Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices

Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, Prashant Shenoy
{gmathur, pjd, dganesan, shenoy}@cs.umass.edu
Department of Computer Science
University of Massachusetts, Amherst, MA 01003

Abstract

Recent gains in energy-efficiency of new-generation NAND flash storage have strengthened the case for in-network storage by data-centric sensor network applications. This paper argues that a simple file system abstraction is inadequate for realizing the full benefits of high-capacity low-power NAND flash storage in data-centric applications. Instead we advocate a rich object storage abstraction to support flexible use of the storage system for a variety of application needs and one that is specifically optimized for memory and energy-constrained sensor platforms. We propose *Capsule*, an energy-optimized log-structured object storage system for flash memories that enables sensor applications to exploit storage resources in a multitude of ways. Capsule employs a hardware abstraction layer that hides the vagaries of flash memories for the application and supports energy-optimized implementations of commonly used storage objects such as streams, files, arrays, queues and lists. Further, Capsule supports checkpointing and rollback of object states to tolerate software faults in sensor applications running on inexpensive, unreliable hardware. Our experiments demonstrate that Capsule provides platform-independence, greater functionality, more tunability, and greater energy-efficiency than existing sensor storage solutions, while operating even within the memory constraints of the Mica2 Mote. Our experiments not only demonstrate the energy and memory-efficiency of I/O operations in Capsule but also shows that Capsule consumes less than 15% of the total energy cost in a typical sensor application.

Categories and Subject Descriptors

D.4.2 [**Software**]: Operating Systems Storage Management Secondary storage; D.2.13 [**Software**]: Software Engineering Reusable Software Reusable libraries

General Terms

design, performance, experimentation

Keywords

storage system, flash memory, energy efficiency, objects, embedded systems, sensor network, file system

SenSys'06, November 1–3, 2006, Boulder, Colorado, USA.

1 Introduction

Storage is an essential ingredient of any data-centric sensor network application. Common uses of storage in sensor applications include archival storage [9], temporary data storage [6], storage of sensor calibration tables [10], in-network indexing [20], in-network querying [19] and code storage for network reprogramming [7], among others. Until recently, sensor applications and systems were designed under the assumption that computation is significantly cheaper that both communication and storage, with the latter two incurring roughly equal costs. However, the emergence of a new generation of NAND flash storage has significantly altered this trade-off, with a recent study showing that flash storage is now *two orders of magnitude cheaper* than communication and comparable in cost to computation [11]. This observation challenges conventional wisdom and argues for redesigning systems and applications to exploit local storage and computation whenever possible in order to reduce expensive communication.

While the case for in-network storage has strengthened with the emergence of high-capacity energy-efficient NAND flash memories, existing storage systems built for flash devices (see Table 1) have a number of drawbacks:

Mismatch between storage abstraction and application needs: Many flash-based storage systems, such as YAFFS, YAFFS2 [25], Matchbox [5] and ELF [3], provide a file system abstraction to the application. While a file-based abstraction is useful in many scenarios, it is a poor fit for handling the varied needs of sensor applications. For instance, a common use of local storage is to store a time series of sensor observations and maintain a index on these readings to support queries. A data stream abstraction—an append-only store of time series data—and hash-based index structures, as proposed in MicroHash [27] are better suited for these needs. However, even supporting only this abstraction is restrictive, since it does not enable flash to be used for other purposes such as for "live" application data storage, calibration tables, packet queues for radio transmission, hibernation, etc. Instead, we argue that the storage substrate should support a "rich" object storage abstraction with the ability to create, store and retrieve data objects of various types such as files, streams, lists, arrays, queues, etc. This will enable sensor applications to exploit flash storage in a multitude of ways.

Supporting Use as a Backing Store: Current flash-based storage systems use flash as a persistent data storage

Table 1. Comparison of Capsule to related efforts.

	Storage Devices	Energy Optimized	Memory Optimized	Wear Leveling	Checkpointing	Abstraction	Usage Models
Matchbox	NOR	No	Yes	No	No	Filesystem	File storage; Calibration Tables
MicroHash	MMC	Yes	No	Yes	No	Stream/Index	Stream Storage and Indexing
ELF	NOR	No	Yes	Yes	No	Filesystem	Same as Matchbox
YAFFS	NAND	No	No	Yes	No	Filesystem	Portable devices
Capsule	NAND, NOR	Yes	Yes	Yes	Yes	Object	Data Storage and Indexing; Packet Queues; Temporary Arrays

medium. However, memory is often a scarce commodity on small sensor platforms, with Telos and Mica motes containing 10KB and 4KB of RAM, respectively. With empirical studies showing the energy cost of accessing flash approaching that of RAM, it is now possible for applications to use higher-capacity flash for storing live application data and manipulating it in an energy efficient manner. For instance, tasks can exploit flash as a form of backing store to store large data structures, intermediate results for data processing tasks, etc. It is also feasible to implement local data processing algorithms that manipulate data sets larger than the size of RAM. Such use of flash as a backing store also argues for supporting a richer storage abstraction than the traditional file abstraction.

Optimizing for Energy and Memory Constraints: Energy efficiency and the small amount of available memory are key constraints of tetherless sensor platforms – consequently, the storage subsystem for sensor platforms must optimize both constraints. In contrast, traditionally storage systems have been optimized for bandwidth and access latency. Even in energy-aware file systems such as BlueFS [14], the target energy efficiency is far less than the requirement of long-lived sensor platforms. Among existing approaches designed specifically for sensor devices, only MicroHash [27] makes claims about energy-efficiency. However, MicroHash is designed specifically for stream storage and indexing, rather than as a general-purpose storage system. Further, none of these systems is explicitly designed for platforms with scarce memory. In fact, it is currently difficult to use existing NAND-flash based file systems such as YAFFS [25] on memory-constrained sensor platforms due to their large RAM foot-print.

1.1 Research Contributions

In this paper we propose *Capsule*, an energy-efficient flash-based storage substrate for sensor platforms that overcomes the above drawbacks. The design and implementation of Capsule has led to the following contributions:

Object-based abstraction: Capsule provides the abstraction of typed storage objects to applications; supported object types include streams, indexes, stacks and queues. A novel aspect of Capsule is that it allows composition of objects—for instance, a stream and index object can be composed to construct a sensor database, while a file object can be composed using buffers and a multi-level index object. In addition to allowing reads and writes, objects expose a data structure-like interface, allowing applications to easily manipulate them. Capsule also includes a flash abstraction layer that uses a log-structured design to hide the low-level details of flash hardware from applications. Storing objects on flash enables flexible use of storage resources, for instance, data-centric indexing using indices, temporary buffers using arrays, buffering of outgoing network packets using queues and storing time-series sensor observation using streams. Furthermore, the supported objects can also be used by applications to store live data and use flash as an extension of RAM.

Energy-efficient and memory-efficient design: While traditional storage systems are optimized for throughput and latency, Capsule is explicitly designed for energy- and memory-constrained platforms. Capsule achieves a combination of very high energy-efficiency and a low memory footprint using three techniques: (a) a log-structured design along with write caching for efficiency, (b) optimizing the organization of storage objects to the type of access methods, and (c) efficient memory compaction techniques for objects. While its log-structured design makes Capsule easy to support on virtually any storage media, this paper focuses on exploiting the energy efficiency of NAND flash memories.

Support for Compaction: A unique aspect of Capsule is its support for compaction of data when storage resources are limited in comparison with the storage needs of an application. Each object in Capsule supports a compaction procedure that moves data to reclaim space in flash.

Handling Failures using Checkpointing: Sensor devices are notoriously prone to failures due to software bugs, system crashes, as well as hardware faults due to harsh deployment conditions. Capsule simplifies failure recovery in sensor applications by supporting checkpoints and rollback—it provides energy-efficient support for checkpointing the state of storage objects and the ability to rollback to a previous checkpoint in case of a software fault or a crash.

Implementation and Evaluation: We have augmented the Mica2 Motes with a custom-built board that allows us to experiment with NAND flash memories. We have implemented Capsule in TinyOS running on the Mica2 platform with an option to use either the NOR flash memory on the Mica2 or our custom NAND flash board. We perform a detailed experimental evaluation of Capsule to demonstrate its energy-efficiency. For instance, writing 64 bytes of data to a Stream object consumes 0.028mJ of energy while reading the same data consumes 0.043mJ. The compaction of a Stream holding 128KB of data consumes 48.9mJ energy and takes 3.2 seconds. In comparison, transmitting a 64 byte packet using the Mica2 CC1000 radio at 1% duty cycling consumes 40mJ and takes 1.1 seconds. We also show that in a representative application involving regular light sensing, a large component of archival storage and periodic summary transmission, Capsule consumes less than 15% of the total energy cost. We compared our file system implementation against Matchbox and concluded that Capsule provides useful additional fea-

tures with a better overall energy profile and performance than Matchbox.

