Energy Efficient Sensor Data Logging with Amnesic Flash Storage

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

We present FlashLogger, an energy-efficient sensor data logging system that uses lazy amnesic compression in a flash-efficient manner. FlashLogger incorporates a suite of compression algorithms suitable for progressively compressing time series scalar, audio, and image data. It uses a novel data structure for efficiently organizing and querying compressed data on flash memory. All our methods are designed for the limited memory and processing capabilities typical of low power sensor nodes, and are prototyped on Tmote Sky platform running TinyOS. Evaluation of FlashLogger with several real world data sets shows orders of magnitude energy savings for both logging data and retrieving data within a time range.

Categories and Subject Descriptors: E.2 [**Data Storage Representation**]: Composite structures, H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms: Algorithms, Performance, Theory

Keywords: amnesic compression, flash memory

1. INTRODUCTION

We present techniques to efficiently use flash storage for logging data on a sensor node. We focus on *amnesic* storage systems that have a number of properties particularly attractive for sensing applications. An amnesic storage system archives streaming data using two key techniques. First, data is compressed (usually with lossy compression methods) in an online fashion before being archived. Using compression provides two major benefits: i) it enables archiving a larger amount of data, which is useful in long-lived sensor deployments where the entire data stream collected by a sensor may not fit in its flash memory, and ii) compression helps reduce the overall energy consumption for data archival.

The second key technique an amnesic storage system uses is *aging* archived data—by reducing the fidelity of older data to make space for newer data. Data is aged by progressively recompressing it with lower fidelity (e.g., higher decompression error of scalar data, lower resolution of image data, etc.) Such aging (also known as amnesic compression [12, 13], time-decaying compression [2], and multi-resolution compression [5]) is natural in many sensing applications where older data is less important than newer data. Moreover, it can enable archiving data over a longer period of time. For example, in the Environmental Observation and Forecasting System [18], sensor stations need to locally buffer data during occasional network disconnections. However, since a station does not know how long it will be offline and has a finite buffer, it needs to use amnesic compression. Similar use of data aging has been reported in remote sensing [4], sensor-assisted robots [6], sensor data aggregation [2], multi-resolution in-network storage of sensor data [5], etc.

In addition to showing the usefulness of amnesic compression, existing works have also proposed algorithms to optimize for various objective functions. For example, in DIMENSIONS [5], authors used *lazy schemes* to age and distribute multi-resolution compressed data within the network for efficient iterative, drill-down queries. In [12, 13], authors presented *aggressive schemes* to maintain amnesic compressed streaming data with optimal decompression error or storage footprint.

In this paper, we focus on a complimentary problem to the amnesic compression problem: efficiently storing and indexing amnesic compressed streaming data on flash memory. The problem is important because flash I/O cost can be excessively high without a suitable scheme to organize archived data on flash. This is particularly true for aggressive schemes (more details in Section 6). Interestingly, we show that even a lazy scheme can incur a significantly high I/O cost if the physical layout of compressed data on flash is not chosen carefully. This may not be a major concern if the I/O cost is negligible (e.g., when data is maintained in DRAM), or can be ignored (e.g., when available energy or tolerable latency is practically infinite); but in a flash-equipped energy-constrained sensor node, neither of these holds. Moreover, flash memory wears out after a limited number of erase/write operations, further emphasizing the requirement of I/O efficient archival algorithm.

In this paper, we address the problem by designing and implementing FlashLogger, a sensor data logging system that enables energy-efficient amnesic compression on flash memory. FlashLogger consists of two main modules. First, its Compression Module enables applications to use a suitable

IPSN'09, April 13–16, 2009, San Francisco, California, USA.

amnesic compression algorithm that can compress (scalar, audio, or image) data for a given fidelity requirement. In contrast to existing error- or space-efficient amnesic compression schemes, our scheme is I/O-efficient—the I/O cost is reduced by compressing data in a lazy fashion (like [5]), which also enables efficient I/O batching. An application can extend this module by plugging in its own compression algorithm.

Second, FlashLogger provides Am-Store (Amnesic Storage), a novel flash-efficient abstraction for archiving amnesic compressed data. Maintaining such data on NAND flash is challenging because of its unique write- and erase-constraints. Unlike streaming compression algorithms [1] that only need to append compressed data at one end of the archive, amnesic compression requires updating and overwriting existing compressed data. Such operations are very expensive on flash memory. Am-Store uses a number of techniques to reduce such expensive operations. Finally, Am-Store maintains a skip-list based index of its archived data that enables FlashLogger to efficiently retrieve compressed data within a given time range.

We have evaluated FlashLogger using two prototypes. Our first prototype is written in nesc and has been evaluated with Moteiv Tmote Sky nodes. The second prototype is written in C and has been compiled for a PC-class, as well as for MSP430 and ARM processors. The first prototype uses sensor's on-board flash chip, and the second prototype uses an external flash chip and a compact flash card. Evaluation of FlashLogger with several real world data sets (including scalar, audio, and image data) shows significant energy savings for both logging and retrieving data within a time range.

In summary, we make the following contributions.

1. We show that existing amnesic algorithms, even lazy ones, perform poorly on flash, and they are not suitable for flash-equipped energy-constrained sensor nodes.
2. We propose FlashLogger, an energy efficient sensor data logging system that uses online amnesic compression of sensor data in a flash-aware manner. FlashLogger incorporates a suite of compression algorithms for numeric, audio, and image data, and is extensible.
3. We present Am-Store, a novel and efficient abstraction for maintaining amnesic compressed data on flash memory. It also provides techniques to efficiently retrieve compressed data within a given time range.
4. We evaluate two prototypes of FlashLogger with real senor platforms, flash memory, and datasets. Our results demonstrate orders of magnitude energy savings for both logging and retrieving data.

Unlike previous work [16, 17], the goal of this paper is not to design new compression algorithms for sensor networks, rather to show how to efficiently maintain compressed data on flash. We use particular compression algorithms for concreteness of our implementation and evaluation. On the other hand, our effort is complementary to some prior work; for example, in DIMENSIONS, a sensor node can use FlashLogger to efficiently organize compressed data on its local flash.

Task		Energy overhead (μJ)
Compression	MSP430	4.4
	ARM	7.6
Decompression	MSP430	4.2
	ARM	7.2
Fastest sequential I/O in a Lexar CF card	Read	109.6
	Write	101.6
Fastest random I/O in a Lexar CF card	Read	211.2
	Write	7945.1
I/O in a Toshiba TC58DVG02A1FT00 flash chip [8]	Read	462.6
	Write	598.5
	Erase	3429
I/O in an external FujiFilm XD card flash chip	Read	343.2
	Write	350.1
	Erase	7891

Table 1: Energy overhead of compressing and accessing 4KB data with different processors and flash media.

The rest of the paper is organized as follows. Section 2 discusses design considerations and Section 3 characterizes an amnesic storage system. Sections 4 and 5 present FlashLogger. We present our implementation details and evaluation results in Section 6. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2. DESIGN CONSIDERATIONS

In this section, we first discuss various constraints that make the design of FlashLogger challenging. We then list a few design principles that result from these constraints.

2.1 Constraints

Flash Constraints. A NAND flash memory consists of many blocks, and each block consists of a fixed number of pages. For example, the Telos mote uses the ST Microelectronics M25P family of flash memories; this memory chip has 16 64KB blocks, each block consisting of 256 256B pages. Reads and writes happen at a page granularity, while erases happen at a block granularity. A unique property of flash memory is its block-erase constraint—once a page is written to flash, it cannot be updated or rewritten until the entire block containing the page is erased. Moreover, a flash block wears out after a limited number of erases. Flash packages such as SD cards and CF cards employ a *flash translation layer* (FTL) to hide many of these complexities and to expose a disk-like interface to applications. However, since the FTL needs to internally deal with flash constraints, random I/O operations are very expensive. More details can be found in [8, 10, 11].

