ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes

Hui Dai Michael Neufeld Richard Han
University of Colorado at Boulder
Computer Science Department
Boulder, CO, 80302
{huid, neufeldm, rhan}@cs.colorado.edu

Categories and Subject Descriptors

E.5 [**File Systems Management**]: Miscellaneous; D.4.3, F.2.2, H.2 [**FILES**]; I.2.9 [**Sensors**]

General Terms

Design Management Reliability

Keywords

file system, flash, sensor, reliability, log structured, eeprom

ABSTRACT

An efficient and reliable file storage system is important to micro sensor nodes so that data can be logged for later asynchronous delivery across a multi-hop wireless sensor network. Designing and implementing such a file system for a sensor node faces various challenges. Sensor nodes are highly resource constrained in terms of limited runtime memory, limited persistent storage, and finite energy. Also, the flash storage medium on sensor nodes differs in a variety of ways from the traditional hard disk, e.g. in terms of the limited number of writes for a flash memory unit. We present the design and implementation of ELF, an efficient log-structured flash-based file system tailored for sensor nodes. ELF is adapted to achieve memory efficiency, low power operation, and tailored support for common types of sensor file operations such as appending data to a file. ELF's log-structured approach achieves wear levelling across flash memory pages with limited write lifetimes. ELF also uniquely provides garbage collection capability as well as reliability for micro sensor nodes. A performance evaluation of an implementation of ELF based on TinyOS and MICA2 sensor motes is presented.

1. INTRODUCTION

The popularity of wireless sensor networks (WSNs) as an important new research domain has grown dramatically [1]. WSN systems are capable of providing novel distributed in-situ sensing of environmental phenomena, and typically consist of a combination of small, resource-constrained micro sensor nodes that collect data and cooperatively relay the data back to a more powerful collection point. Building such a system requires the integration of multiple hardware platforms, operating systems, network systems, and back-end data services. Standard micro sensor systems include Berkeley's Mote/TinyOS architecture [2], MIT's cricket [3, 4], the MANTIS system [5], Europe's Smart-Its [6], Eyes [7], and BTNodes [8] projects.

Some of the above systems have been deployed in a wide variety of real world situations with different data storage requirements. In some cases, sensor readings should be immediately relayed from the sensor nodes to the collection point, *e.g.* in the deployment on Great Duck Island to monitor the habitat of seabirds [9], and forest monitoring [10]. Other applications may only permit sporadic transfer of data to the data sink, such as vineyard monitoring [11]. In both cases, it is essential to be able to flexibly and efficiently exploit local storage on the sensor nodes. Even if data relaying occurs frequently, hardware failure or environmental conditions might also prohibit timely relaying in some cases. In these cases, it is important to have the option of holding onto data locally until a relaying opportunity arises. This asynchronous communication is shown in Figure 1. Furthermore, sensor networks often operate with a very low duty cycle for radio transmission in order to conserve energy. If the periodicity of radio transmission is significantly longer than the sensing periodicity, then multiple sensor samples will be accumulated between two transmissions. These data also need to be stored locally.

There are several candidates for local storage of sensor samples on micro sensor nodes. While run-time RAM is an option, the run-time RAM on micro sensor nodes is exceedingly scarce, *e.g.* 4 KB for today's motes [12]. This scarce resource can be quickly overwhelmed by logged sensor data. Moreover, RAM is also needed for execution of code, besides serving as storage for data. Excessive logging of sensor data could inhibit the ability of both applications and the operating system to execute as designed. In comparison, persistent storage on micro sensor node offers a more attractive alternative at the cost of slower read/write speed. Persistent storage on micro sensor nodes typically offers about two orders of magnitude more memory than RAM, *e.g.* 512 KB of external flash for Berkeley MICA2 motes. In addition, persistent storage enables the design of reliability mechanisms to recover sensor data after crashes. Storing sensor data in persistent storage instead of RAM thus enables larger storage, reduces interference with RAM executables, and enhances robustness. Currently, most sensor platforms employ solid state flash memory for persistent storage.

An efficient yet reliable file system that handles the details of interacting with persistent storage on micro sensor platforms can greatly ease the process of collecting sensor data. There are already some file-based applications,*e.g.* TinyDB [13, 14], that are useful

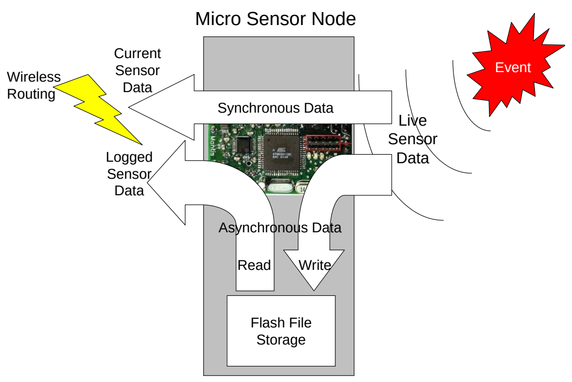

Figure 1: Micro sensor nodes that log sensor data in flash can asynchronously communicate their data at a later time.

to the sensor network. A rudimentary file abstraction providing basic file operations such as *open()*, *read()*, *write()*, and *delete()* would assist these applications in managing local storage. A file system is also useful for flexibly organizing and partitioning local storage, *e.g.* saving OS configuration parameters or even full binary system images as required so that sensor nodes can be dynamically reprogrammed [15, 9, 5, 16, 17]. Such system-critical tasks typically have higher reliability requirements than simple data recording. These reliability and robustness requirements could be met by a sufficiently capable file system.