The rest of this paper first provides an overview of flash hardware in Section 2, followed by the design and implementation of Capsule in Sections 3-6. We present our evaluation in Section 7, followed by related work and conclusions in Sections 8 and 9.

2 Flash Memory Characteristics

We present a brief overview of NAND flash hardware, focusing on the constraints that it imposes on storage system design and their read and write characteristics.

2.1 Flash Memory Restrictions

Flash chips have emerged as the storage technology of choice for numerous consumer devices as well as sensor platforms. Their low energy consumption, ultra-low idle current, and high capacity make them an attractive choice for long-lived sensor applications.

A key constraint of flash devices is that writes are one-time – once written, a memory location must be reset or *erased* before it may be written again. Erase operations are relatively slow and expensive and must be performed in granularity of an erase block. While the erase block is same as a page in some devices, it can span multiple pages in newer devices, thereby complicating data management.

While NAND memories impose stringent constraints, they are also the most energy-efficient storage solution for sensor devices[11]. The number of non-overlapping writes allowed between erases to each page on the device are limited, and often between 1 and 4. Larger page size NAND memory devices often also require writes within an erase block to be sequential. The amount of RAM available on most sensor devices may make working with these devices difficult. Popular sensor devices have RAM sizes ranging from 4KB on the Mica2 platform [26] through 10K on the TelosB [17]. In contrast to this, the size of available flash devices (upto 2GB) is five orders of magnitude larger illustrating the disparity between primary and secondary storage.

2.2 NAND Flash Read/Write Characteristics

Since we are interested in designing an energy-efficient storage system, in this section we describe a simple model that captures the energy cost of the read and write operations on a NAND flash. While the measurements presented here are specific to the NAND flash, the model is applicable to both NOR and NAND flash memories.

Table 2 shows both the device and system-level energy and latency costs involved with the read and write operations of a Toshiba TC58DVG02A1FT00 1Gb (128 MB) NAND flash[22] board attached to a Mica2 mote using an add-on board fabricated by us (discussed in detail in Section 6). We find that the energy cost of both the storage subsystem and the entire mote to be a linear function of the number of bytes read from flash. Much like the seek overhead in magnetic disks, there is a fixed cost of accessing a page, and then a per-byte overhead associated with each additional byte written to (or read from) the page. Unlike disks though, accessing adjacent pages does not impact the fixed cost. The fixed cost corresponds to the time during which an address is clocked in and the flash read or write operation is enabled. Once

		Write	Read
NAND Flash Energy Cost	Fixed cost	13.2μJ	1.073μJ
	Cost per-byte	0.0202μJ	0.0322μJ
NAND Flash Latency	Fixed cost	238us	32us
	Cost per-byte	1.530us	1.761us
NAND Flash + CPU Energy Cost	Fixed cost	24.54μJ	4.07μJ
	Cost per-byte	0.0962μJ	0.105μJ
NAND Flash + CPU Latency	Fixed cost	274us	69us
	Cost per-byte	1.577us	1.759us

Table 2. Cost of flash operations

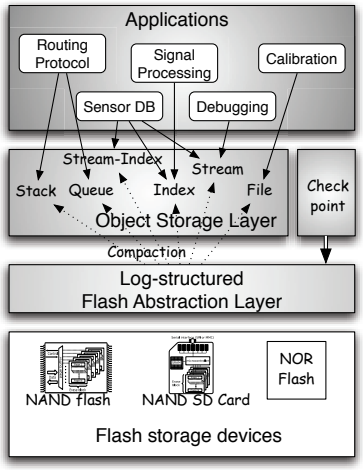

Figure 1. Object Storage architecture

enabled, the cost of clocking data in/out of the flash chip is linearly dependent on the size of data being operated upon. Note that the cost of reading or writing n pages is n times the cost of reading or writing a single page since each page is addressed separately.

Based on our measurements, we find that the energy cost of writing ($W(d)$) and reading ($R(d)$) d bytes of data to and from flash respectively are:

$$W(d) = 24.54 + d \cdot 0.0962\mu J \quad (1)$$

$$R(d) = 4.07 + d \cdot 0.105\mu J \quad (2)$$

The above numbers have significant implications on the design of the storage system. Of particular importance is the observation that the fixed energy cost is 13 times greater for writes than reads, whereas the cost per additional byte is almost the same for both writes and reads. These results are at the core of the read and write caching techniques that we will use in Capsule.

3 Object Storage Architecture

Capsule employs a three layer architecture consisting of a flash abstraction layer (FAL), an object layer, and an application layer (see Figure 1). The FAL hides the low-level flash hardware details from the rest of the object store. This layer comprises a low-level device driver that interfaces with the hardware. The FAL addresses important design decisions that highlight the energy and memory-optimized nature of Capsule. To deal with the memory limitations of sensor platforms, the FAL offers a buffered log-based design, *i.e.* objects chunks are written in an interleaved append-only manner. The FAL also permits direct access to flash storage via

raw reads and writes to support checkpointing. The FAL is responsible for performing error detection and correction for flash memories. It supports storage space reclamation where previously written blocks are cleaned to remove invalid object data and create space for new ones. The cleaner in the FAL layer triggers compaction on objects in the system when the flash memory usage exceeds a certain threshold. Section 4 discusses the FAL in more detail.

The object layer resides above the FAL. This layer provides native and flash-optimized implementation of basic objects such as streams, queues, stack and static indices, and composite objects such as stream-index and file. Each of these structures is a named, persistent object in the storage layer. Applications or higher layers of the stack can transparently create, access, and manipulate any supported object without dealing with the underlying storage device. In this paper, we describe techniques that can be used to optimize energy usage of objects based on the access patterns of applications. Each object in Capsule supports efficient compaction methods that are invoked when the FAL triggers a cleaning task. Finally, checkpointing and rollback are supported to enable recovery from software faults or crashes.

Applications can use one or more of the objects in Capsule with simple and intuitive interfaces. We discuss and evaluate three uses of our system— archival storage, indexing and querying of stored sensor data and batching packets in flash to improve communication efficiency.

4 The Flash Abstraction Layer (FAL)

The purpose of the FAL is to hide the vagaries of the flash device and present a simple storage interface to the upper layers of the storage stack (Figure 1), while optimizing energy efficiency and memory use.

We now discuss the log-structured FAL design, followed by the storage reclamation support that the FAL provides. We briefly discuss error handling for flash hardware errors and special block allocation mechanisms for applications that wish to bypass the FAL and need direct access to flash.

4.1 Log-structured Design

The combination of memory and energy constraints of sensors, in addition to flash technology constraints, places several restrictions on the design of a storage system. Existing NAND flash-based storage systems for portable devices have a large memory footprint and are not energy efficient. On the memory front, some systems such as JFFS [24] maintain an in-memory logical-to-physical block map to simplify erase management, while others such as YAFFS [25] maintain an in-memory map of the file blocks – neither techniques use memory efficiently. From an energy perspective, these systems allocate the flash in granularity of one or more pages to each file, and data modification requires a re-write of the allocated page(s) to a new location on flash, resulting in high energy consumption. Both these reasons make these systems infeasible for sensor platforms. A log-structured organization of the FAL overcomes the memory inefficiency of existing systems as it requires the maintenance of minimal in-memory state. Additionally, our design permits fine-grained selection of the data sizes by the application, which enables better application-specific energy optimization.

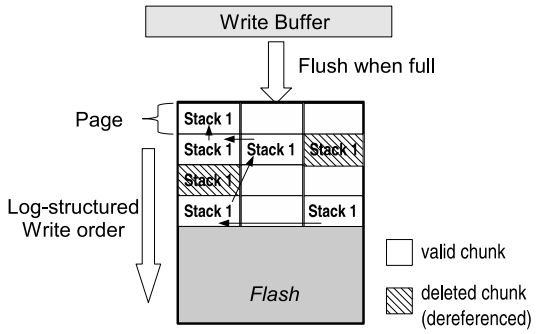

Figure 2. Design of the log-structured FAL.

Log structured file systems were first popularized by the work of Rosenblum and Osterhout (LFS [21]), and have since been used in a number of other storage systems. Figure 2 shows the log-structured organization of the FAL, which treats the storage device as a "log" – it sequentially traverses the device from start to the end writing data to pages. Once data has been written to a segment of the storage device it cannot be modified, only erased.

A log-structured design provides a number of benefits to our system. First, it avoids energy- and memory-intensive block reads and writes, while still working within the hardware constraints imposed by NAND flash memories. Second, since the constraints of NAND flashes are a super-set of the constraints of NOR flashes, a log-structured design can be easily ported to both these types of flash memories with minimal modification to the FAL. This enables Capsule to be supported on existing sensor platforms such as Motes that use NOR flash memories as well as on NAND-flash based sensor platforms. Third, a log-structured design enables data from different objects to be batched together before writing to flash, thereby enabling better write energy optimization. Our design results in chunks[1] being stored in a non-contiguous manner; writes incur greater fixed costs than reads (as shown in Equation 1 and 2), making write optimization more critical. Finally, we expect the dominant use of storage on sensor devices to be for writing sensor data, which further validates the choice of a write-optimized design. We now describe how writes, and reads are handled by the log-structured FAL.