Energy Constraints. Our target sensor platforms operate on limited battery power. Table 1 shows the measured energy overheads of two processors, typically used in low power sensor nodes, to compress and decompress 4KB data (Temperature data used in Section 6). For this measurement, we consider lossless QuickLZ algorithm [15], although the compression algorithms we use in our prototype are much simpler than QuickLZ. The results show that flash I/O is significantly more expensive (more than two orders of magnitude) than compressing/decompressing data. Therefore, optimizing for I/O cost is more important than optimizing for compression cost.

Memory Constraints. Many of today's sensor platforms have very limited available memory (RAM). This makes many simple optimizations infeasible. Consider time-based indexing of logged sensor data. Sensor data is typically time-stamped and accessed sequentially in time, potentially within given time ranges. For instance, in an event reporting network, the base station may request logged data from a sensor node within a time window surrounding a reported event. To efficiently support such queries, a time-based index needs to be maintained over compressed data. One may consider using existing flash-friendly log-structured index (e.g., FlashDB [11]) or file system (e.g., ELF [3]). However, while such a design significantly reduces write cost, it increases query cost. Existing systems are designed for a relatively small number of unique keys (or names) for lookup, and their memory footprints of frequently accessed data structures increase with the number of unique keys. Building an index on timestamps implies a large number of unique keys, which further implies a very large memory footprint in FlashDB. Log-structured file systems do not support range lookup; and building one on top of them would require FlashDB-like structure with a large memory footprint. Therefore, these existing schemes can not readily be used for building a time-based index in a memory constrained platform.

2.2 Design Principles

Under the above constraints, the design of FlashLogger should follow a few design principles. First, the system should avoid or minimize I/O operations that are expensive on flash memory. Such operations include in-place update in a flash chip, random writes in a flash package with FTL, and deletion or allocation/deallocation of space in a sub-block granularity. Moreover, I/Os should be batched to match the write granularity of flash memory. Existing flash-aware systems use different techniques to achieve these goals [3, 8, 10, 11].

Second, the system should aggressively optimize to reduce I/O cost, even if it comes with a small additional processing cost. This can be done by using flash-friendly I/Os (mentioned above), by being lazy and batching I/Os (like DIMENSIONS), and by using better compression algorithms.

Finally, in order to reduce memory footprint, the system should maintain most of its index structure in flash. In-memory structures (e.g., buffers, frequently modified parts of the index) should be as small as possible.

3. AMNESIC COMPRESSION

We consider lossy compression and archival of a potentially long data-stream in a limited storage. We consider an *amnesic system*, where an application prefers keeping old data, rather than discarding them, by reducing their fidelity (i.e., by aging them) to make space for newer data. For example, an image sensing application may prefer to keep older images with lower resolutions than newer images. In general, the data distortion ϵ of compressed data gradually increases with the age of data. When application needs space for newly compressed data and the flash is full, it further compresses older compressed data with a higher ϵ value to free up space. If old data can not be meaningfully compressed within the limited space, it is discarded. Suppose, at any point of time T, the system contains all (compressed) data that arrived after T'; then we call the time range $[T', T]$ the *active window*.

An amnesic system consists of two key components: an amnesic function and an amnesic compression algorithm.

3.1 Amnesic Error Function

The function $E(x)$, which bounds the amount of error a piece of data with a given age $x \geq 0$ can have, is called the *amnesic function*. The age x is defined as the time difference between the current time and the time when the data arrived the system. $E(x)$ can be absolute or relative; in the former, $E(x)$ is given by an absolute value, while in the latter, $E(x)$ is given as a function of $E(0)$ and x. For example, a biologist may decide that data that is twice as old can have twice as much error, and thus, specify a relative linear amnesic function $E(x) = xE(0)$. In contrast, an environmental scientist using classic models might well specify an absolute exponential amnesic function $E(x) = a^x$.

A key property that an amnesic function has to satisfy is its error-monotonicity property.

DEFINITION 1. *An amnesic error function $E(x)$ is called monotonic if and only if $E(x) \leq E(x+1)$, for every x.*

Intuitively, a monotonic amnesic function allows a piece of data to be progressively compressed with higher compression errors over time.

3.2 Amnesic Compression Algorithm

Suppose a piece of data d of size $|d|$ is compressed to d_ϵ with decompression error ϵ, where $|d| \geq |d_\epsilon|$. An amnesic compression algorithm is able to recompress d_ϵ to $d_{\epsilon'}$, for a given error $\epsilon' > \epsilon$. The key property an amnesic compression algorithm needs is its space-monotonicity property.

DEFINITION 2. *An amnesic compression algorithm is monotonic if it can compress d_{ϵ_1} to d_{ϵ_2}, $\epsilon_1 < \epsilon_2$, such that $|d_{\epsilon_1}| \geq |d_{\epsilon_2}|$, for any ϵ_1 and ϵ_2.*

Intuitively, a monotonic amnesic compression algorithm ensures that compressing with a higher error requires less storage; hence, the same piece of data can be progressively compressed more and more over time to free up additional space. Examples of such algorithms include piecewise linear approximation, wavelet transforms, etc. (Section 5.1).

Existing algorithms. Even though general compression is a well explored area, a little has been explored to use them efficiently in an amnesic setting. In [12], authors consider the problem of online amnesic compression given a storage size and a monotonic amnesic function such that, at any point of time, the decompression error is minimized. A dual problem is to optimize storage space given a maximum decompression error. We call such algorithms *error-efficient* and *space-efficient* amnesic compression algorithms respectively. An error- or space-efficient algorithm works as follows. Every

time a new data point comes, it recompresses some carefully chosen section of old (compressed) data with a lower fidelity, and uses the freed-up space for new data. In [5], authors propose a lazy scheme that compresses data in batch; batches are chosen in order to optimize drill-down query constrains and to bound decompression error below a given threshold.

3.3 I/O Efficiency

Even though these existing systems address important questions of *how often* and *which* piece of data should be compressed, they do not address the orthogonal problem we consider: how to physically organize compressed data on flash memory. In an I/O expensive media such as a flash memory, existing schemes pose several challenges:

1. Excessive I/Os. Error- or space-efficient schemes aggressively optimize for decompression error or space usage by recompressing old compressed data *on every new data arrival*. Recompressing old data requires reading them from flash, deleting them, and writing new compressed data—all of which are very expensive on flash, and can accumulate prohibitively for a long data stream. Moreover, updates are performed on small amount of data, which mismatches the access granularity on flash. For example, when data is compressed using piecewise linear approximation, each update involves reading two adjacent line segments (eight real numbers) and writing one line segment (four real numbers). Even though this involves a small amount of data, in flash, it requires at least one page read, one page write, and one block erase operation, making the operation very expensive. Moreover, such excessive I/O can quickly wear out a flash.

The above problem can be addressed by using lazy schemes such as the ones used by DIMENSIONS. However, irrespective of a scheme being aggressive or lazy, it can still incur a high flash I/O overhead unless compressed data is physically organized with a suitable data structure, as discussed below.

2. Fragmentation of Adjacent Data. In the amnesic model, compressed data with adjacent timestamps may become scattered in non-adjacent physical locations of the flash. For example, assume that starting with an empty flash, compressed data is written sequentially from the start to the end of the flash. Now, to make room for new data, some older data at its existing location, l, in the flash may be re-compressed. This will create free space near l where subsequent data can be written. But this newly written data may not be contiguous with the most recent data at the end of the flash, making the new data physically non-adjacent from previous data.

