Dynamic Precision Scaling. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI* (GLSVLSI '15). ACM, New York, NY, USA, 337-342.
38. R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477)*, San Diego, CA, USA, 1999, pp. 30-35.
39. Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-46). ACM, New York, NY, USA, 1-12.

40. C. Alvarez, J. Corbal and M. Valero, "Fuzzy memoization for floating-point multimedia applications," in *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922-927, July 2005.
41. J. Park, J. H. Choi and K. Roy, "Dynamic Bit-Width Adaptation in DCT: An Approach to Trade Off Image Quality and Computation Energy," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 5, pp. 787-793, May 2010.
42. J. S. Miguel, J. Albericio, A. Moshovos and N. E. Jerger, "Doppelgänger: A cache for approximate computing," *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, 2015, pp. 50-61.
43. Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (PACT '12). ACM, New York, NY, USA, 377-388.
44. J. S. Miguel, J. Albericio, N. E. Jerger and A. Jaleel, "The Bunker Cache for spatio-value approximation," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, 2016, pp. 1-12.
45. K. Barker et al., "PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual", *Pacific Northwest National Laboratory and Georgia Tech Research Institute*, December 2013.
46. A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," in *IEEE Design & Test*, vol. 34, no. 2, pp. 60-68, April 2017.
47. James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<*T*>: a first-order type for uncertain data. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (ASPLOS '14). ACM, New York, NY, USA, 51-66.
48. P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, 1991, pp. 232-236.

49. Y. Tian, T. Wang, Q. Zhang and Q. Xu, "ApproxLUT: A novel approximate lookup table-based accelerator," *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD),* Irvine, CA, 2017, pp. 438-443.
50. W. Kim, D. M. Brooks and G. Wei, "A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation," *2011 IEEE International Solid-State Circuits Conference,* San Francisco, CA, 2011, pp. 268-270.
51. M. Cho, J. Schlessman, W. Wolf and S. Mukhopadhyay, "Reconfigurable SRAM Architecture With Spatial Voltage Scaling for Low Power Mobile Multimedia Applications," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 1, pp. 161-165, Jan. 2011.
52. E. Chen *et al*., "Advances and Future Prospects of Spin-Transfer Torque Random Access Memory," in *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 1873-1878, June 2010.
53. A. Ranjan, S. Venkataramani, X. Fong, K. Roy and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC),* San Francisco, CA, 2015, pp. 1-6.
54. A. M. H. Monazzah, M. Shoushtari, S. G. Miremadi, A. M. Rahmani and N. Dutt, "QuARK: Quality-configurable approximate STT-MRAM cache by fine-grained tuning of reliability-energy knobs," *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED),* Taipei, 2017, pp. 1-6.
55. Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-46). ACM, New York, NY, USA, 1-12.
56. A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy and A. Raghunathan, "STAxCache: An approximate, energy efficient STT-MRAM cache," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, 2017, pp. 356-361.
57. H. Naemi et al. STTRAM Scaling and Retention Failure. *Intel Technology Journal*, May 2013.
58. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti,

Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (August 2011), 1-7.

59. Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004.
60. M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, San Francisco, CA, 2014, pp. 10-14.
61. Alameldeen, A. R. and Wood, D. A. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches*. University of Wisconsin-Madison, UW-Madison, 2004.
62. Chen, X., Yang, L., Dick, R. P., Shang, L. and Lekatsas, H. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 8 (Aug. 2010 2010), pp. 1196-1208.
63. Mittal, S. and Vetter, J. S. A Survey of Architectural Approaches for Data Compression in Cache and Main Memory Systems. IEEE Trans. Parallel Distrib. Syst., 2016.
64. Sathish, V., Schulte, M. J. and Kim, N. S. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In PACT, 2012.
65. Ekman, M. and Stenstrom, P. A Robust Main-Memory Compression Scheme. In ISCA-32, 2005.
66. Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A. and Mowry, T. C. Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In MICRO-46, 2013.
67. L. Fei-Fei, R. Fergus and Perona, P. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. In Proceedings of the IEEE CVPR (2004).
68. https://clarifai.com/
69. https://www.cadence.com/
70. https://www.mathworks.com/help/images/ref/ssim.html

71. https://www.mathworks.com/help/vision/ref/psnr.html

72. Y. Chen et al., "Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications," in Proceedings of the IEEE, vol. 96, no. 5, pp. 790-807, May 2008.

73. Evans, D. (2011). The Internet of Things: How the Next Evolution of the Internet is Changing Everything. Cisco Internet Business Solutions Group (IBSG). 1. 1-11.

74. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet large scale visual recognition challenge,” Int. J. Computer Vision, vol. 115, no. 3, pp. 211–252, 2015.

75. A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," Proceedings. 31st Annual International Symposium on Computer Architecture, 2004., Munchen, Germany, 2004, pp. 212-223.

76. V. Wong, M. Horowitz, "Soft Error Resilience of Probabilistic Inference Applications", Workshop on Silicon Errors in Logic-System Effects, 2006.

77. Krishna V. Palem, Lakshmi N.B. Chakrapani, Zvi M. Kedem, Avinash Lingamneni, and Kirthi Krishna Muntimadugu. 2009. Sustaining moore's law in embedded computing through probabilistic and approximate design: retrospects and prospects. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems* (CASES '09). ACM, New York, NY, USA, 1-10.

78. Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. QAPPA: A framework for navigating quality-energy tradeoffs with arbitrary quantization. Technical Report CMU/CSE-17-03-02, March 2017.

79. Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. 2010. ERSA: error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE '10). European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1560-1565.

80. F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester and M. Alioto, "13.8 A 32kb SRAM for error-free and error-tolerant applications with dynamic energy-quality management in 28nm CMOS," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, 2014, pp. 244-245.

81. Filipe Betzel, Karen Khatamifard, Harini Suresh, David J. Lilja, John Sartori, and Ulya Karpuzcu. 2018. Approximate Communication: Techniques for Reducing Communication Bottlenecks in Large-Scale Parallel Systems. *ACM Comput. Surv.* 51, 1, Article 1 (January 2018), 32 pages.
82. I. Akturk, N. S. Kim and U. R. Karpuzcu, "Decoupled Control and Data Processing for Approximate Near-Threshold Voltage Computing," in IEEE Micro, vol. 35, no. 4, pp. 70-78, July-Aug. 2015.
83. Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (ICSE '10), Vol. 1. ACM, New York, NY, USA, 25-34.
84. V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar and A. Raghunathan, "Scalable Effort Hardware Design," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 9, pp. 2004-2016, Sept. 2014.
85. A. G. M. Strollo and D. Esposito, "Approximate computing in the nanoscale era," 2018 International Conference on IC Design & Technology (ICICDT), Otranto, 2018, pp. 21-24.
86. S. Borkar, "How to stop interconnects from hindering the future of computing!," 2013 Optical Interconnects Conference, Santa Fe, NM, 2013, pp. 96-97.