Data Write: Figure 2 shows the write operation of the FAL. Successive writes are appended to a write buffer, which is flushed to flash when it becomes full. The size of the write buffer at the FAL is determined by the NAND flash constraints and the memory constraints of the device. Thus, if a NAND flash with page size p has the limitation that each page can be written to only k times, the buffer size should be at least $\frac{p}{k}$. Larger write buffer sizes are beneficial since writes have high fixed cost as shown in Equation 1 – thus, we advocate that *the FAL write buffer should be as large as possible* given the memory constraint of the node.

In our implementation, the FAL adds a small 3 byte

[1] A *chunk* denotes the data written by the FAL, which consists of object data along with some metadata added by the FAL.

header (2 byte length, and one byte checksum) to each object chunk that is passed down from the object layer before storing in the write buffer. Typically the buffer is flushed to flash when full, but it can also be force-flushed by the application. After the buffer has been written to flash, the FAL moves the current write pointer forward to the next write location.

Data Read: Upon receiving a read request, the FAL reads the bytes corresponding to the requested object chunk from flash. Our custom driver for the NAND flash allows reading data using an efficient single-phase read mechanism. As the header bytes are being clocked in from flash, the FAL processes the header to determine the length of the object data in the chunk. The read operation is terminated after the header and data bytes have been read.

The FAL does not use read buffering, and read operations are performed when requested. While batching reads is cheaper than individual ones, we see from Equation 2 that the constant cost of performing a read operation is low, hence the energy savings from batching reads is not significant. In addition, a particular object's chunks are likely to be interleaved with other objects as shown in Figure 2 due to the log-structured design, making batching difficult.

4.2 Memory Reclamation

Since flash memory is limited, the FAL handles storage space exhaustion by deleting data. One option is to delete the earliest data written to flash and let the storage objects handle any failures arising due to non-existent data when trying to follow valid pointers referring to this section of the flash. If all the data in the section of flash to be deleted is invalid and unreferenced, then no issues will arise from erasing this part of the flash. However, this approach is unsatisfactory from the perspective of storage objects storing *high priority* data such as long-term summaries of the sensor data, history of detected events or calibration tables.

Log-structured file systems employ a *cleaner* task [12, 21] to reclaim memory, and a multitude of techniques have been proposed for performing cleaning. Cleaning techniques proposed for disks such as hole-plugging [23] are not feasible on flash memories that lack in-place modification capability. Cleaning in Capsule is based on three design principles. Firstly, the FAL does not actually perform the cleaning but only keeps track of *when* the cleaning is required. At this point the FAL will inform each storage object, and each is responsible for its own cleaning. Secondly, we adopt a design that pushes the time at which cleaning is performed to be as late as possible. Cleaning involves data compaction, which involves reading the current valid data and re-writing it to the current write frontier, freeing the earlier blocks for erasing. All these operations are expensive energy-wise and should be performed only when absolutely necessary. Thirdly, we choose simple cleaning policies rather than sophisticated ones that have been considered in log-structured file systems due to the resource-constrained and memory-limited nature of sensor devices.

The *cleaner* is a part of the FAL, and exposes a compaction interface to which objects holding high priority data are wired at compile time. The FAL continually tracks the percentage of flash that has been filled and uses this measure to decide when to trigger the cleaner. Once the certain pre-determined threshold (default is half the size of flash) is exceeded, the Cleaner sends a compaction event to the objects wired to the compaction interface.

Each Capsule object implements the *compaction* interface (see Appendix A), supporting a `compact` procedure that traverses and reads all valid data (Figure 2) and re-writes it to the current write frontier on flash. Once all the wired objects have performed compaction, the FAL marks the older blocks for deletion. These blocks are erased only when the rest of the flash fills – this retains even the low-priority data as long as possible.

4.3 Error Handling

Flash memory is prone to single-bit errors, and the probability of errors occurring on NAND flash is substantially higher than NOR. The FAL provides support for a simple checksum for each chunk. This checksum enables the FAL and higher layers to check for errors when reading a chunk. In addition, the FAL uses a single-error-correction double-error-detection (SECDED) algorithm to generate codes at the page level. If the chunk checksum cannot be verified, the entire page is read into memory, and the error correction operation can be performed using the SECDED code. In extremely memory limited sensor platforms or on NOR flashes, error correction can be turned off since it necessitates allocating an extra page-sized read buffer and our experience has shown these errors to be rare in practice. Our current implementation supports the chunk-level checksums, and including support for the the page-level error correction is part of our future plans.

4.4 Block Allocation

The FAL also offers a raw read and write interface that bypasses the log-structured component and accesses the flash directly. The FAL designates a part of the flash (a static group of erase blocks) to special objects or applications that want direct access to flash. Such a facility is necessary for *root directory* management performed by the Checkpoint component (discussed in Section 5.4), which is used for storing critical data immediately that needs to be easily located and should remain persistent across system failures. Another component that needs direct flash access is network re-programming (*e.g.* Deluge [7]) – to re-program a Mote, the network re-programming module needs to store new code contiguously in flash, without the FAL headers.

5 Object Storage Layer

The object layer resides above the FAL and supports efficient creation and manipulation of a variety of storage objects (see Figure 1). In this section, we discuss key concepts behind the design of Capsule objects. First, we provide a taxonomy of sensor applications and the type of objects that would be useful to them – this guides the choice of objects that we support in our system. Second, we describe the four basic storage objects we support: *stack*, *queue*, *stream* and a *static-index*, which form the first-order objects in our system. For each object, we describe how they are created, the access methods that they support and how they can be compacted efficiently. Third, we describe how these basic objects can be composed into a number of useful composite objects

Application	*Data Type*	*Storage object*
Archival Storage	Raw Sensor Data	Stream
Archival Storage and Querying	Index	Stream-Index
Signal Processing or Aggregation	Temporary array	Index
Network Routing	Packet Buffer	Queue/Stack
Debugging logs	Time-series logs	Stream-Index
Calibration	Tables	File

Table 3. Taxonomy of applications and storage objects

such as a *file* and an *stream-index* object. Finally, we describe how checkpointing and rollback is supported in our system to recover from failures.

5.1 Taxonomy of Storage Objects

In this section, we survey common uses of storage in sensor applications that inform the choice of objects supported by Capsule (shown in Table 3).

A number of data-centric applications and research efforts need the capability to perform in-network archival storage, indexing and querying of stored data. Sensor data is typically stored as time series streams that are indexed by timestamp [9], value [27], or event [20]. Such applications need to efficiently store *data streams* and maintain *indices*.

A second class of applications that can take advantage of efficient storage are those that need to use flash memory as a backing store to perform memory-intensive computation. Many data-rich sensing applications such as vehicle monitoring, acoustic sensing, or seismic sensing need to use large *arrays* to perform sophisticated signal processing operations such as FFT, wavelet transforms, etc.

A number of system components can also benefit from efficient storage. The radio stack in TinyOS [8] does not currently support packet queuing due to memory limitations. *Queue* objects could be used to buffer packets on flash in an energy-efficient manner. Debugging distributed sensors is often a necessary aspect of sensor network deployments, and requires efficient methods for storing debugging *logs* [18] and retrieving these logs. Other uses of the object store include support for persistent storage of calibration *tables* corresponding to different sensors on the node. Finally, there is a need to support a *file system* abstraction to easily migrate applications that have already been built using existing sensor file systems such as Matchbox.

Based on the taxonomy of flash memory needs of applications (Table 3), we identify a core set of basic objects (Stack, Queue, Stream, and Index) that are the first-order objects in Capsule, and a set of composite objects (Stream-Index, File) that are composed from multiple basic objects.

5.2 Basic Storage Objects

In this section, we describe the design of basic objects in Capsule and provide a cost analysis of the access methods that they support (summarized in Table 4). We also describe the compaction methods that each object provides that can be invoked.

5.2.1 Stack Object

The stack object represent the simplest of the storage objects. The push operation of the stack accepts data passed by the application, appending an object-level header to the data. The stack header includes a pointer to the location of the previously stored element on the flash. The stack object

Object Name	Operation	Energy Cost
Stack	Push	1 chunk write
	Pop	1 chunk read
	Compaction	N header reads, chunk reads and chunk writes + $\frac{N}{k}$ chunk reads and writes
Queue	Enqueue	1 chunk write
	Dequeue	N-1 chunk header reads + 1 chunk read
	Compaction	Same as Stack
Stream	Append	1 chunk write
	Pointer Seek	0
	Seek	N-1 chunk header reads
	Next Traversal	N-1 chunk header reads + 1 chunk read
	Previous Traversal	1 chunk read
	Compaction	Same as Stack
Index	Set	H chunk write
	Get	H chunk read
	Compaction	$\frac{k^H-1}{k-1}$ chunk reads and writes

Table 4. Analysis of energy consumption of storage objects. Here N = number of elements in the storage object, H is the number of levels in the Index, and k = number of pointers batched in a chunk for compaction or indexing.

then passes the header and the data to FAL, which copies it to its write buffer. The location of the stored element is then returned to the stack object, which updates its in-memory pointer to reference the written element as the last element stored. Popping data from the stack results in reading the element pointed to by the current in-memory pointer, and then updating the pointer to the value of the header of the element. When the stack object has been used and is no longer required, it can be invalidated, and the storage space that it consumes can be reclaimed by the cleaner.