Such fragmentation is problematic if compressed data within a given time-range need to be retrieved sequentially, in order of their timestamps. Two simple data structures that can support this are an Array and a List. An *Array* keeps logically adjacent compressed data physically adjacent as well, by compacting the flash if needed after each recompression. Even though such compaction is expensive, an Array allows one to use efficient binary search to locate the starting of a time-range, and to sequentially read subsequent data. In contrast, a *List* organizes compressed data as a linked list so that following its links enables accessing data in a time-sorted order. The nodes of the List must be chained with backward pointers (i.e., newer data point to older data); forward pointers are extremely expensive in flash as older elements on the flash cannot be modified to point to newer elements without incurring high costs [8]. Compared to an Array, a List does not require expensive flash compaction, but it makes lookup operations expensive as one can no longer use binary search to locate data with a given timestamp. Thus, these two data structures pose a tradeoff between archiving and lookup cost.

3. Modification of Existing Data. Suppose the data is organized as a *List* where each node contains all compressed data within a time range. In the amnesic model, compressed data in a node N, possibly in the middle of the list, is further compressed. The modified node can be written to the same location, which requires expensive in-place update. Alternatively, the modified node can be written as a new node N' in a new location. This would require changing pointers of the existing node pointing to N. Because a pointer cannot be updated in place, this node, with pointer to N', must be written to a new location. However, this would require updating the pointer of the node that points to the updated node, and so on. Thus, recursively, many nodes would be required to be written to new locations due to a single recompression operation.

In summary, even a lazy compression scheme may be I/O expensive if compressed data is archived with a sub-optimal data structure on the flash. Next we present how FlashLogger uses an I/O efficient amnesic compression scheme that employs a carefully designed flash-friendly data structure.

4. FlashLogger

We now present FlashLogger, a flash-optimized abstraction for compressing and aging sensor data streams according to application specified amnesic function and compression algorithm. It consists of two main components: (1) the Compression Module contains a suite of amnesic compression algorithms that an application can choose from, and (2) the Am-Store (Amnesic Storage) maintains compressed data on flash, and is the main focus of this paper. Am-Store also allows applications to efficiently retrieve data with timestamps within a target range. The core compression algorithm and data structures required to organize compressed data on flash have very small memory footprint, making Am-Store suitable for power constrained devices such as motes.

4.1 Addressing the challenges

Before presenting the details of our solution, we will highlight the basic approach to address the challenges mentioned in Section 3.3.

To address the first challenge, instead of being aggressive about optimizing for decompression error or storage space, we age data *lazily*. This is illustrated in Figure 1. Given a monotonic amnesic function F_1, a space- or error-efficient algorithm would recompress old data *at every time step* to

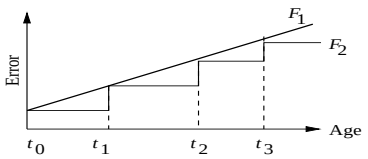


Figure 1: Piecewise constant approximation of an amnesic function

ensure that overall decompression error is minimized and is bounded by the function F_1. As mentioned before, this incurs excessive I/O cost. Instead of doing that, we batch the recompression of multiple data items periodically, and between two recompression events, fidelity of a piece of data does not change. In Figure 1, a piece of data is compressed at ages t_0, t_1, t_2, and t_3; therefore, its decompression error follows a piecewise constant approximation F_2 of F_1.

To determine F_2 for a given F_1, one needs to determine the ages t_i at which compression fidelity of a piece of data is changed. In [5], authors presented how this can be formulated as an optimization problem and be solved with simple greedy algorithms. Note that, such a lazy scheme does not compromise the semantics of amnesic compression, since F_2 is still bounded by F_1. The only side effect is that at any point of time, some data may be stored with a higher fidelity than required by the amnesic function F_1, and hence may take more space than necessary. However, we show in Section 6 that such overhead is typically small ($< 20\%$), and acceptable in situations where optimizing for energy is more critical. Moreover, such piecewise constant approximation of the error function is natural in many scenarios; for example, a sensor may decide to log its data over last one hour without any decompression error, over last one day with a small error, over last one week with a higher error, and so on.

Our lazy scheme is *I/O efficient* as it provides two advantages in reducing energy consumption. First, it reduces the frequency at which a particular piece of data is compressed (only at ages t_0, t_1, t_2, and t_3, instead of at every time step, in Figure 1). Second, it allows recompressing multiple pieces of data (e.g., all data with age between t_0 and t_1) in batches. This is extremely useful for flash, as read and write granularity is a flash page that can contain multiple pieces of data.

The second challenge mentioned in Section 3.3 comes from expensive pointer modification on flash. This is addressed by two techniques in Am-Store: i) maintaining multiple lists (called buckets), instead of one list; with this organization, we ensure that data is always inserted in the beginning of a bucket, modified or deleted at the end of a bucket, and therefore, only an in-memory pointer needs to be updated during insertion or recompression in a bucket; and ii) using timestamped pointers, which enables leaving dangling pointers to deleted items unmodified. These features are described in Section 4.3.

To address the third challenge, i.e., to support efficient lookup of data with a given timestamp, each bucket is organized as a *skip list*, which enables logarithmic time lookup. In Section 5.2, we elaborate how we maintain a skip list on flash and how we support queries on it.

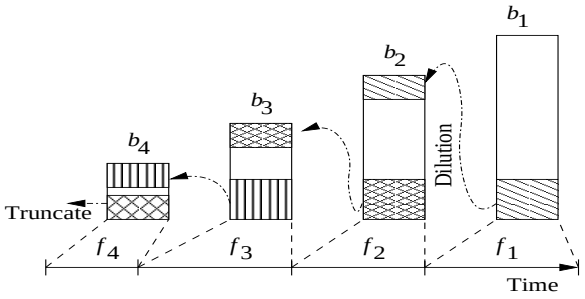


Figure 2: The Am-Store data structure. The i'th bucket b_i stores data in f_i fraction of the active window.

Algorithm 1 $Compress()$

Require: Compresses and adds a datastream to an Am-Store.
Definitions:
n: number of buckets in Am-Store
$F : \{f_1, f_2, \ldots, f_n\}$
$E : \{\epsilon_1, \epsilon_2, \ldots, \epsilon_n\}$

1: $cfile \leftarrow$ new Am-Store(n, F, E)
2: **for** each data item (v_i, t_i) in the stream **do**
3: $cfile.AddItem(v_i, t_i)$
4: **if** $cfile.size = FlashSize$ **then**
5: $cfile.Truncate()$
6: **for** each bucket i from $(n-1)$ downto 1 **do**
7: $cfile.Dilute(i)$

4.2 Am-Store overview

Given the fractions $F = \{f_1, f_2, \ldots, f_n\}$, $\sum f_i = 1$, and errors $E = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_n\}$, $\epsilon_1 < \epsilon_2 < \cdots < \epsilon_n$, Am-Store maintains an amnesic compressed archive of an evolving data stream such that data in the most recent f_1 fraction of the active window is compressed with a guaranteed maximum error of ϵ_1, that in the previous f_2 fraction of the active window is compressed with a maximum error of ϵ_2, and so on. In essence, the parameters F and E define the piecewise constant amnesic function for compression. Am-Store uses the algorithm shown in Algorithm 1, which consists of the following main ideas. First, Am-Store uses an amnesic compression algorithm (Line 3), provided by the Compression Module, as the basic building block. Concrete examples of such algorithms will be given in Section 5.1.