The design and implementation of a file system for sensor nodes faces a variety of challenges. First and foremost, micro sensor nodes are typically quite resource-constrained in terms of CPU, memory, radio bandwidth, and energy. These constraints often arise because of application-dictated cost and/or size requirements. For example, the MICA2 mote operates using a low power Atmega microcontroller (4 or 7 MHz, 4 KB RAM, 4 KB internal EEPROM, 128 KB internal flash, 512 KB external flash). The limited run-time memory available severely inhibits the ability of a file system to keep large data structures in RAM or use large caches to improve performance. As a result, many standard file systems that assume sufficient RAM resources for operation and caching cannot be directly applied to micro sensor nodes.

Another challenge in the design and implementation of sensor file systems is that the characteristics of flash memory are far different from the traditional hard disk. Most sensor platforms employ flash memory for persistent storage because of its high reliability, high density, and relatively low cost. Compared with the conventional hard disk, the flash memory has several unique features. Flash memory is usually divided into sectors or blocks, which are further divided into “pages”. The size of each flash memory page is fairly small, *e.g.* 256 bytes on the mica2 platform with an extra 8 bytes for “out of band” storage space, which is intended to be used for metadata or error correction codes. Writing to flash memory is on a per-page basis, and consumes significantly more time and energy than a read operation. Flash access times are comparable to disk access times, on the order of tens of milliseconds, but are fixed rather than variable. Due to these fixed flash access times, many file system optimizations to reduce disk seek times and rotational latency, such as geographically collocating files, are no longer applicable. Flash also exhibits a unique property, in that each flash memory page can only endure a limited number of rewrites, typically about 10,000 operations. Repeated writes to the same page will quickly exhaust the lifetime of a flash page. To ensure that no single flash page reaches its lifetime limit before the rest of the flash memory pages, it is important to ensure that erase-write cycles are evenly distributed around the flash; a process normally called *wear levelling* [18]. File systems that handle these issues have been constructed, but not in the severely resource-constrained context of sensor nodes.

Our objective in this work is to design and implement an efficient, reliable and flexible file system for micro sensor platforms. Since solid state flash memory is likely to be used on micro sensor platforms for storage, the design is flash-based. A key design principle of the ELF sensor file system is to achieve memory and energy efficiency with a small RAM footprint. ELF employs a log structured file system in order to manage flash pages so that they age evenly, thereby promoting wear levelling. The log-structured system is also leveraged to provide reliable recovery. ELF’s implementation is based on the mica2 platform although the same design principles could be applied to other micro sensor platforms.

The paper is organized as follows. Section 2 presents related work in file systems. Section 3 discusses the design goals and architecture for ELF. Section 4 describes the specific implementation choices of ELF, including how ELF implements common file operations, allocates space, collects garbage, and maintains consistency as well as reliability of files. Section 5 presents the performance evaluation of the ELF file system on the mote micro sensor platform. Section 6 discusses several remaining issues and future work. Finally, Section 7 concludes.

2. RELATED WORK

ELF draws on concepts and principles in traditional log structured file systems, flash-based file systems, and existing file systems used in sensor nodes. In this section, we discuss relevant examples of prior work in each of these areas.

2.1 Log Structured File Systems

Sprite LFS [19] introduced the concept of log based file system design. The file system is represented as a log of metadata. One important feature of LFS is that a large number of data blocks are gathered in a cache before writing to disk in order to maximize the throughput of collocated write operations, thereby minimizing seek time and accelerating the performance of writes to small files. BSD-LFS [20] enhances the performance of log-structured file systems by improving upon the block allocation policies of garbage collection. These log-structured file systems assume that files are cached in main memory, and that increasing cache sizes will improve performance of read and write requests. For micro sensor nodes, there is insufficient RAM to support a large write cache. Also, gathering writes to improve disk seek time is not an appropriate motivation for flash memory. Seek time is much less of an issue for flash memory, though there is still a small, fixed latency cost to access a new page.

2.2 Flash File Systems

Several flash-based file systems have been designed in the literature to work with popular operating systems. The flash-memory-based storage system eNVy [21] tries to provide high performance in a transaction-type application area. It consists of a large amount of flash memory, a small amount of battery-backed SRAM for write buffering, and a large-bandwidth parallel data path between RAM and SRAM. A controller is used for page mapping and cleaning. In addition to the hardware support, eNVy uses a combination of two cleaning policies, i.e. FIFO and locality gathering, in order to minimize the cleaning costs for both uniform and hot-and-cold access distribution. Microsoft Flash File System (MFFS) [22] provides MS-DOS-compatible file system functionality with a flash memory card. It uses data regions of variable size rather than data blocks of fixed length. Files in MFFS are chained together by using address pointers located within the directory and file entries. Douglis et al. [23] observed that MFFS write throughput decreased significantly with more cumulative data and with more storage consumed.