Stack compaction: Compaction of the stack object involves accessing each object from first to last and re-writing them to the head of the log. Since flash memory only allows backward linking, each read will involve traversal through the entire stack until the next inserted element before reading this element and writing it to the head of the log. To optimize the compaction cost, we use a two-step scheme as shown in Figure 3. First, the stack is traversed from head to tail (last inserted to first inserted element) at cost $N \cdot R(h)$ (refer Equations 1 and 2), where N is the number of elements in the stack and h is the total header size, *i.e.* the sum of FAL and stack headers. The pointers for each of these elements are written into a temporary stack of pointers, perhaps after batching k pointers together in each write incurring cost $\frac{N}{k} \cdot W(d)$, where d is the size of a stack chunk. Next, the stack of pointers is traversed (at cost $\frac{N}{k} \cdot R(d)$) and each data chunk corresponding to the pointer is now read and then re-written to the FAL to create the new compacted stack (at cost $N \cdot (R(d) + W(d))$). In the above cost analysis, we assume that the size of the object chunk, d, is the same for both stacks. The total cost is, therefore, $(N + \frac{N}{k}) \cdot (R(d) + W(d)) + N \cdot R(h)$ as shown in Table 4.

5.2.2 Queues

The need to support First-In First-Out (FIFO) semantics makes the implementation of a queue object more complex than the stack since the flash does not allow forward pointing to data. For example, once the first element of the queue has been written, it cannot be modified to point to the sec-

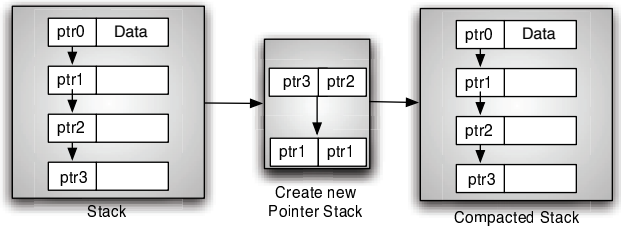

Figure 3. **Compaction of the Stack object**

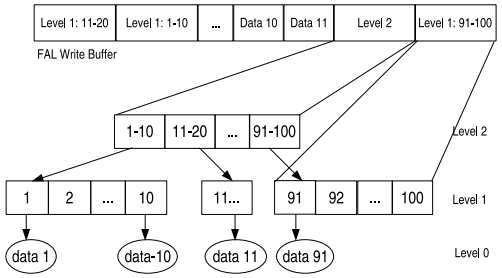

Figure 4. **Organization of a Static Index object on flash.**

ond element of the queue which will be written at a later point in time. Therefore, our queue implementation uses reverse pointers instead of forward pointers, *i.e.* instead of the first element of the queue pointing to the second, the second points to the first, and so on. This makes the queue organization inherently Last-In First-Out (LIFO) or that of a stack. Enqueuing data to this queue does not present a problem since it is similar to pushing data to the stack. However, dequeuing requires a complete traversal of the queue from the tail of the queue to the head to locate the element after the head. Thus, each dequeue operation requires reading N headers before reading the first element in the queue. The compaction procedure for queues is similar to the one that we described for stacks. An alternate queue implementation could use the index object, while still providing a queue interface.

5.2.3 Stream

A stream object is similar to the stack and queue objects but supports a wider set of access methods. Storing data into a stream object involves a backward-pointer chaining like the queue. Streams can be traversed in two ways – either last to first like a stack or first to last like a queue. Due to the log-structured nature of the FAL, a last-to-first access method is considerably more energy efficient. The stream can be *seek*-ed to any point within it and traversed either backward or forward from there, making it easy to navigate through it. Compacting a stream involves exactly the same process as that for compacting a stack, yielding a similar cost.

5.2.4 Static-sized Index and Array

Our index object permits data to be stored using $(key, opaque\ data)$ format, and supports a fixed range of *key* values that is fixed at compilation time. Since this object provides an access pattern similar to that of an array, we use both interchangeably in this paper. Figure 4 shows the structure of the index object – it is hierarchical and the number of levels in the hierarchy is static. We do not support dynamically growing indices since TinyOS does not currently support a dynamic memory management scheme, making it infeasible to dynamically allocate and deallocate buffers for the various levels of the index.

Figure 4 shows the construction of a two-level index. Level zero of the implementation is the actual $opaque\ data$ that has been stored. Level one of the index points to the data, and level 2 of the index aggregates the first level nodes of the index. Each level of the index has a single buffer that all nodes at that level share. For example, the `set` operation on the index looks up and then loads the appropriate first level index corresponding to the *key*. It then writes the *value* to FAL and updates the first level of the index with the location of the written *value*. If the next *set* or *get* operation operates on the same first level index node, then this index gets updated in memory. But if the next operation requires some other first level index node, the current page is first flushed to flash (if it has been modified) and then the second level index is updated similarly, and finally the relevant first level page loaded into memory.

Compaction of the index object involves traversing the object in a depth-first manner, reading the pointers and writing it to FAL. The cost of the compaction operation is therefore the same as the cost of reading the index in a depth-first manner, and writing the entire index into a new location in flash. If the index has H levels, and each index chunk can store k pointers, the total number of index chunks in the tree is $\frac{k^H-1}{k-1}$. Compaction involves reading and writing every chunk in the index, and has cost $\frac{k^H-1}{k-1}(R(d)+W(d))$.

5.3 Composite Storage Objects

The object store permits the construction of composite storage objects from the basic set of objects that we described. The creation and access methods for the composite objects are simple extensions of the primary objects, but compaction is more complex and cannot be achieved by performing compaction on the individual objects. Object composition is currently done "by-hand" in Capsule; making nesting of objects simpler is part of our future plans. We present two composite storage objects. The first composite object that we describe is a *stream-index* object that can be used for indexing a stored stream. For instance, an application can use a stream-index to store sensed data and tag segments of the stored stream where events were detected. Second, we describe our implementation of a file system in Capsule using a *file* composite storage object that emulates the behavior of a regular file. This object facilitates porting applications that have already been developed for the Matchbox filesystem [5] to Capsule.

5.3.1 Stream-Index

The stream-index object encapsulates a stream and an index object and offers a powerful interface (see Appendix A) to the application. The application can directly archive its data to this object by using the `add` method, which saves the data to the stream. When an event is detected in the sensed data, it can be tagged using the `setTag` method, which stores

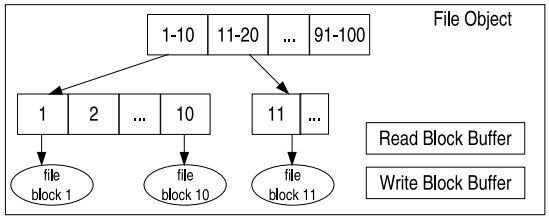

Figure 5. Design of a filesystem using Capsule.

the pointer to the stored stream data in the next free *key* in the index object. This interface can also be trivially modified to tag ranges of sensor readings instead of a single reading. The `seek` method allows the application to seek into the stream based on a given tag, and the `next` and `previous` methods allow the application to traverse data in either direction.

5.3.2 File System

Capsule can be used to construct a simple filesystem with the help of the index object discussed earlier — Figure 5 shows our design and we provide a brief overview of our implementation. Our file system is composed of two objects – the *file* object is used to perform operations on a single file and a singleton *file-system* object that stores the metadata associated with each file. The file system object is responsible for assigning each file a unique *file-id* and mapping the filename to its associated file-id. Each file-id maps to a unique file object that is actually a static index storing the contents of the file in separate fixed sized *file-blocks*, with the index pointing to the location of each file-block on flash. The object also maintains the current length of the file (in bytes) and any working read/write pointers that the file may have. Additionally, it maintains two file-block sized buffers, one used as a read and the other as a write cache. When performing a data write, the data is copied into the write cache that is flushed when filled. Similarly, data is first read into the read cache before being returned to the application. Organizing the file blocks using the index object allows us to support *random access* to each block. We use this feature to modify previously written data. This is achieved by first loading the appropriate file-block from flash, modifying the relevant bytes and then writing the block to a new location and updating the index accordingly. The previous block is now automatically de-referenced.