Second, Am-Store maintains n buckets $\{b_i\}_{i=1}^{n}$ such that bucket b_i maintains data compressed with error ϵ_i, beginning with bucket b_1 that maintains the most recent data with the smallest error ϵ_1. Figure 2 shows the basic organization of buckets.

Third, when the flash is full, we use two techniques, as shown in Figure 2, to reclaim space. The first technique is *bucket truncation* (Line 5 of Algorithm 1), which keeps on discarding oldest data from the last bucket, until enough space (at least one flash block) is reclaimed for new data. The second technique is *bucket dilution*, used in Line 7 of Algorithm 1. Starting from the second last bucket, it removes some old data d from the end of each bucket b_i, further compresses it to d' with a larger error ϵ_{i+1}, and adds it to the front

of the bucket b_{i+1}. Since $\epsilon_{i+1} > \epsilon_i$, $|d'| \leq |d|$; therefore, some space is freed-up. Conceptually, newly compressed data is added to b_1, diluted over time with higher compression errors and moved to higher indexed buckets, and finally discarded from the last bucket. We will describe these operations in more detail in Section 4.3.2.

4.3 Am-Store design

We now present the detailed design and key features of Am-Store. Logically, Am-Store consists of a set of buckets $\cup_i B_i$ stored on a flash media. At a high level, the Am-Store supports the following operators:

1. *new Am-Store(n, F, E):* Creates a new Am-Store with n buckets, such that the i'th bucket b_i contains compressed data with errors $E[i]$ over fractions $F[i]$ of active window.

2. *AddItem(v,t):* Archives data v with timestamp t. It uses the Compression Module to compress new data and adds the output to the first bucket of the Am-Store.

3. *size*: Returns the size of the entire Am-Store.

4. *Dilute(i):* Dilutes the i'th bucket b_i, by recompressing its data with error $E[i+1]$, and puts the compressed data in b_{i+1}. The space freed up by the process is reclaimed for later use.

5. *Truncate():* Discards a batch of the oldest data from the highest-indexed non-empty bucket.

4.3.1 Bucket layout and maintenance

The physical layout of buckets involves two important design decisions. The first decision involves choosing an appropriate data structure. In DRAM or magnetic disk, there are a variety of possible organizations (array, stack, queue, singly- or doubly-linked list, etc.) that may be desirable depending on the context. However, on flash, certain organizations can be extremely expensive to maintain. For example, as mentioned before, older elements on the flash cannot be modified to point to newer elements without incurring high costs; hence data structures with forward pointers, such as a queue or a doubly-linked list, are not suitable in flash. An array, although efficient, is not a suitable choice because the size of a bucket cannot be reasonably estimated a priori (this depends on the compression ratio of data). Therefore, we organize each bucket as a linked list with *backward pointers*, such that newer elements point to older elements in the list.

The second design decision involves the node size of the linked list organization of a bucket. If Am-Store has direct access to flash, possible granularities of a node size include fraction of a flash page, a flash page, and a flash block. Note that flash allows erasing data only at a block granularity. Thus, with the first two choices, when nodes are deleted during bucket dilution or truncation, valid data within the block containing the node must first be copied elsewhere. The overhead can be significant since a page is very small compared to a block. Thus, if Am-Store has access to physical flash, it keeps the node size to match with flash block size. On the other hand, if Am-Store operates over a flash package with

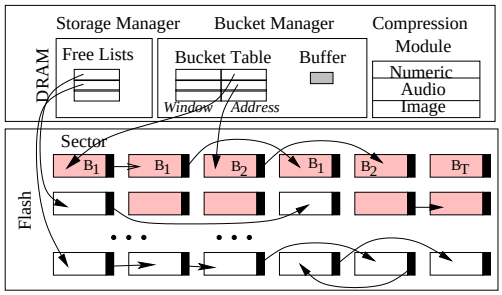


Figure 3: Physical layout of Am-Store buckets

FTL (e.g., a compact flash card), then Am-Store has no access to flash blocks but can only read, write, or allocate space in the granularity of logical sectors exposed by the FTL; therefore, in such cases, node size is kept the same as a sector size. For simplicity in the rest of paper, we use the term *sector* to denote the space taken by a node, which can be a flash block on a physical flash, or a logical sector on a flash card.

Figure 3 depicts the physical layout of Am-Store buckets. The upper box shows the in-memory portion. For each Am-Store bucket b_i, we maintain an in-memory data structure called $b_i.header$. The header contains a *sector pointer* that points to the first flash sector in that bucket, and a *window* that represents the first and the last timestamp of data in that bucket. Search or retrieval of items in a bucket starts with the sector pointer. We also maintain an in-memory *write buffer* that can temporarily hold one flash page worth of data. Note that at any point in time data is written to only one bucket—during sensor data collection, data is always added to the first bucket, and during dilution, data is transferred from the i-th bucket to the $(i+1)$-th bucket. The write buffer is shared for all writes; when an item is added to a bucket, it is temporarily put in the write buffer. When the write buffer holds enough data, the buffer is flushed to the next available pages within the sector pointed to by the sector pointer in corresponding header. Am-Store uses two modules to maintain this layout:

Bucket manager. The Bucket Manager (BM) writes the in-memory write buffer to the next available page within the sector h pointed by current bucket's sector pointer. When no empty page is available in sector h, a new sector h' is allocated by the Storage Manager (described below). A pointer to the sector h is stored at the end of sector h' and the sector pointer is updated to h'. Thus the sectors in a bucket are chained together with backward pointers and the address of the last sector is maintained in in-memory header of the bucket.

Storage manager. The Storage Manager (SM) keeps lists of available sectors and allocates them to the Bucket Manager (BM) on demand. When BM discards sectors of a bucket (during dilution or truncation), pointers to the sectors are returned to SM. When BM requests a new sector, SM selects a sector from its available sector list (erases it, if using physical flash), and returns it to BM.

Am-Store also has a crash recovery module that can re-

cover in-memory data structures to a consistent state after a system crash. We omit the details here due to lack of space.

4.3.2 *Bucket truncation and dilution*

Buckets are truncated and diluted when the Am-Store is full and must discard or recompress old data to reclaim space for new data. The process requires the following steps.

1. Bucket truncation. This discards old data in the last sector of the highest-indexed non-empty bucket and adjusts the bucket window accordingly. The sector is returned to the Storage Manager for later allocation.

2. Identifying data to dilute. Bucket b_i is supposed to have f_i fraction of the active window, and at any point of time, Am-Store can easily compute the first and the last timestamps t_1 and t_2 of the active window that should be within b_i. Thus, all data in b_i with timestamps earlier than t_1 can be diluted and put in b_{i+1}. Since Am-Store allocates and deallocates space at a sector granularity, Am-Store dilutes only the sectors all of whose data have timestamps $< t_1$. We will discuss in Section 5.2 how to efficiently locate the first such flash sector within a bucket.

3. Bucket dilution. Next, Am-Store recompresses the data selected in Step 2 with a larger error bound ϵ_{i+1}, by using application specified amnesic compression algorithm provided by Compression Module. The new data is inserted in the beginning of bucket b_{i+1}.

4. Timestamped pointer. Recall that a bucket is simply a linked list of sectors. Removing sectors from the end of a bucket makes any pointer that points to these sectors invalid. Modifying any existing pointers in place is expensive in flash. We address this by appending each pointer with the earliest timestamp in the pointed sector and maintaining in the in-memory bucket header the latest of timestamps of any diluted sectors of the bucket. We call the timestamp associated with each pointer its *look-ahead timestamp* and the timestamp maintained in bucket header the *fence* of the bucket. This enables us keeping pointers to deleted sectors unchanged, since pointers with lookahead timestamps earlier than the bucket's fence are implicitly invalid and are ignored.