The work presented in [18], as well as many other current applications of flash, utilize the flash to emulate a block device. It simulates the smaller sector size for write requests by reading the whole erase block, modifying the appropriate part of the buffer, and then erasing and rewriting the entire block. Standard file systems are constructed over the emulated devices. One common practice is to have sectors of the emulated block device stored in varying locations on the physical medium, and a "Translation Layer" [22] is used to keep track of the current location of each sector in the emulated block device. This approach is inefficient and can result in insufficient wear levelling.

A far more efficient use of flash technology would be to employ a file system designed specifically for use on such devices, with no extra layers of translation in between. JFFS and JFFS2 provide such native solutions [24]. JFFS and JFFS2 are journaling flash based file systems that keep journalled metadata in order to avoid errors and corruption. In order to achieve efficient memory usage, they do not use an extra layer of indirection, such as a translation table mapping between virtual blocks and the actual flash memory. JFFS and JFFS2 are designed for use on flash-based PDA systems, e.g. the Hewlett Packard iPAQ. As in LFS, each write in JFFS2 creates a new record in the log. ELF adopts a logging approach similar to JFFS2, but differs in several important respects. First, ELF does not create a new node for every write operation. Write-appends do not generate new nodes, only write-modifies. This reduces runtime memory consumption for micro sensor platforms. In addition, ELF provides a best-effort reliability mechanism.

2.3 Sensor Storage and File Systems

Distributed databases and data storage systems such as DCS [25, 26] and Dimension [27, 28] have been studied for sensor networks. These works focus on providing high level distributed event storage in sensor networks using techniques such as a geographic hash table [29]. The low-level storage implementation on each sensor platform is assumed to be a simple circular log structure. In contrast, ELF is focused on designing and implementing a storage system within a micro sensor platform that provides more capabilities, essentially a mini file system.

Up until now, the only file system that has been available for sensor networks, besides ELF, is Matchbox [30]. Matchbox allows application to open several files simultaneously. Matchbox also provides rudimentary wear levelling and remote access file management. The code size is small, e.g. 10 KB, and the minimum footprint is 362 bytes, which will increase as the number of files increases. Matchbox provides only append operations and doesn't allow random access, such as write modifications, to existing data in a file, unlike ELF. Matchbox provides a CRC checksum for each flash memory page that is used to verify the integrity of the file during recovery from a system crash.

3. DESIGN OVERVIEW

3.1 Design Goals

This work is targeted at providing a practical, efficient, and reliable file system for micro sensor nodes that employ flash memory for persistent storage. The design goals of the ELF file system are to:

- Allow access to flash memory with simple file operations
- Extend the operational lifetime of the flash with wear levelling techniques
- Achieve a small memory footprint
- Optimize common sensor file operations
- Avoid excessive energy consumption
- Provide optional best-effort data reliability

Before describing how ELF achieves these design goals, we will first discuss the characteristics of the data that ELF expects to handle, the common operations that ELF is tailored to perform on these data, and the attributes of the flash medium in which the files will be stored.

3.2 File Access Behavior on Sensor Nodes

ELF expects to encounter three major sources of data: configuration data, binary images, and sensor data. Each source has different expected access patterns and reliability requirements. Generally, the degree of reliability required for sensor data is likely to be satisfied by verifying the integrity of the logged data via CRC/checksum. Greater reliability, such as recovery after a crash, may also be desirable, though this is not expected for all files. Reliability is far more important for program images and configuration data.

- **Sensor Data** The majority of the data recorded in a sensor system is likely to be normal sensor readings. These data are typically written sequentially without modifying prior records. Also, these data records are likely to be erased periodically in order to make room for newer information.
- **Configuration Data** Configuration data will experience modification and updates in the post-deployment stage. However, the frequency of these changes is unlikely to be high. These configurations are considered critical to the correct functioning of the sensor node and require a higher level of reliability than data samples.
- **Binary Program Image** Dynamic reprogramming or retasking of sensor nodes is becoming increasingly important. One approach is to transmit the whole binary system image during reprogramming, *i.e.* reflash the entire OS. Others only transmit updates or patches in order to reduce communication overhead. [16] In either case, a version of the program image or image diff needs to be stored in flash first before the system can be rebooted to use the new software. This type of data is expected to require the highest degree of reliability since an error can easily make the entire sensor node unusable.

Based on the above analysis, we expect that the most common operation on files will be sequential appending and reading of sensor data. Write-Modify capability, *e.g.* for configuration data files, is provided but expected to occur much less frequently. Likewise, files that require a higher degree of reliability and crash resistance than provided by simple data block checksum are important, but not expected to be the common case. ELF provides an optional reliability mechanism to guarantee the consistency of such files.