Our implementation supports both reads and writes simultaneously to the file. Multiple files can be open and operated upon at the same time. Our file system takes advantage of the checkpoint-rollback capability of Capsule to provide consistency guarantees. These features are not supported by ELF and Matchbox, and demonstrate the flexibility of object composition within Capsule. Section 7.3.2 provides a comparison of the performance of Capsule and Matchbox.

5.4 Checkpointing and Rollback

Capsule also supports capability for checkpointing and rollback of objects – checkpointing allows the sensor to capture the state of the storage objects, while rollback allows the sensor to go back to a previously checkpointed state. This not only simplifies data management, it also helps deal with software bugs, hardware glitches, energy depletion, and other myriad faults common in unreliable sensor nodes.

The inability of flash to over-write data once written actually simplifies the implementation of checkpointing. The internal pointers of an object (*e.g.*, the next pointer for a stack or a queue) cannot be modified once they are written to flash. The in-memory state of a storage object (which typically points to its written data on flash) becomes sufficient to provide a consistent *snapshot* of the object at any instant. The cumulative state of all active storage objects provides a snapshot of the system at any given instant. We implement checkpointing support using a special *checkpoint* component (see Appendix A), which exposes two operations – `checkpoint` and `rollback`. The `checkpoint` operation captures the snapshot of the system and stores it to flash. This saved snapshot can be used to revert to a consistent state in the instance of a system failure, or object corruption.

The Capsule storage objects implement a `serialize` interface (see Appendix A). The checkpoint component calls the `checkpoint` method on this interface when it needs to take a snapshot. This provides each Capsule object a shared memory buffer where it stores its in-memory state, which is then written to flash. The checkpoint component uses a few erase blocks in flash as the *root directory* that it manages explicitly, bypassing the FAL (Section 4.4). Once the checkpoint data has been written to flash, a new entry is made to the root directory pointing to the created checkpoint.

In the event of node restart or an explicit rollback call from the application, the root directory is examined to find the most recent checkpoint, which is used to restore system state. CRCs are maintained over the checkpoint data and the root directory entries to prevent corrupt checkpoints (possibly caused by the node crashing while a checkpoint is being created) from being recovered. The root directory entry provides a pointer to the saved checkpoint state. The checkpoint component uses the `rollback` method in the serialize interface to replace the in-memory state of linked objects using the same shared buffer mechanism as `checkpoint`.

6 Implementation

Implementing Capsule[2] presented a number of unique challenges. TinyOS is event-driven and this necessitated writing Capsule as a state machine using the split-phase paradigm. The checkpointing component required careful timing and co-ordination between components for its correct operation. We went through multiple iterations of Capsule design to maximize code and object reuse even within our implementation – *e.g.*, the checkpoint component uses a stack object to store state information, similar to the stream and stack compaction methods that also use a stack to hold temporary data during compaction (Section 5.2). Another major concern was the overall memory foot-print of Capsule. We optimized the Capsule architecture to minimize buffering and maximize code reuse; buffers have only been used at stages where they have a sufficient impact on the energy efficiency. A test application that does not use checkpointing/recovery but uses one instance of each of the following

[2]Capsule source code is available at `http://sensors.cs.umass.edu/projects/capsule/`

objects – index, stream, stream-index, stack and queue, requires only 25.4Kb of ROM and 1.6Kb of RAM. Another application that uses one each of the stack and stream objects along with checkpointing support, had a foot-print of 16.6Kb in ROM and 1.4Kb in RAM. While the Capsule code base is approximately 9000 lines of code, the percentage of the code used by an application depends largely on the number and type of objects instantiated and the precise Capsule features used.

7 Evaluation

In this section, we evaluate the performance of Capsule on the Mica2 platform. While Capsuleworks on the Mica2 and Mica2dot NOR flash as well as our custom NAND flash adapter, the energy efficiency of the NAND flash [11] motivated its use as the primary storage substrate for our experiments. However, the experiment comparing the Capsule and Matchbox file systems is based on the Mica2 NOR flash (Section 7.3.2), demonstrating Capsule's portability.

Our evaluation has four parts – first, we benchmark the performance of FAL including the impact of read and write caching. Second, we perform a thorough evaluation of the performance of different storage objects, including the relative efficiency of their access methods. We measure the impact of access pattern and chunk size on the performance of memory compaction as well as checkpointing. Third, we describe interesting trade-offs that emerge in an application study that combines the different pieces of our system and evaluates system performance as a whole. And finally, we compare the performance of a file system built using Capsule (Section 5.3.2) with Matchbox.

Experimental Setup: We built a NAND flash adapter for the Mica2 using the Toshiba TC58DVG02A1FT00 1Gb (128 MB) flash [22] for our experiments – it has a page size of 512 bytes, an erase block size of 32 pages and permits a maximum of 4 non-overlapping writes within each page. Our measurements involved measuring the current at the sensor and flash device power leads, with the help of a 10Ω sense resistor and a digital oscilloscope. The mote was powered by an external power supply with a supply voltage of 3.3V; energy consumption "in the field" with a partially discharged battery may be somewhat lower.

7.1 FAL Performance

The choices made at the FAL are fundamental to the energy usage of Capsule. We ask two questions in this section: *How much write buffering should be performed at the FAL layer?* and *How much buffering should be performed by a higher layer before writing to FAL?* To answer these, we vary write and read buffer sizes and examine the energy consumption of the write and read flash operations. Figure 6 shows our results, where each point corresponds to the energy consumed by writing or reading one byte of data amortized over a buffer of that particular size.

Impact of FAL Write Buffer Size: For the particular flash that we use, given the page size of 512 bytes and a maximum of 4 writes per page, the minimum write buffer size is 128 bytes (see Section 4). The per-byte write curve in Figure 6 shows that write costs reduce significantly for increasing write buffering. A reduction in buffer size from 512 bytes to 128 bytes saves 9.4% of the available memory on the Mica2, but the per byte write energy consumption increases from 0.144μA to 0.314μA, *i.e.* 118%. Reducing memory consumption from 512 bytes to 256 bytes results in memory savings of 6.3% of the available memory on the Mica2 mote, but at 40% additional energy cost.

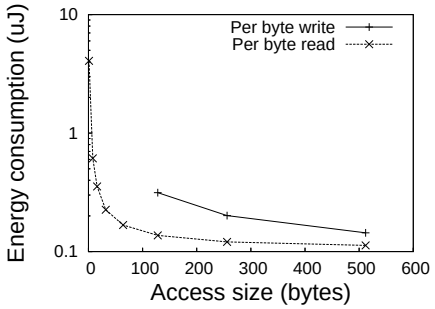

Figure 6. The amortized energy consumption of the read and write operations was measured for different data sizes using the Mica2 and the fabricated Toshiba 128 MB NAND adapter. The figure clearly shows that the write operation has a high fixed cost involved in comparison to the read operation.

Thus, increased write buffering at the FAL has a considerable impact on reducing the energy consumption of flash write operations – consequently, the FAL write buffer should be as large as possible.

Higher Layer Buffer size: In our log-structured design, chunks are passed down from the object layer to FAL and are not guaranteed to be stored contiguously on flash. As a result, reads of consecutive chunks need to be performed one at a time since consecutive object chunks are not necessarily spatially adjacent on flash. To amortize the read cost, data buffering needs to be performed at the object or application layer. We aim to find the appropriate size of the higher layer buffer through this experiment.

Figure 6 shows how the size of the chunk impacts the cost of flash reads. Similar to write costs, the per-byte cost of a read reduces with increasing buffer sizes. The read cost reduces sharply by 72% as the buffer size increases from 8 bytes to 64 bytes. However, beyond 64 bytes the per byte cost decreases more slowly and larger read buffers have relatively less impact. For example, increasing the write buffer from 128 bytes to 512 bytes results in a gain of 0.17μJ, whereas the same increase in read buffer size results in a gain of only 0.024 μJ, *i.e.* only 14% of the write benefit.

Thus, approximately 64 bytes of data buffering at the storage object or application layer is sufficient to obtain good energy efficiency for flash read operations.

7.2 Performance of Basic Storage Objects

In this section, we first evaluate the energy efficiency of each access method supported by the core Capsule objects. Then, we present some important trade-offs that arise in the choice of chunk size based on the access pattern of an object, using the Index object as a case study. Finally, we measure the performance of compaction and checkpointing.

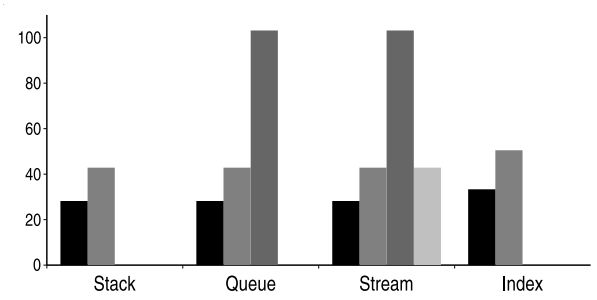

Figure 7. Breakdown of energy consumed by the operations supported by core Capsule storage objects storing 64 bytes of data. The operations resulting in writes to FAL consume substantially less energy than the read operations.