5. Bucket window adjustment. Finally, the windows (in headers) of the two buckets involved in the dilution operation are adjusted to reflect the smallest and the largest timestamps of data contained in corresponding buckets.

5. COMPRESSING AND QUERYING IN FlashLogger

5.1 Amnesic compression algorithms

The Compression Module of FlashLogger provides the following collection of amnesic compression algorithms. An application can also extend the Compression Module by implementing its own compression algorithm.

Scalar data. For simple time series data, e.g., from temperature sensor, one can use a piecewise polynomial approximation scheme (such as piecewise constant, piecewise linear, etc.). We have implemented an online algorithm that compresses a numeric data-stream with a guaranteed maximum decompression error ϵ, where ϵ is tunable [1]. The algorithm converts a data-stream into a list of *segments* (Figure 4) where each segment has a *height* of 2ϵ and is represented by three parameters: t_s, t_e, and v_0, where t_s and t_e are timestamps of first and last data points in the segment, and v_0 is the minimum data point represented by the segment. An incoming data point is greedily packed into the latest segment; if it cannot be fit, a new segment is created. During decompression, any data item x_i with a timestamp between t_s and t_e is represented with the value $\hat{x_i} = v_0 + \epsilon$. It is easy to see that this invariant ensures our error bound: $|x_i - \hat{x_i}| \leq |(v_0 + 2\epsilon) - (v_0 + \epsilon)| = \epsilon$. It can also be shown that given an error bound ϵ, the above algorithm finds a minimum segment approximation whose error is at most ϵ. Dilution of data with a higher error bound $\epsilon' > \epsilon$ is done by greedily packing the existing segments into new segments with height $2\epsilon'$.

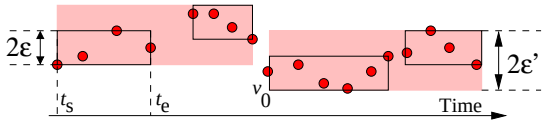


Figure 4: Compressing a numeric data stream into segments. Rectangles are segments with error ϵ, while shaded boxes are segments with error $\epsilon' > \epsilon$.

Audio data. For audio data, one can use polynomial approximation schemes, or discrete transforms such as Fourier Transform (DFT), Cosine Transform (DCT), Wavelet Transform (DWT), etc. We have implemented a DWT-based audio compression algorithm, as used in DIMENSIONS. For a given chunk of audio signal, the algorithm computes its DWT coefficients and stores n largest (alternatively, first) coefficients, where n is determined by the desired fidelity. Like previous work [17], we use *average distortion* $\delta = \sum_{i=i}^{N} |S_i - S'_i|/N$ as the fidelity metric, where S is the original signal of length N and S' is the decompressed signal. During decompression, the dropped coefficients are approximated with zeros. To dilute a compressed segment with n_1 coefficients to a lower fidelity segment with $n_2 < n_1$ coefficients, the $(n_1 - n_2)$ smallest (alternatively, last) coefficients are dropped.

Image data. Image and video data can also be compressed using DFT, DCT, DWT, etc., and thus one can use the DWT implementation mentioned above. In addition, we have implemented a simple run length encoding algorithm that works well with a sequence of images (e.g., Figure 5). Given two images I_1 and I_2 of resolution $x \times y$, the algorithm computes their similarity score as $\sigma = 1 - \sum_{i=1}^{x} \sum_{j=1}^{y} |I_1(i,j) - I_2(i,j)|/xy$, where $I(i,j)$ denotes the (i,j)'th pixel, normalized within the range $[0,1]$, of image I. For a given threshold ϵ, two images are similar if $\sigma \geq \epsilon$. The algorithm drops an image if it is similar to the last saved image. For example,

for the sequence of images in Figure 5, images (b) and (c) will be dropped for a similarity threshold of 0.9. To dilute a sequence of images, progressively larger values of similarity thresholds are used.

5.2 Handling time-range queries

Given a time range $[t_1, t_2]$, Am-Store can return all compressed data items with timestamps within the time window $[t_1, t_2]$. Note that, within each Am-Store bucket, compressed data are sorted in descending order of timestamps as a linked list. Moreover, adjacent buckets store adjacent parts of the active window. These two properties simplify answering a time-range query. Am-Store first needs to find the most recent data item I_0 with a timestamp $\leq t_2$; Am-Store can then sequentially scan the bucket for as long as it finds items with timestamps $\geq t_1$. If the scan does not finish within the current bucket, it is restarted from the beginning of the next bucket.

Since the window of a bucket (maintained in the bucket header) specifies the range of timestamps of data in the bucket, Am-Store can easily determine which bucket contains the first data point I_0. However, the bucket can be very large, and scanning all the data within it to locate I_0 can be very expensive. Since buckets of the Am-Store are organized as linked lists, one cannot use binary search to efficiently locate I_0. Therefore, Am-Store needs to maintain a suitable data structure to locate I_0 within a bucket.

Skip-list organization. To facilitate efficiently locating I_0, Am-Store organizes sectors within a bucket as a *randomized skip list* [14]. A skip list is an ordered linked list with additional links, added in a randomized way with a geometric/negative binomial distribution, so that a search in the list may quickly skip parts of the list. In terms of efficiency, it is comparable to a binary search tree ($O(\log n)$ average time for most operations). Skip list has been used in many other applications (for flash memory as well [10]), but we construct it in a novel way using timestamped pointers.

Implementing a general skip list, which allows inserting items in the middle of the list, would be expensive in flash; since insertion or deletion in the middle of the list would require updating existing pointers. Fortunately, sectors in a Am-Store bucket are always inserted at the front of the bucket, for either new sensor data collection or due to dilution. Inserting a new node at the front of a skip list can be efficiently implemented on flash as follows. Every node (i.e., sector) in the skip list is assigned with a level $\leq MaxLevel$ such that all sectors have level ≥ 1, and a fraction p (a typical value for p is $\frac{1}{2}$) of the nodes with level $\geq k$ have level $\geq (k+1)$. (See [14] for more details.) The in-memory bucket header and each node maintain $MaxLevel$ *skip pointers* such that the level i skip pointer points to the most recent node with level $\geq i$. All skip pointers are initialized to null. To insert a sector s, we first generate its level l_s. Then, for all $i < l_s$, the level i skip pointer of the bucket header is copied to the node's level i skip pointer. The level l_s skip pointer of the node is set to itself. Finally, for all $i \leq l_s$, the level i skip pointers at in-memory bucket header is set to s. Thus, inserting a sector requires writing to just the bucket header and the first page of a new sector, both of which are in memory. This takes constant time.

During dilution, data is removed from the end of a skip list and added to the skip list of the next bucket. Ideally, this requires changing skip pointers that point to the deleted data to null. However, as mentioned before, modification of pointers in flash is expensive. We address this by timestamping each pointer (in a sector or in header)—with each pointer, we maintain the timestamp of the oldest data in the sector the pointer is pointing to. This allows us to keep a pointer unmodified even when the data pointed by the pointer is moved to a different bucket during dilution—a pointer with a timestamp outside the window of the bucket is implicitly considered as null.