Table 1: Flash Memory Attributes in Motes

Read a page into cache	less than 250 μs
Page Erase	8ms
Write Limit	10,000 times
Writing a Page	14ms
Erasing and Writing a Page	20ms
Typical Number of Pages	2048
Typical Page Size	264 Byte
Power Consumption	4mA Read Current 2μA Standby Current

Table 2: EEPROM attributes in atmega128

EEPROM Write	8848 cycles (8.5ms at 1Mhz)
EEPROM Read	1 cycle (From CPU) CPU halted 4 cycles after read
EEPROM Erase	about 4 ms
Write Limit	100,000
Power Consumption	2-8mA when programming

3.3 Persistent Storage Capabilities on Sensor Nodes

ELF is designed for flash memory since it is likely to be the storage medium for micro sensor nodes. In this section, we use the mica2's flash memory as an example to explain important flash memory characteristics in detail. The general attributes are listed in Table 1. A flash is usually divided into sectors, and each sector is divided into 264 bytes per page. Pages can only be erased and written as a whole. Data in flash memory can be either read out directly or loaded into an on-chip cache, whose size is 264 bytes, for later retrieval. In the latter case, a whole page is must be loaded into the on-chip cache each time. The data in the on-chip cache can be modified before being flushed to flash memory. Writing data to a flash memory page consists of two steps. Data is first written to the on-chip cache. Then a command is sent to flush the contents of the on-chip cache to a page in flash memory. This operation will first erase all the data in the target flash memory page and then copy the data in the cache onto this erased page. To modify a portion of a flash memory page, the whole page must first be read into the cache, modified, and then written back to flash in its entirety. Table 1 shows that write time is much larger than the read time. Flash read throughput is roughly 800 KB/sec while raw flash write throughput is only 10 KB/sec. Usually, flash memory prohibits concurrent read and write operations. Unlike most magnetic media, each flash page only has a limited write lifetime, on the order of 10,000 operations. To avoid exhausting a particular page, wear levelling techniques are needed in order to distribute writes throughout the flash. These are general properties of flash-based systems, and ELF will function on any similar flash-based storage medium.

As stated earlier, ELF provides best effort reliability mechanisms for certain files. Such reliability can be implemented based on a persistent storage medium such as flash memory. Instead, the implementation of ELF on the *mica2* is based on the internal EEPROM. The characteristics of EEPROM are described in Table 2. The essential general properties of this medium are that it is persistent, is slower than RAM, and is not as plentiful as flash. ELF takes advantage of this additional EEPROM space on motes to store the directory structure. This avoids having to store the directory in limited RAM or in more wear-limited flash. This also reduces boot time and file access time compared to having to reconstruct a newly accessed file without a directory. ELF further stores crash recovery data for the file system in EEPROM, leveraging the persistence of EEPROM to enhance reliability. Though directory updates and crash recovery metadata may rewrite the same page, wear levelling is not as urgent an issue for EEPROM, which has an order of magnitude greater lifetime than flash. For micro sensor nodes that lack EEPROM, ELF's reliability mechanisms can be adapted to use flash memory instead of EEPROM, with some cost in wear lifetime.

3.4 Log-Structured File System Techniques in ELF

A traditional log-structured file system creates a new sequential log entry for *each* write operation that occurs. This type of operation naturally encourages very good wear levelling since the flash memory may be used sequentially all the way through, only returning to previously used blocks after all of the blocks have been written to at least once. ELF encourages this even more by keeping each log entry on a separate flash page, *i.e.* there are never multiple log entries on a single flash page. This is designed from reliability considerations. If multiple log entries are stored in the same page, any damage to this page will result in the loss of all log entries.

Unlike traditional LFS techniques, ELF does not create a log entry for each Write-Append operations. This is because the runtime memory representation of a traditional log-structured file with many small appends will rapidly grow to unwieldy size if a log entry is created for each operations. In the interests of lowering runtime performance and memory consumption, an append operation in ELF utilizes an existing log entry on a flash page if one is available. While this may decrease wear levelling to some extent, wear levelling should still occur naturally since append operations will eventually fill up a page, requiring a new log entry and flash page to be used at that point. ELF uses a write cache for each file to gather appends to the same page so that the number of Write-Appends is further reduced. The details are explained in the next section.

For much less frequent Write-Modify operations, ELF does follow a traditional logging approach, i.e. modifications will not be overwritten to the same flash memory page, which enhances wear levelling. This takes advantage of the wear levelling property of write logs without overly extending the run-time representation of files. Each modification will be written to a new flash page and a log entry is created. Since Write-Modifies are not expected to be frequent, *e.g.* for occasionally changing configuration parameters or retasking, the file representation should grow only slowly and be able to fit within RAM in the common case.

Similar to classic log-structured file systems, a simple garbage collection mechanism is provided by ELF. The garbage collector, or cleaner, provides another opportunity to implement wear levelling in the reallocation of free flash pages. A write counter is kept in the metadata of each page. Using this write counter, the cleaner can preferentially mark pages with less wear for use.

In addition to Write-Modify and Write-Append operations, other standard file operations provided by ELF include reading and creation and deletion of files. ELF does not attempt to provide sophisticated optimization for all forms of file access, due to the resource constraints imposed by sensor systems. ELF only focuses on providing basic file operations for common tasks in a sensor system.