7.2.1 Energy consumption of Object operations

Table 4 shows an analysis of the energy consumption of the access methods supported by the core objects. In this experiment, we benchmark the energy consumption of methods on different objects for a specific choice of operating parameters. For the Stack, Queue, and Stream objects, we assume that there are 5 elements stored in the object ($N = 5$), each of size 64 bytes ($d = 64$). The Index object is assumed to have a two-level index where the second level is cached in memory, and the first-level is read from and written to flash as discussed in Section 5.2.4. We assume that each node of the index structure can hold 5 pointers ($k = 5$) to the next lower level. In addition to this, the FAL uses a 512 byte write buffer. Our micro-benchmarks are specific to this set of N, d and k parameters.

Figure 7 provides a breakdown of the energy consumption of individual methods supported by Capsule objects. The energy cost of the `push` operation supported by the Stack is 34% lower than that `pop`, as the `push` is buffered in FAL while the `pop` results in a read to flash. The `enqueue` operation on the Queue object has the same cost as the `push` operation as both of them effectively add a new element to the front of the list. These operations are all independent of N. The first `dequeue(1)` operation corresponds to removing one element from the Queue and is 3.7 times more expensive than `enqueue`, since it results in additional pointer traversal upto the start of the list (Section 5.2.2). However, the cost of the fifth dequeue, `dequeue(5)` is the same as that of a stack pop, showing the dependence of `dequeue` cost on N.

The Stream object combines traversal methods of both the Stack and the Queue – the `append` operation maps to a `push`, `previous` operation maps to a `pop` and the `next` to the `dequeue`. The cost exhibited by the Stream in Figure 7 confirm this. The `get` operation supported by the Index object turns out to be 52% more expensive than the `set` method and independent of N – again, this is due to write-buffering being performed by FAL for the `set`. Our measurements verify the cost analysis presented in Table 4.

7.2.2 Impact of Access Pattern and Chunk Size

The access pattern of an object has considerable impact on the energy consumed for object creation and lookup, and we evaluate this in the context of the Index object (refer Section 5.2.4). We consider four different access patterns in this study: sequential and random insert, and sequential and random lookup. Our evaluation has two parts. First, we analytically determine the energy cost for each access pattern. Second, we quantify the cost for different combinations of insertion and lookup to identify the best choice of chunk size in each case. This study only considers the cost of indexing, and does not include the cost of storing or accessing the *opaque data* pointed to by the index.

Cost Analysis: We use the following terms for our analysis: the size of each index node is d, the number of pointers in each index node is k, and the height of the tree is H. The cost of writing d bytes of data is $W(d)$ and reading is $R(d)$ as per Equations 1 and 2 respectively.

Insertion into the Index object has two steps — first, H index chunks (root to leaf) are read from flash to memory, then insertion results in H index writes from the leaf up the root as we described in Section 5.2.4. Index lookup operations need to read in the H index chunks corresponding to each level of the tree before retrieving the stored data.

Sequential Insert: If data is inserted in sequence, nodes of the index tree can be cached and are written only when the next element crosses the range supported by the node — this reduces the number of re-writes. Since one chunk write and chunk read is performed for each of the H levels for every k elements, the amortized cost associated with inserting an element sequentially is: $\frac{H}{k} \cdot (W(d) + R(d))$.

Random Insert: If data is inserted randomly, each write results in a read followed by write of an index chunk at each level of the index. The cost of is: $H \cdot (W(d) + R(d))$.

Sequential Lookup: Similar to writes, sequential reads can take advantage of index node caching and prove to be cheaper – the amortized cost of sequential lookup is: $\frac{H}{k} \cdot R(d)$.

Random Lookup: Random reads force each level of the index to be loaded afresh for each lookup operation, increasing the lookup cost to $H \cdot R(d)$.

Measurement-driven Analysis: Figure 8 quantifies the index insertion and lookup costs for varying chunk sizes for an index of height $H = 2$, based on our cost analysis and index measurements. In this experiment, we fix the number of elements inserted into the index at 32768 (*i.e.* $N = 32768$) and vary the size of each index node d, and thus the number of pointers, k, in each index node. We now discuss how to choose chunk sizes to suit the insert and lookup pattern.

Random Insert - Random Lookup: This scenario corresponds to a value-based index, where elements are inserted randomly and the lookup is for a specific value. For this case, Figure 8 shows us that the optimal size of each index node is the smallest possible – in this case 64 bytes or 15 elements per node, where each pointer is 4 bytes long.

Random Insert - Sequential Lookup: This scenario will arise when the time-series data is stored in a value-based index and the entire index is read sequentially to build, say, a probability distribution of the data. The choice of index node size depends on number of inserts as well as the number of lookups in this case – the insert is optimal at 64 bytes while the lookup become more efficient after 256 bytes. Small number of lookups in comparison with inserts mean that we

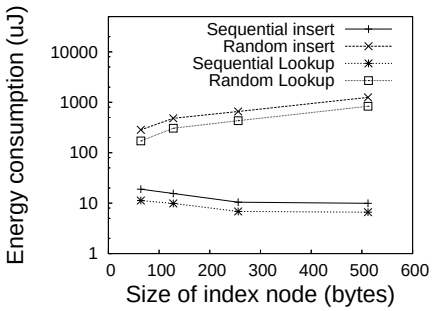

Figure 8. **Energy consumption of the index for varying insertion and lookup operational patterns, varying index node sizes – these are assume no associated data. Sequential insert and lookup are substantially more energy efficient than their random counterparts.**

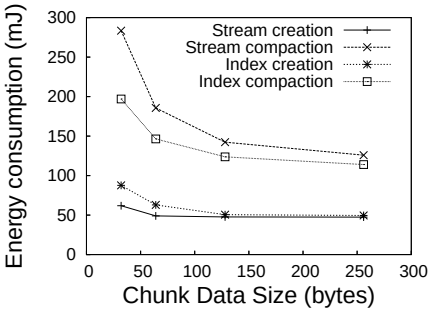

Figure 9. **The energy consumed by compaction not only depends on the amount of data, but also on the size of each data chunk of the object. The energy consumed by an Index and a Stream object holding 128KB of data is shown here for varying chunk data sizes. Larger object-level buffering requires fewer number of chunks to be read and written – the compaction costs more than double when changing buffering strategy from 32 bytes to 256 bytes.**

should optimize the insert, while greater number of lookups indicate that the lookup should be optimized.

Sequential Insert - Sequential Lookup: An index maintaining time series data would store and later access the data sequentially. Larger chunk sizes result in better energy optimization, however, a buffer size of 256 bytes is sufficient as both insert and lookup costs are close to their lowest value.

Sequential Insert - Random Lookup: An index maintaining time-series data would store data sequentially, but temporal queries on past data can result in random lookups. The optimal size again depends on the number of inserts and lookups – the insert is optimal at 64 bytes while the lookup become more efficient after 256 bytes. The ratio of the number of lookups to inserts would determine the choice of index size (similar to the random insert-sequential lookup case).

Our experiments show that smaller index chunk sizes are favorable for random insertion and lookup operations since smaller sizes lead to lower cost of flash operations. Larger chunk sizes are better for sequential operations, since they utilize buffering better, resulting in greater in-memory updates and fewer flash operations.

7.2.3 Memory Reclamation Performance

Memory reclamation (Section 4.2) is triggered using the compaction interface (see Appendix A) when the flash fills upto a pre-defined threshold. Our current implementation uses a simple compaction scheme where the storage objects read all their valid data and re-write it to the current write frontier on the flash. We select the stream and index objects for this experiment. The compaction procedure and costs for the stack and queue objects are identical to those of the stream object (Table 4).

In our experimental setup, we trigger compaction when 128 KB of object data has been written to flash; our goal is to find the worst case time taken for compaction. In the case of the Stream object, an intermediate stack is used to maintain ordering of the elements post-compaction, as discussed in Section 5.2.3. For the 2 level Index object (discussed in Section 5.2.4), we set the second level of the index to hold 100 pointers to level 1 index nodes ($k_1 = 100$) and each level 1 node holds pointers to 50 data blobs ($k_2 = 50$). In the experiments, we vary the size of the data being stored in each object chunk from 32 bytes to 256 bytes, in order to measure the range of compaction costs. We first perform a measurement of the energy consumption of the compaction process followed by a measurement of the time taken.