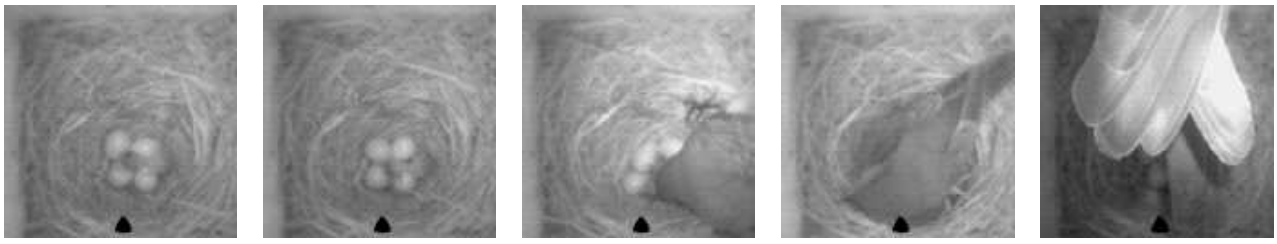
Searching within a time range. Searching for compressed data within a time window uses a combination of skip search and binary search. First, the bucket that contains I_0 is selected by comparing t_2 with the bucket's window. Then skip search is used to locate the sector containing I_0 as follows. Starting from the header of the bucket, Am-Store searches for a sector by traversing the highest level pointers that do not undershoot the sector containing the item with timestamp t_2 (recall that items are sorted in descending order of timestamps). When no more progress can be made at the current level of pointers, or the timestamp associated with the pointer is outside the window of the bucket, the search moves down to the next level. When no more progress at level 1 can be made, the search must be immediately in front of the sector that contains the desired item (if it is in the list). The selected sector is then considered as an array of pages and Am-Store uses binary search to locate the page containing I_0. After locating the page, subsequent pages are read sequentially from the same sector. If the last page of the sector does not contain a timestamp $< t_1$, the read continues from the first page of the next sector of the bucket. If the last data of the current bucket does not contain a timestamp $< t_1$, the scan starts from the first sector of the next bucket. The scan halts as soon as a timestamp $< t_1$ is encountered.

6. IMPLEMENTATION AND EVALUATION

In this section we evaluate FlashLogger with two prototypes. The focus of our experiments is not to evaluate any particular compression algorithm, but to understand FlashLogger's performance in archiving compressed data on flash.

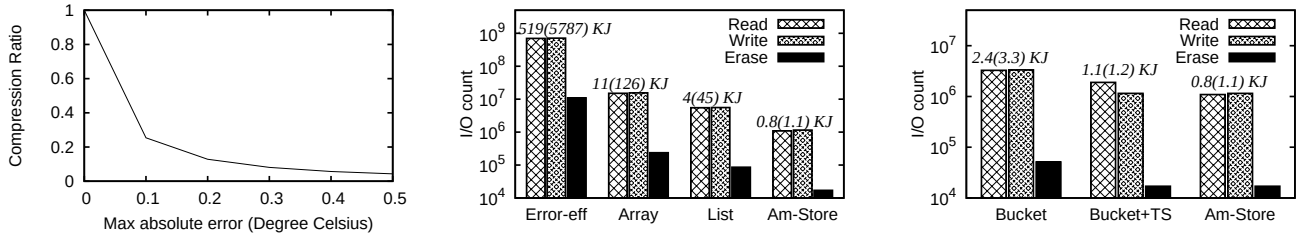
6.1 FlashLogger Implementation

TinyOS implementation. Our first prototype of FlashLogger is implemented on a Moteiv Tmote Sky node running TinyOS 2.1. The prototype is implemented using approximately 500 lines of nesc code and it has around 14KB ROM footprint and 1.5KB RAM footprint. It captures data from a temperature sensor, and archives it in the local flash storage. We use piecewise constant approximation to compress data



(a) Reference image (b) Score = 0.99 (c) Score = 0.91 (d) Score = 0.84 (e) Score = 0.71

Figure 5: 5 images from a Cyclops sensor looking at a bird nest in James Reserve, CA, and their similarity scores with respect to the first image.



(a) Compression ratio (b) Comparison of Am-Store (c) Optimizations in Am-Store

Figure 6: Performance of Am-Store with Temperature data. In (b) and (c), the number at the top of each bar shows the total energy consumption with the flash chip (and with the CF card, inside parentheses).

(Section 5.1).

Our implementation uses TinyOS's LogStorage abstraction as follows. We statically partition the flash chip into volumes, each of which works as an Am-Store bucket with a given fidelity. Within each volume, data is written as a circular buffer supported by the LogStorage abstraction. During dilution of a bucket, data is read from the head of its volume, compressed it with a higher compression error, and written at the tail of the next volume. The data that has been copied from the head of the volume gets overwritten by new data added at the tail of the circular buffer.

Linux implementation. The above implementation does not allow us to experiment with flash packages other than Tmote's on-chip 1MB flash, which is very small for evaluating many features of FlashLogger, especially for audio and image data. We address this limitation in our second and the default prototype for this evaluation. It is written in C and has been compiled in several processors including MSP430 and ARM. However, for our evaluation, we have used it in an Intel P4 1.7 GHz PC running Linux, as it gave us the flexibility to use several existing compression libraries (e.g., for DWT) as well as several compact flash (CF) cards and flash chips. For measuring the energy consumed by a flash card, we connect a Lexar 2GB CF card, through a CF Extend 180 Extender Card[1] and measure voltage and current with a DAQ card.[2] For accessing raw flash chip from the PC, we connect a FujiFilm 1GB XD card through an Olympus Camadia MAUSB-10 card reader. The XD card contains a raw flash chip and, unlike other flash packages such as CF or SD cards, gives access to the physical flash chip without any FTL. For measuring power consumed by flash I/Os, we connect a DAQ card to the internal circuit of the card reader. We measure the power drawn only by flash memory; the consumed energy should be roughly similar had the card/chip connected to other types of processors. We ignore the CPU cost of this prototype, since compression cost is very small compared to I/O cost (Table 1).

6.2 Experimental setup

We use the following datasets and compression algorithms:

Temperature. This is a stream of temperature data collected from 35 sensors deployed in a data center. The temperature varies by up to 25 degree Celsius at different times of the day. The entire dataset contains around 70 million points (total 1GB). For this data, FlashLogger's Compression Module uses the optimal piecewise constant compression scheme, as described in Section 5.1.

Audio. This is an audio signal, sampled at 100Hz, capturing several birds occasionally chirping in a forest. The original audio signal is 30 minutes long; we create a larger audio signal (total 200MB) by concatenating the signal multiple times. For this data, FlashLogger's Compression Module uses DWT, as described in Section 5.1.

Image. This is a sequence of images captured by a network of Cyclops camera sensors monitoring bird nests in James Reserve, CA[3]. The images have a resolution of 128×128, and each camera takes one image every 15 minutes approximately. (Figure 5 shows a few example images.) The data set contains 12000 images, total 200MB in size. For this data, FlashLogger's Compression Module uses the compres-

[1] http://www.sycard.com
[2] http://www.measurementcomputing.com
[3] http://lecs.cs.ucla.edu/~cyclops/nestboxes/

sion scheme described in Section 5.1.

We report the performance of Am-Store and other schemes in steady state by starting with an already full flash. If the dataset is too small to fill the flash, we concatenate multiple copies of it.

6.3 FlashLogger benefits

We experimentally compare Am-Store with the following alternative schemes.

Error-efficient. This is the aggressive, error-efficient scheme in [12] (Section 3). The I/O cost of this scheme is the same as its dual space-efficient scheme.

Array. This scheme uses lazy, batched compression as used in FlashLogger and in previous work such as DIMENSIONS. However, to organize data on flash, it considers the flash as an array of blocks. On recompression of data, the array is compacted to allow binary search to locate data with a given timestamp (Section 3.3).

List. This scheme uses lazy compression as well, but organizes compressed flash blocks as a linked list. Re-compressed data is written to new blocks (Section 3.3).

The lazy compression used by Array and List, as well as by Am-Store, uses parameters $F = \{x, 2x, 3x, 4x, 5x, 6x\}, x = 1/21$ and $E = \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. For each scheme, we allocate 200MB in the flash memory. We use the Temperature dataset in this section; Figure 6(a) shows compressibility of the data for different fidelities.