3.5 ELF Architecture Overview

Figure 2 illustrates the architecture of the ELF file system. At the

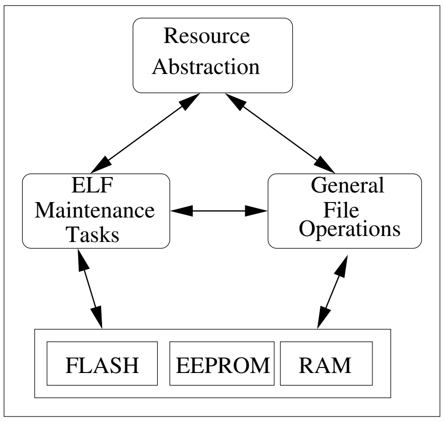

Figure 2: ELF File System Architecture

top is the "Resource Abstraction" maintained in run-time memory. It consists of the in-memory representations of open files, cleaning policy, and other ELF configuration data. The "General File Operations" is the logical abstraction of all file/directory operations. The "ELF Maintenance Tasks" is the abstraction of system maintenance tasks, *e.g.* maintaining a snapshot of directory structure and file metadata in EEPROM. It also includes the garbage collection task. When predetermined conditions are met, the garbage collection task will be posted, though these functions can also be explicitly called by the users. The cleaning policy is selected at compile time.

4. IMPLEMENTATION ISSUES

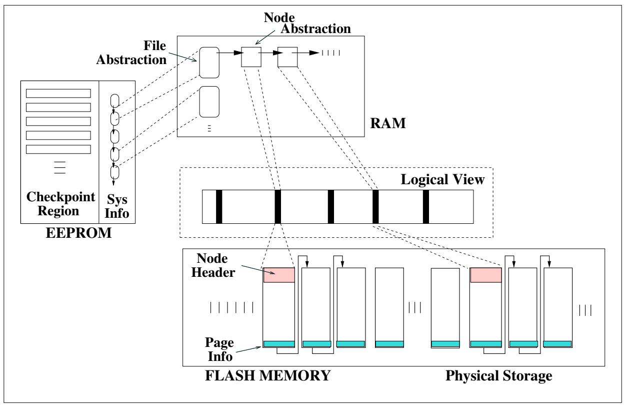

Figure 3: Distribution of data structures used by ELF in RAM, flash, and EEPROM.

This section begins with a discussion of the implementation of ELF on the Berkeley *mica2* platform. It first describes ELF data structures in RAM, flash, and EEPROM. The high level layout is presented in Figure 3. Basic file operations are then illustrated with examples. This is followed by explanations of ELF's garbage collection and data reliability mechanisms.

4.1 On-Flash Data Structures

4.1.1 Physical Nodes

The high-level layout of ELF in flash follows a single log format. The log in ELF consists of a sequence of physical nodes. A newly created node is appended to the end of the log. Currently, ELF supports three types of physical nodes in flash: file inodes, common nodes and directory entries. Each physical node is identified by a unique 16-bit node identifier. A 16-bit version number is associated with each physical node to indicate the age of the physical node. Another 32-bit length field identifies the length of this node and the included data. The three types of physical nodes are:

- **ELF_DIR**

  An "ELF_DIR" node represents a directory entry in the file system. As directory-related operations on a sensor network file system are expected to be simple, ELF only supports basic operations and a simple directory structure. In addition to the fields mentioned earlier, an "ELF_DIR" node also consists of the following fields

  – Directory Name

  – Number of contained files and subdirectories

  – Flags indicating the status of the node

  .

- **ELF_FILE** An "ELF_FILE" node includes the metadata necessary for operating on a file. As in traditional UNIX-like file systems, file inodes are entirely distinct entities from directory entries in ELF. File content is stored after the metadata in the node. Similar to "ELF_DIR", an "ELF_FILE" node also stores the file's name, inode number, and flags reflecting the node status.

- **ELF_COMMON**

  An "ELF_COMMON" node normally represents a write modification, a deletion operation, or a renaming operation. For these operations, an "ELF_COMMON" node is created and appended to end of the list of physical nodes for a given file. For operations such as write modification, newly added data is carried in this physical node instead of rewriting over the original flash pages. An "ELF_COMMON" node contains metadata such as the start offset of the data in the file, the length of the carried data, the inode number of the file it belongs to, and the version information of the included data.

Each flash page contains at most **one** physical node that is located at the very beginning of the page.

4.1.2 Per-Page Metadata

On Berkeley mica2 motes, each page in flash contains 256 bytes for data, and an extra 8 bytes intended for per-page metadata. Figure 4 depicts the metadata structure stored in each flash page in ELF.

```
struct page_info{
        uint16_t crc;
        uint16_t nextPage : 11;
        uint16_t flags : 5;
        uint16_t writeCounter;
        uint16_t magicNumber;
 }
```

Figure 4: Per Page Metadata Structure in Flash

A simple CRC checksum is stored in the metadata to verify the data integrity of the page. ELF provides two different reliability configurations for the checksum calculation. The user can specify whether the CRC is the checksum of the whole page or only of the metadata, excluding the CRC field. The 11-bit nextPage field provides the pointer to the next data page belonging to the same physical node. For example, when appended data exceeds the first page's capacity, a second page will be allocated. The "nextPage" field in the first page's metadata will then be updated to point to this newly allocated page. In this approach, allocated flash pages gradually form a linked list, as is shown in Figure 3. The 5-bit flag indicates the status of the page, such as free, in_use or dirty. The field "writeCounter" indicates how many times this page has been rewritten, which provides age information for implementing wear-levelling techniques. The final 16-bit field is reserved for future use. It is currently serving the purpose of a magic number to identify the ELF system.