Energy consumption: Figure 9 shows the energy cost of compaction in comparison to the cost of sequential data insertion. We first consider the write and compaction costs for the Stream object – we observe that increasing chunk size reduces the cost of writing and compacting. The reduction in write costs is attributed to reduced header overhead of writing fewer chunks. The reduction in the stream compaction cost is considerably greater. As the size of data chunks increase, the number of elements in the stream decreases, which results in fewer pointer reads and writes to the intermediate stack during the compaction phase. Additionally, the efficiency of both read and write operations improves as the data size increases (refer Section 7.1). The compaction overhead can be reduced considerably by increasing the chunk size from 32 to 128 bytes – in fact, the savings equal about three times the cost of writing the original data. Further increase in chunk size results in smaller improvements in compaction performance.

The write and compaction costs for the Index object follow a similar overall trend. Interestingly, the write cost for the Index object is greater than that of the Stream object whereas the compaction cost of the Stream object is considerably higher than that for the Index object. This is because creating an Index object is more expensive due to the writing and reading of the level 1 index nodes. The compaction of the Index is less expensive than Stream compaction because Index compaction requires only a depth-first traversal of the index, while Stream compaction requires the creation and traversal of the intermediate pointer stack, which requires additional energy. If the fraction of discarded data is f for either the stream or the index, then the cost of compaction will be $(1-f)$ times the corresponding point in Figure 9.

Latency: Figure 10 shows the latency of the compaction operation. This is also an important measure of compaction as no operation can be performed on the object while the object is being compacted. We find that in all cases the entire

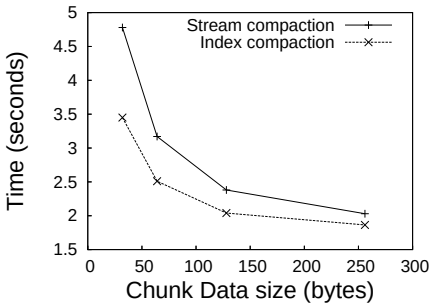

Figure 10. **The compaction time of the storage object is linked to both the amount of data the object holds and the size of each data chunk. The time taken to compact an Index and a Stream object holding 128KB of data is shown here for different data chunk sizes.**

Operation	Latency (μs)	Energy consumption (μJ)
Checkpoint	996	82.5
Rollback	284	42.1
Restore	460	50.87

Table 5. **Energy consumption and latency of performing checkpointing operations on a Stream and Index object.**

compaction operation executes in less than 5 seconds. This can be improved to 2.5 seconds for the Stream and to 2 seconds for the Index by increasing the data size to 128 bytes. This shows us that even while compacting 128K of object data, the storage object will be unavailable only for a short duration and this can be dealt with easily by providing some minimal application-level buffering.

The energy and latency results of compaction show that these operations can be performed efficiently on a small sensor platform. We find that a buffer size of 128 bytes provides a good balance between the memory needs of compaction and the energy consumption/latency of the process.

7.2.4 Checkpointing

Capsule supports checkpointing with the help of the special Checkpoint component that permits three operations: `checkpoint`, `rollback` and `restore`. For our experiment, we consider a Stream and an Index object and link these to a single Checkpoint component. We then perform each of the operations permitted on the Checkpoint component and measure the latency of the operation and the energy consumed by the device – Table 5 presents our results. We see that the latency of all the operations is less than 1 ms. The energy consumption of the `checkpoint` operation is approximately 3 times that of a stack `push` operation or only 2 times that of a `pop` operation with 64 bytes of data. The energy consumed by the `restore` operation is a little more than that of performing a `pop`, and the cost of `rollback` is equivalent to the cost of performing a `pop` operation on the stack. These measurements indicate that checkpointing support in Capsule is extremely low-cost and energy-efficient, allowing Capsule to support data consistency and crash recovery with minimal additional overhead.

7.3 Experimental Evaluation

Having discussed the performance of the basic objects provided in Capsule, we evaluate how these objects can be used by applications and system components. In particular, we highlight how the choice of appropriate object parameters can impact the overall energy consumption. We discuss an archival storage and indexing application that performs sensing, storage and communication. Our second experiment performs an empirical comparison between our Capsule file system implementation and Matchbox.

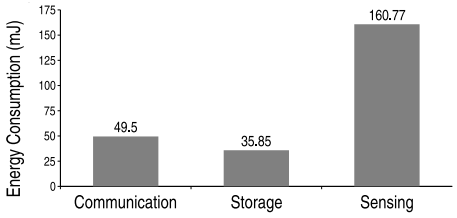

Figure 11. **Component level breakdown of a Mica2 application using the light sensor (MicaSB - MTS 300) and storing 12000 sensor readings in a Stream-Index object in batches of 60 readings each. It also stores batches of 1200 readings in an Index object and performs a computation on this stored data, batching 10 of these results and transmitting them (1% duty cycling). The measurements shown are per cycle.**

7.3.1 Archival Storage and Indexing

Sensors are commonly used for archival storage and indexing and this experiment focuses on the energy consumption of Capsule in comparison to the storage, communication and sensing sub-systems. Our Mica2 application uses the photo sensor on the MicaSB [2] to measure the light intensity once every second; 60 sensor readings are batched in-memory, and then stored in the stream-index object. This batch is also stored in a separate index object and every 1200 sensor readings, we access the stored readings performing a simple averaging operation requiring sequential traversal of the data stored in this index. The result is stored as a summary along with a tag indicating the relevant part of the stored data stream in the stream-index object. When 10 summaries accumulate, they are then transmitted together in a single 20 byte packet to the base-station using the CC1000 radio on the Mica2 [26], using BMAC [16] and set to 1% duty cycling. We neglect the effects of sleep, wakeup and packet loss, which in fact would adversely impact the sensing and radio measurements.

Figure 11 shows the results of our experiment – a component-level breakdown of the energy consumption of the storage, communication and sensing sub-systems for this application per cycle. We observe that Capsule consumes only 14.5% of the total energy consumption, having written 48000 bytes of sensed data and subsequently reading 24000 bytes. The communication subsystem comes in second, consuming 20.1% of the total energy to transmit only 20 bytes of data, with the sensing occupying the remaining 65.4% and capturing 12000 sensor readings.

This demonstrates that using Capsule in sensor applications is feasible and extremely energy-efficient, while permitting the application to process large datasets.

7.3.2 Comparison with Matchbox

We now compare our implementation of a file system based on Capsule (Section 5.3.2) with Matchbox[5]. Our

	Capsule		Matchbox	
	Energy (mJ)	Latency (ms)	Energy (mJ)	Latency (ms)
Create	1.79	19.16	1.03	14.16
Write (80b x 10)	8.83	85.6	10.57	91.60
Open	0.0093	0.184	0.093	1.384
Read (80b x 10)	1.20	18.440	1.12	16.520
Total (c+w,o+r)	11.83	123.4	12.82	123.7
Write Bandwidth	18.0kbps		11.3kbps	
Read Bandwidth	54.2kbps		60.4kbps	
Memory Foot-print	1.5K RAM, 18.7K ROM		0.9Kb RAM, 20.1K ROM	

Table 6. Energy consumption and latency of Matchbox and Capsule operations.

implementation also provides the following additional features: the ability to work with multiple files simultaneously, random access to a block in the file, modifying previously written data, and file consistency guarantees even in the event of system failure in the middle of a write operation.

Our experiment was performed on the Mica2 [26], using the platform's Atmel NOR flash. On both file systems, we created a new file and wrote 80 bytes of data in each of 10 consecutive operations (a total of 800 bytes). We then closed the file, re-opened it and read the 800 bytes similarly in 10 consecutive read operations of 80 bytes each. Table 6 shows the performance of the Capsule file system in comparison to Matchbox. The memory foot-print of both file systems is comparable; providing support for checkpointing as well as buffering at FAL, file and the index objects are the reason for the higher RAM foot-print of Capsule. The individual energy consumption of file system operations on both is comparable. The write bandwidth provided by the Capsule file system is 59% more than Matchbox while the read bandwidth lags by 10%. Considering the net energy consumption of the experiment, the Capsule file system turns out to be 8% more energy-efficient than Matchbox while taking approximately the same amount of time. *Thus, our Capsule file system implementation provides rich additional features at an energy cost equivalent or less than that of Matchbox.*

8 Related Work

There have been four other efforts at building a sensor storage system that we are aware of: Matchbox [5], ELF [3], MicroHash [27] and TFFS [4]. Other filesystems like YAFFS2 [25] and JFFS2 [24] are targeted at portable devices such as laptops and PDAs and do not have sensor-specific implementations. A direct head-to-head comparison between Capsule and Matchbox is provided in Section 7.3.2, where we show that Capsule provides additional features at a lower or comparable cost. One of the difficulties that we faced in performing a direct quantitative comparison against MicroHash was that it had been implemented on a custom node (the RISE platform [13]) which has significantly greater memory than available on the Motes that we use. Thus, our discussion of the relative merits and demerits of these approaches is restricted to a qualitative one.