Figure 6(b) shows I/O counts and total energy consumptions (shown at the top of the bars) of different schemes to compress the entire Temperature dataset in steady state. It shows several important points. First, Error-efficient incurs a large number of I/Os, which are needed to ensure that decompression error (or space usage) of the compressed data is optimal at any point of time.

Second, the I/O cost can be largely reduced by being lazy on recompression, as discussed in Section 4.1, and as demonstrated by the last three schemes in Figure 6(b). However, Array still incurs a large number of I/Os, many of which occur because sometimes unmodified data needs to be moved during compaction to keep data adjacent. Such data movement is avoided in the List organization, which allows data with adjacent timestamps to be scattered into non-adjacent physical locations. Leaving the unmodified data in place improves the performance of List by 66% than Array.

Third, List still incurs a large number of I/Os, many of which come from expensive pointer updates. Am-Store avoids this by using several optimizations. As shown in Figure 6(b), Am-Store improves performance by $\approx 80\%$ over List. Overall, Am-Store is $> 5\times$, $> 15\times$, and $> 650\times$ more energy efficient than List, Array, and Error-efficient, respectively.

Finally, Figure 6(b) shows an additional, yet subtle, advantage of Am-Store when used with a CF card. For Error-efficient, Array, and List, total energy consumption for CF card is an order of magnitude higher than that for flash chip; while the difference for these two flash media is very small with Am-Store. This is due to the fact that most writes in Am-Store are sequential (in contrast, other schemes use many random writes), and sequential writes are at least an order of magnitude cheaper than random writes in existing CF cards.

In summary, a lazy compression scheme, as used in FlashLogger, can provide significant performance benefit over existing error-efficient algorithms. However, simply using a lazy compression is not sufficient to minimize I/O costs if the compressed data is physically organized with simplistic data structures such as an Array or a List. Using Am-Store significantly reduces the I/O cost over these structures.

6.4 Effect of Am-Store optimizations

To understand *how different optimization techniques within Am-Store contribute towards its efficiency*, we consider the following stripped down versions of Am-Store. **(1) Am-Store:** Our proposed data structure.**(2) Bucket+TS:** This is Am-Store without skip list. **(3) Bucket:** This is Am-Store without skip list and timestamped pointers.

Figure 6(c) shows performance of these different versions of Am-Store. First, the Bucket structure is a significant ($\approx 40\%$) improvement over the List structure in Figure 6(b). This shows the benefit of maintaining multiple lists or buckets, since recompressed data are added in front of the a bucket and only the in-memory sector pointers need to be updated. Second, removing data from the end of a bucket still requires modifying in-flash pointers to deleted data to null. This is avoided by using timestamped pointers, which allows pointers to deleted nodes to be left unmodified. As shown in the figure, this reduces energy consumption by $\approx 50\%$. In addition to the above optimizations, Am-Store also uses skip list to efficiently locate data within a bucket for re-compression. This further reduces total costs by around 25%.

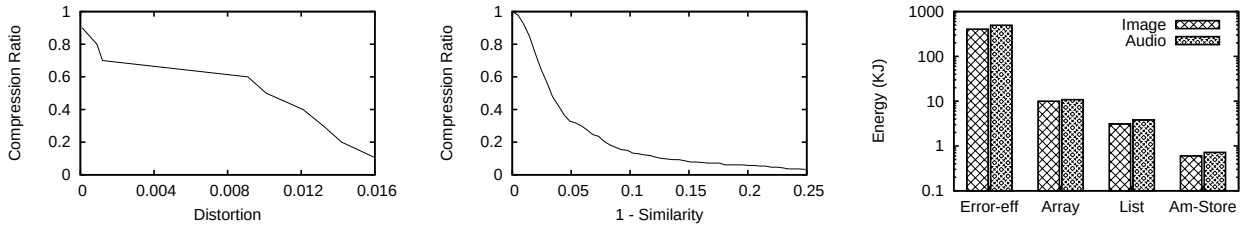
6.5 Am-Store overhead

As mentioned in Section 4, benefits of lazy compression of FlashLogger come with some space overhead. We use the entire $s = 200MB$ space in Am-Store, and measure how much space $s' < s$ Error-efficient takes to compress the same data within the active window of Am-Store. Then, we define $(s - s')/s$ as the space overhead Am-Store.

With our aforementioned experimental setup, the space overhead of Am-Store is $\approx 20\%$. However, this overhead depends on the number of buckets and their associated errors. We tried several other configurations; we found that space overhead can be further reduced ($< 10\%$), but that increases the energy consumption as the cost of dilution increases. Our target devices are constrained more on energy than on flash space; therefore, such a small storage overhead can be easily afforded in many devices (given that very large flash cards can be incorporated to some sensors).

6.6 Using audio and image data

Figures 7(a) and (b) show compressibility of our Audio and Image data for different fidelities (defined in Section 5.1). Figure 7(c) compares Am-Store with other schemes with the



(a) Compression ratio of Audio data (b) Compression ratio of Image data (c) Performance comparison

Figure 7: Performance of Am-Store with Audio and Image data

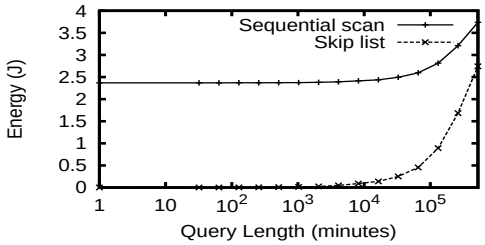


Figure 8: Query cost in Am-Store

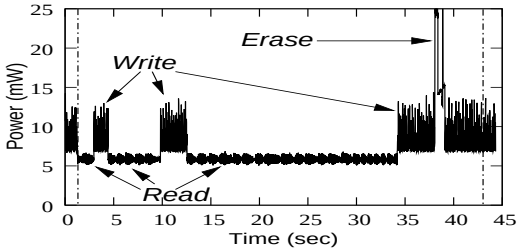


Figure 9: Energy measurement with Tmote Sky

Audio and the Image data, on the flash chip. As before, Am-Store consumes four orders of magnitude less energy that Error-efficient, and around $20\times$ and $8\times$ less energy than Array and List. The performance difference between Temperature data and Audio/Image data comes due to the difference in size of an individual piece of data (each Temperature data point is 8 bytes, each image is 16KB, and each audio segment is 1KB), and different compression ratio.

6.7 Time-range queries in Am-Store

To evaluate querying cost, we first compress and archive Temperature dataset in an Am-Store. We then measure the energy consumed to extract all the records with timestamps within a time window $[t_1, t_1 + length]$, where t_1 is uniformly and randomly distributed within the active window. Figure 8 shows the energy consumed for different values of $length$ in minutes. We consider two alternatives to locate the first data item ($\leq t_1 + length$) in a bucket: sequentially scanning the bucket (as done in a simple linked list) and using skip list. The results show that the query cost increases with the $length$. For smaller query length ($< 10^4$ minutes), using skip lists consumes an order of magnitude less energy than a sequential scan. The benefit comes from a small number of page reads required to locate the first record in the bucket. However, the benefit diminishes as the query size grows bigger. This is because the cost of locating the first record becomes insignificant compared to the cost of reading all the records in a large query range.