4.2 In-Memory Data Structures

Data structures are created in RAM for run-time support of operations on opened files. Since a file in ELF consists of a list of physical nodes, the memory representation of an opened file includes two parts: a file abstraction and a list of physical node abstractions. Figure 5 shows the file and physical node abstraction data structures in RAM. The relationship between these two abstractions is illustrated in Figure 3.

```
struct elf_node_abstract{
       uint16_t version;
       uint32_t beginOffset;
       uint32_t locaLen;
       uint16_t startPg : 11;
       uint16_t flag : 5;
      struct elf_node_abstract *next;
}__attribute__((packed));
typedef struct elf_node_abstract elf_node_abstrac_t;


  struct elf_file_abstract{
         uint16_t version;
         uint16_t inodeNumber;
         uint32_t totalLen;
         uint8_t flag;
         uint8_t writeCounter : 4;
         uint8_t readCounter : 4;
         uint16_t pInode;
         elf_node_abstract_t *first;
   }__attribute__((packed));
```

Figure 5: File and Node Abstractions in RAM

4.2.1 *File Abstraction*

A file abstraction stores the metadata of the file, such as the length of the file, number of opened handles and a pointer to the list of physical node abstractions. Modifications to the file are reflected in the file abstraction as they occur. Each file abstraction can be associated with a small write cache in order to improve the write performance and extend the flash memory lifespan by consolidating writes to the same page. The size of the data cache is set to 64 bytes by default, but is configurable at compile time. As it is not expected to have a large number of files opened simultaneously on a sensor platform, ELF only allows a limited number of files to be opened at the same time. ELF defines a virtual file abstraction as the root of the whole file system at the start of the system.

4.2.2 *Node Abstraction*

Each physical node of the file on flash has a corresponding abstraction in RAM. The abstraction only contains the necessary summary information of the physical node in order to keep the memory footprint to a minimum. Each physical node represents a portion of the file contents, and also contains the pointer to the next physical node abstraction in RAM.

4.2.3 *File Descriptor*

To operate on a file, an application first needs a file descriptor. A file descriptor in ELF consists of the file abstraction, current offset in that file and the open mode.

4.3 In-EEPROM Data Structures and Related Operations

ELF may also exploit EEPROM-style memory where available. ELF caches its directory in EEPROM for fast access to files. ELF also stores a snapshot of the system in EEPROM as a means to enable file consistency. These allow fast startup after a graceful reboot and for crash recovery of reliable file operations. The EEPROM is separated into a checkpoint region and a system information region containing the directory and other data structures, as shown in Figure 3.

4.3.1 *System Reboot*

Normally, a micro sensor system rebooting procedure includes initial setup and scanning of flash memory to collect file information. The in-flash file data structures provide sufficient information to rebuild the entire directory hierarchy on system start. However, this rebuilding process requires a complete linear scan through all of flash memory, and is time-consuming. To improve booting performance, the entire directory structure and file representations in RAM, are stored in EEPROM before a graceful reboot. Thus, a system can quickly read the ELF directory structure from the EEPROM instead of scanning all flash pages. This shortens the reboot time. In addition, if there is a non-graceful involuntary system reboot, e.g. a crash, then the non-volatility of EEPROM can be used to store checkpoint information to improve recovery for reliable files after a crash.

4.4 File Operations in ELF

ELF provides standard file functions such as *open()*, *close()*, *read()*, *write()*, *lseek()*, *delete()* and *truncate()* and directory operations such as *mkdir()* and *rmdir()* similar to traditional unix file systems. ELF also supports both random read and write access, which will be addressed in detail in a later section. We illustrate examples of typical ELF file operations in the following subsections on a file called "foo".

4.4.1 *Open and Create*

To open "foo", ELF will first examine whether "foo"'s file abstraction is already loaded into RAM. If "foo"'s file abstraction is not found, the system will construct the file abstraction from the system directory snapshot, which is currently stored in EEPROM. If a file is to be "created", ELF will allocate a free flash page to the file and create the "ELF_FILE" physical node on it. ELF will then create the file abstraction and physical node abstraction in RAM. ELF only keeps a limited number of file-abstractions in RAM. If all slots for opened files are occupied, the "open"/"create" function will return an error. Otherwise, a file descriptor will be assigned to the application by the system. This file descriptor includes a pointer to "foo"'s file abstraction, an 8-bit flag indicating the open modes, and the offset in the file. The result of opening a file on RAM is shown in Figure 6(a). ELF also allows clients to specify that a file