Energy Efficiency: Of the systems that we compare, only MicroHash and Capsule make claims about energy efficiency. MicroHash is, in essense, an implementation of Capsule's Stream-Index object for SD-cards with greater emphasis on the indexing and lookup techniques than in our paper. A fundamental difference between the two systems is that MicroHash uses a page buffer for reads as well as writes, and does not provide the ability to tune the chunk size to the access pattern. This is unlike our system, which can adapt the choice of the chunk sizes to the insert and lookup patterns, thereby better optimizing energy-efficiency (Section 7.2.2).

Portability: Embedded platform design is an area of considerable churn, as evident from the plethora of sensor platforms that are being developed and used by research groups. Storage subsystems for these platforms differ in the type of flash (NAND or NOR), page size (256b to 4096b), erase block size (256b to 64KB), bus speeds (SPI or parallel), and energy consumption. It is therefore essential to design a general purpose storage system that can be easily ported to a new platform with a new storage subsystem, while being sufficiently flexible to enable developers to take advantage of new architectures. We believe Capsule achieves these dual goals – it currently works on the Mica2, Mica2dot (both NOR) and our custom NAND board; the Telos port is work in progress.

Functionality: In comparison to the research effort that has gone into the design of the radio stack on sensors, there have been relatively few efforts at building the sensor storage system. As storage becomes a more important part of sensor network design, increased attention is needed to address questions of storage capacity, failure handling, long-term use, and energy consumption that are not addressed by existing efforts. Capsule attempts to fill this gap by building up a functionally complete storage system for sensors.

Other Efforts: There have been a number of object and file systems developed for disks that relate to our work, such as LFS, a log-structured file system [21], and Vagabond [15] a temporal log-structured object database. However, Capsule is designed specifically for sensor platforms using NAND or NOR flash memory based storage and for energy efficiency, rather than for goals of disk-based read-write optimization, security and network sharing. The differences in hardware and optimization metrics also makes the compaction techniques that we use in Capsule significantly different from storage reclamation techniques for disks such as hole-plugging [23] and heuristic cleaning [1].

9 Conclusions

Storage is an essential component of data-centric sensor applications. Recent gains in energy-efficiency of new-generation NAND flash storage strengthen the case for in-network storage by data-centric sensor network applications. In this paper, we argue that a simple file system abstraction is inadequate for realizing the full benefits of flash storage in data-centric applications. Instead, we advocate a rich object storage abstraction to support flexible use of the storage system for a variety of application needs and one that is specifically optimized for memory and energy-constrained sensor platforms. We proposed *Capsule*, an energy-optimized log-structured object storage system for flash memories that enables sensor applications to exploit storage resources in a multitude of ways. Capsule employs a hardware abstraction layer that hides the vagaries of flash memories from the application and supports highly energy-optimized implementations of commonly used storage objects such as streams,

files, arrays, queues and lists. Further, Capsule supports checkpointing and rollback to tolerate software faults in sensor applications running on inexpensive, unreliable hardware. Our Capsule implementation is portable and currently supports the Mica2 and Mica2Dot NOR flash as well as our custom-built NAND flash memory board. Our experiments demonstrated that our system provides greater functionality, more tunability, and greater energy-efficiency than existing sensor storage solutions, while operating within the resource constraints of the Mica2.

Future Work: We plan to examine the platform-specific design of Capsule in light of more resource-rich platforms such as the iMote2. For example, a memory-rich platform would allow Capsule to use a per-object log-segment allocation strategy that would place each object's data chunks contiguously, permitting FAL to do read-buffering. We are also working on fabricating an SPI based NAND flash daughter board for the Telos.

Acknowledgements: We thank our anonymous reviewers and our shepherd Philippe Bonnet for their valuable feedback and comments. This research is supported in part by NSF grants EEC-0313747, CNS-0626873, CNS-0546177, CNS-052072, CNS-0325868, and EIA-0080119.

10 References

[1] T. Blackwell, J. Harris, , and M. Seltzer. Heuristic cleaning algorithms in Log-structured File Systems. In *USENIX Winter Conf*, pages 277–288. Jan 1995.

[2] MicaSB Sensor board MTS 300. `www.xbow.com`.

[3] H. Dai, M. Neufeld, and R. Han. ELF: An efficient Log-structured Flash file system for micro sensor nodes. In *SenSys*, pages 176–187, New York NY, 2004.

[4] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *USENIX* , pages 89–104, Anaheim CA, Apr 2005.

[5] D. Gay. Design of Matchbox: The simple Filing system for Motes. In TinyOS 1.x distribution, `www.tinyos.net`, Aug 2003.

[6] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond Average: Towards sophisticated sensing with Queries. In *IPSN*, Palo Alto CA, 2003.

[7] J. W. Hui and D. Culler. The Dynamic behavior of a Data dissemination protocol for Network programming at Scale. In *SenSys*, Nov 2004.

[8] P. Levis, S. Madden, J. Polastre, et al. TinyOS: An Operating System for wireless Sensor networks. In *Ambient Intelligence*. Springer-Verlag, 2005.

[9] M. Li, D. Ganesan, and P. Shenoy. PRESTO: Feedback-driven Data management in Sensor networks. In *NSDI*, May 2006.

[10] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An Acqusitional Query Processing system for Sensor networks. In *ACM TODS*, 2005.

[11] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low Power Data Storage for sensor networks. In *IPSN/SPOTS*, Nashville N, Apr 2006.

[12] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-structured file systems with Adaptive methods. In *SOSP*, pages 238–251, 1997.

[13] A. Mitra, A. Banerjee, W. Najjar, D. Zeinalipour-Yazti, D. Gunopulos, and V. Kalogeraki. High-Performance, Low-Power sensor platforms featuring Gigabyte scale Storage. In *SenMetrics*, San Diego CA, Jul 2005.

[14] E. B. Nightingale and J. Flinn. Energy-efficiency and Storage Flexibility in the Blue file system. In *OSDI* , San Francisco CA, Dec 2004.

[15] K. Nrvag. Vagabond: The design and analysis of a Temporal Object database management system. PhD thesis – Norwegian University of Science and Technology, 2000.

[16] J. Polastre, J. Hill, and D. Culler. Versatile Low power Media Access for wireless sensor networks. In *SenSys*, Nov 2004.

[17] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-low Power wireless research. In *IPSN/SPOTS*, Apr 2005.

[18] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor network Debugger. In *SenSys*, Nov 2005.

[19] S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, L. Y. S. Shenker, and F. Yu. Data-centric storage in sensornets. In *HotNets*, 2001.

[20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT - A Geographic Hash-table for Data-centric storage. In *WSNA*, 2002.

[21] M. Rosenblum and J. K. Ousterhout. The design and implementation of a Log-structured File system. *ACM TOCS*, 10(1):26–52, 1992.

[22] Toshiba America Electronic Components, Inc. (TAEC), `www.toshiba.com/taec`. *Datasheet: TC58DVG02A1FT00*, Jan 2003.

[23] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage system. *ACM TOCS*, 14(1):108–136, 1996.

[24] D. Woodhouse. Journalling Flash File System. `http://sources.redhat.com/jffs2/jffs2.pdf`.

[25] Aleph One. Yet Another Flash File System. `www.aleph1.co.uk/yaffs`.

[26] Xbow. Mica2 Data sheet. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-0%6_A_MICA2.pdf`.

[27] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An efficient Index structure for Flash-based sensor devices. In *USENIX FAST*, SF CA, Dec 2005.

Appendix A Capsule Interfaces

```
interface FAL {
  command result_t write(ObjectPtr optr, bool chksum,
                         FlashPtr fptr);
  event void writeDone(result_t res);
  command result_t read(ObjectPtr optr, bool chksum);
  event void readDone(ObjectPtr optr, result_t res);
  command result_t rawWrite(DataPtr optr, bool chksum,
                            FlashPtr fptr);
  event void rawWriteDone(result_t res);
  command result_t rawRead(DataPtr optr, bool chksum);
  event void rawReadDone(DataPtr optr, result_t res);
  command result_t flush();
  event void flushDone(result_t res);
}

interface Compaction {
    command result_t compact();
    event void compactionDone(result_t res);
}

interface Stream-Index {
    command result_t init(bool ecc);
    command result_t add(StreamIndexPtr data, datalen_t len);
    event void addDone(result_t res);
    command result_t setTag();
    event void setTagDone(result_t res, uint tag);
    command result_t getTag(uint tag, StreamIndexPtr data,
                            datalen_t *len);
    event void getTagDone(result_t res);
    command result_t seek(uint skipBackNodes);
    event void seekDone(result_t res);
    command result_t next();
    command result_t previous();
    event void traversalDone(result_t res);
    command result_t invalidate();
}

interface Serialize {
    command result_t checkpoint(uint8_t *buffer, datalen_t *len);
    command result_t restore(uint8_t *buffer, datalen_t *len);
}

interface Checkpoint {
    command result_t init(bool priority);
    command result_t checkpoint();
    event void checkpointDone(result_t result);
    command result_t rollback();
    event void rollbackDone(result_t result);
}
```