6.8 TinyOS evaluation

We have measured the energy consumed by our TinyOS implementation on a Tmote Sky node. For this evaluation, we compress data from Tmote's temperature sensor and use an Am-Store with 4 buckets with errors 0.2, 0.4, 0.6, and 0.8 degree Celsius. Figure 9 shows part of the timeline of total power consumption of a Tmote node, measured with a DAQ card. The region between two dashed lines shows flash activities of the Am-Store during one dilution phase. Our key findings are the following. First, a block erase, a page read, and a page write operation consumes $15.7mJ$, $0.32mJ$, and $0.78mJ$ respectively. Second, a dilution operation takes around 42 seconds, and consumes $\approx 271mJ$. This may appear expensive; but a dilution process creates space for 1558 new readings on average. Assuming that one sensor reading is logged every minute, Am-Store needs to be diluted once per day. During regular operation, Am-Store consumes only $0.012mJ$ to compress and write an 8 byte record (a sensor reading plus timestamp). Thus, the amortized cost per sensor reading (including logging and dilution) is $0.05mJ$, which is $\approx 700\times$ cheaper than an error-efficient scheme, which consumes $\approx 35mJ$ per sensor reading.[4] Third, the logged data occasionally misses a few temperature readings from the on board sensor. This happens because TinyOS does not support multiple threads and thus it cannot perform logging and sensing concurrently. However, with FlashLogger this occurs only during the dilution period, since its regular logging operation is fast. An error-efficient scheme aggravates the situation as all of its logging operations are slow. This may be an important issue with high frequency sensors (e.g., audio).

7. RELATED WORK

Data compression is a well established research field and it has been used to optimize memory [19] and disk space [9] in many systems. Some recent works have also addressed the issue of compressing data for flash memory [7, 20, 21]. However, the primary goal of all these existing systems is to optimize storage space rather than I/O cost. For example, several of these systems pack multiple compressed sectors together even if they cross page boundary; however this is not

[4]We implemented Error-efficient using TinyOS's BlockStorage abstraction since it allows random reads and writes.

desirable for energy aware designs on NAND flash. Some existing flash specific methods [20] require changes to the flash translation layer (FTL) which is typically implemented in the firmware of a flash card and can only be changed by the flash card manufacturer. In contrast to these existing systems, FlashLogger's primary goal is to reduce I/O energy consumption. Moreover, unlike these systems, FlashLogger has been designed to work with limited battery energy, existing flash memory, and limited SRAM size in sensor systems.

Another important aspect that makes FlashLogger different from existing flash-based compression systems is its support for amnesic compression. In addition to showing usefulness of amnesic compression, existing works have also proposed various amnesic compression algorithms to optimize various objective functions. For example, in DIMENSIONS [5], authors proposed *lazy schemes* (based on DWT, one of the compression algorithms in FlashLogger) to distribute multi-resolution compressed data within the network for efficient iterative, drill-down queries. In [13, 12], authors presented *aggressive schemes* to maintain amnesic compressed streaming data with optimal decompression error or DRAM requirement. Our effort is complimentary to this line of work. We focus on how to efficiently organize compressed data on flash memory in a streaming scenario, and our techniques can be used by existing lazy schemes as well. To the best of our knowledge, our work is the first to show how amnesic compression can be efficiently implemented in flash.

Several recent studies have proposed efficient data structures and algorithms for flash storage, including flash-optimized file system [3], B-trees [11], stacks [8], queues [8], online random samples [10], etc. For energy-efficiency, these algorithms, like ours, seek to avoid in-place updates and random writes, but none of them studied our compression and indexing problem. Moreover, existing log-structured indices (e.g., FlashDB [11]) would incur large memory footprint for time-based indexing (See Section 2.1), and hence are not suitable for mote-class devices.

8. CONCLUSION

We have showed that existing aggressive amnesic compression schemes, although optimize for space or decompression error, are not suitable for energy-constrained devices. We also showed that existing lazy schemes can incur high I/O costs if compressed data is organized on flash with suboptimal data structures. We presented FlashLogger, an energy and latency efficient sensor data logging system that uses lazy amnesic compression of sensor data in a flash-efficient manner. All our methods are designed for the limited memory and processing capabilities typical of low power sensor nodes, and are prototyped on TinyOS. Evaluation of FlashLogger with several real world data sets showed orders of magnitude energy savings for both logging data and retrieving data within a time range.

Acknowledgements. The author thanks Aman Kansal, Dimitrios Lymberopoulos, Bodhi Priyantha, and Feng Zhao for helpful discussions. The author also thanks the anonymous reviewers and Phil Levis, the shepherd, for their detailed comments on how to improve the paper.

9. REFERENCES

[1] BURAGOHAIN, C., SHRIVASTAVA, N., AND SURI, S. Space efficient streaming algorithms for the maximum error histogram. In *ICDE* (2007).

[2] CORMODE, G., TIRTHAPURA, S., AND XU, B. Time-decaying sketches for sensor data aggregation. In *ACM PODC* (2007).

[3] DAI, H., NEUFELD, M., AND HAN, R. ELF: an efficient log-structured flash file system for micro sensor nodes. In *ACM SenSys* (2004).

[4] FATLAND, R. Seamonster project page. http://www.robfatland.net/seamonster/, 2008.

[5] GANESAN, D., GREENSTEIN, B., PERELYUBSKIY, D., ESTRIN, D., AND HEIDEMANN, J. An evaluation of multi-resolution storage for sensor networks. In *ACM SenSys* (2003).

[6] HOGG, R., RANKIN, A., MCHENRY, M., HELMICK, D., BERGH, C., ROUMELIOTIS, S., AND MATTHIES, L. Sensors and algorithms for small robot leader/follower behavior. In *SPIE AeroSense Symposium* (2001).

[7] HUANG, W.-T., CHEN, C.-T., CHEN, Y.-S., AND CHEN, C.-H. A compression layer for NAND type flash memory systems. In *Proceedings of the Third International Conference on Information Technology and Applications* (2005).

[8] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: an energy-optimized object storage system for memory-constrained devices. In *ACM SenSys* (2006).

[9] MICROSOFT CORPORATION. What is DoubleSpace and how does it work. MS DOS 6 Technical Reference.

[10] NATH, S., AND GIBBONS, P. B. Online maintenance of very large random samples on flash storage. In *VLDB* (2008).

[11] NATH, S., AND KANSAL, A. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN* (2007).

[12] PALPANAS, T., VLACHOS, M., KEOGH, E., AND GUNOPULOS, D. Streaming time series summarization using user-defined amnesic functions. *IEEE Trans. on Knowl. and Data Eng. 20*, 7 (2008).

[13] PALPANAS, T., VLACHOS, M., KEOGH, E., GUNOPULOS, D., AND TRUPPEL, W. Online amnesic approximation of streaming time series. In *ICDE* (2004).

[14] PUGH, W. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM 33*, 6 (1990).

[15] REINHOLD, L. M. QuickLZ. http://www.quicklz.com/.

[16] SADLER, C. M., AND MARTONOSI, M. Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks. In *ACM SenSys* (2006).

[17] SOROUSH, E., WU, K., AND PEI, J. Fast and quality-guaranteed data streaming in resource-constrained sensor networks. In *ACM MobiHoc* (2008).

[18] STEERE, D. C., BAPTISTA, A., MCNAMEE, D., PU, C., AND WALPOLE, J. Research challenges in environmental observation and forecasting systems. In *MobiCom* (2000).

[19] YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. CRAMES: compressed RAM for embedded systems. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (2005).

[20] YIM, K. S., BAHN, H., , AND KOH, K. A flash compression layer for smartmedia card systems. *IEEE Transactions on Consumer Electronics 50*, 1 (2004), 192–197.

[21] YIM, K. S., KOH, K., AND BAHN, H. A compressed page management scheme for nand-type flash memory. In *Proceedings of the Third International Conference on Information Technology and Applications* (2005).