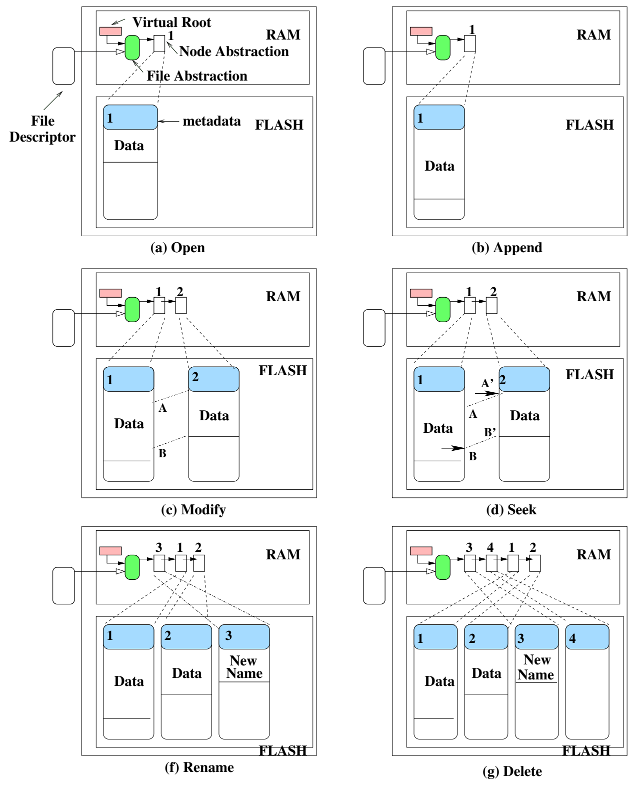

Figure 6: File Operations

act as a sized circular buffer. Append operations that would exceed the specified circular buffer size instead overwrite the bytes at the beginning of the file. This mode is intended to ease management of sample storage.

4.4.2 *Append*

After opening ”foo”, we then append data to it. ELF employs a policy of delaying updates to a file’s physical node on the flash given a sequence of appends. Appends to the same file are thus first cached in the assigned buffer in RAM in order to reduce the wear on a given page. When the buffer is full, the contents of the buffer are written to the last flash page in the file, which contains the tail end of the sensor data. Figure 6(b) illustrates the results of an append in flash and RAM. If this writing exceeds the available space on the flash page, then a new flash page is allocated. The metadata on the old page is updated, namely the nextPage pointer, to point to the new flash page. In this way, Write-Appends create a linked list of flash pages containing “foo”’s data.

One aspect of the append operation that has yet to be described is the updating of the file inode, e.g. to update file length. As is stated before, ELF uses assigned RAM buffer to aggregate small writes in order not to wear out the page containing the file’s inode and improve the write performance. However, the file abstraction and physical node abstraction in RAM is updated in real time. This approach leads to temporary inconsistency between the flash and RAM representations of the file. ELF’s design trades off the requirement for file consistency for an improved lifespan in the flash based file system. As a means of further improving wear levelling, ELF copies the page containing the file inode to a free page when actual updates on the file inode happen. In this case, a new version number is assigned. Otherwise, during a more typical append operation, neither the version number of the physical node nor the file’s version number are updated.

4.4.3 *Write(modify)*

Suppose now that a user desires to modify an existing portion of the open file “foo”. The Write-Modify operation is treated differently in ELF than a Write-Append. As shown in Figure 6(c), the offset of the data in ”foo” begins at A and ends at B. ELF checks the offset A and finds that this offset overlaps with existing data. ELF then allocates a new flash page and creates a new physical node. Corresponding to this new node on flash, a new node abstraction is created in RAM and appended to the physical node abstraction list associated with ”foo”. The file version number field is then increased by one. If there are a series of Write-Modifies, then each Write-Modify adds a physical node to the log in flash and RAM, and the physical node abstraction list of the file grows linearly. This is in contrast to appends, which do not normally add a new physical node.

4.4.4 *Read/Seek*

Reading from a file is complicated by the log of multiple physical nodes in a file. If the log is long, i.e. there are many Write-Modifies, then the performance of the read operation will degrade. However, it is not expected that there will be many modifications on sensor logs in the common case. For a random read operation, two pointers are used, as a file might consist of multiple nodes. A read operation may consist of several sub-read operations on each physical node. When reading each physical node, the first pointer points to the start position while the second one points to the end position of this sub-read. For example, consider the read example shown in Figure 6(d). To read through all the “foo” file, a read operation should first get bytes from offset 0 to A in node 1, then read bytes A' through B' in node 2, and then return to node 1 to finish reading the file. Node 2 must be accessed because the data between A and B in node 1 is obsoleted by node 2. The read operation will then go back to node 1 to read data from B till the end of file. Corresponding to the above description, the first pointer will first be set to 0 while the second pointer is set to position A. Thus, A bytes are read in this sub-read operation. Then ELF will set the first pointer to offset A' and the second pointer to B' subsequently to read $B' - A'$ bytes. Finally, the first pointer is moved back to point to B position in node 1 while the pointer 2 is set to the end of node 1, which is end of the file. The next sub-read operation reads all $EOF - B$ bytes from offset B. The performance of this operation degrades as the number of overlapped physical common nodes in the file increases. A seek operation is similar to the read operation except that it doesn’t actually read the data but simply finds the offsets.

4.4.5 *Rename*

Renaming a file in ELF results in creating a new physical node as well as its abstraction. There is no modification to the original physical node. Renaming a directory in ELF is different. To rename a directory, a new directory node is allocated. All data structures within the old directory node are copied to this new dir_node. The name is changed in the new node. The original directory node is then set to be deprecated.

4.4.6 *Deletion*

Similar to the rename operation, the delete operation is also achieved by adding a physical node to indicate the delete status instead of real removal. The allocated space will be reclaimed during garbage collection.

4.4.7 Truncation

ELF provides a ”truncate” operation. In contrast to the traditional unix ”truncation” command, ”truncate” in ELF allows an application to delete data from the *beginning* of the file. This is intended to aid the management of sensor data, allowing an application to dynamically free segments of the oldest data, which usually resides at the beginning of the log file. For example, as data samples are relayed back to a base station their space on flash may be reclaimed. Also, in sensor networks the latest sensor readings are generally more important than the outdated data. The truncation function is currently under implementation and will be included in the release version.

4.5 Resource Management and Garbage Collection

4.5.1 Resource Management

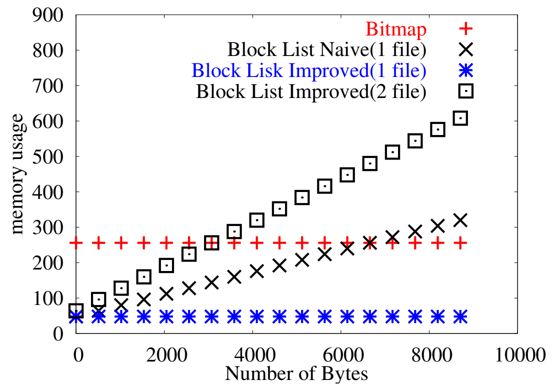

Figure 7: Flash Memory Resource Management

Traditionally, free/dirty/used block *lists* are used in the file system to manage the storage space. For the environment in which ELF operates though, a *bitmap* is a better fit. Figure 7 illustrates the comparison between the bitmap approach and the traditional block-list approach. The size of the bitmap needed to keep track of all pages of size 256 bytes in 512 KB of flash amounts to 256 bytes. The size of the block list needed to keep track of blocks of size 256 bytes, assuming 16 bytes of metadata associated with each allocated block, varies depending on the number of allocated blocks. We conducted experiments based on three different settings. In the first setting, there is only one file in the flash. This file is created before the experiment. Data is then continuously appended to the end until the flash memory is used up. The figure shows that the memory usage of the block-list based scheme exceeds the bitmap approach when around 6500 bytes are allocated. Since the size of the flash memory is 512 KB, then the memory consumption of the list-based approach is not acceptable.

In order to further investigate different resource management schemes, we made some optimizations for the list-based approach. With this optimization, the newly allocated blocks are combined with existing blocks if possible. For a single file, the performance shows significant reduction in memory consumption, albeit at the cost of additional processing. However, when there are two or more files opened simultaneously for writing, this optimization does not perform as well. Figure 7 shows that this improved scheme consumes more memory than a bitmap when fewer than 4000 bytes are allocated.

It is possible to utilize other optimizations to improve the linked list performance at the cost of greater management overhead. For example, each block can be increased to include several memory pages. This will reduce memory usage in a block-list based approach. However, such a scheme suffers from space inefficiency. Also, a block larger than a page increases the difficulty in providing an effective wear-levelling mechanism.

We chose the bitmap as the resource management scheme for ELF for its efficiency and simplicity. A bitmap based approach does not require sophisticated mechanisms, yet at the same time allows allocation of data blocks on a per page basis. Furthermore, by storing portions of the bitmap into persistent storage, such as internal EEPROM, the RAM resident space requirement of the bitmap can be decreased. Another reason for choosing a bitmap is its fixed memory size. A linked list based approach requires dynamic allocation of buffers. At present, TinyOS is only capable of static memory allocation in order to avoid memory overflow. [31]

4.5.2 Garbage Collection

For most log-based file systems, a garbage collection mechanism is important for efficient operation. In ELF, only a simple garbage collection mechanism is implemented. The number of free pages is tracked. Whenever the number of free pages drops below a preset threshold, the “cleaner” task is posted. As discussed in prior work [20], implementing a cleaner in user space allows for flexibly changing or adding a cleaning policy or algorithm. For event-driven embedded systems such as TinyOS, there is no such difference between user space and kernel space. During execution, the cleaner reclaims the pages occupied by deleted, renamed or obsolete files. The cleaner traces the link inside the file pages and sets the corresponding bits in the bitmap. If there is still not enough free space after scavenging, then files can be de-fragmented to reclaim more pages. The cleaner also eliminates pages whose write lifetime has been exceeded. In ELF, any page with more than 9000 writes will be marked as unusable, assuming a lifetime of 10000 writes. In contrast to the relatively memory-intensive segment cleaning in LFS, a ”cleaner” task in ELF simply reads the flash page metadata and flips the corresponding bits in the bitmap. There is no data copied into the RAM. Thus, the RAM consumption for the ELF cleaner is fairly low: typically fewer than 10 bytes without defragmentation, and fewer than 40 bytes when defragmentation is required.

4.6 Crash Recovery

Since certain files require reliable file operations, e.g. binary images and configuration settings, ELF provides optional crash recovery. ELF allows the user to specify whether a given file should use crash recovery. ELF uses a two-pronged approach to recovery: checkpoints, which define consistent states of a file, and roll-forward, which is used to recover as much data as possible. A checkpoint is normally defined as a position in the log at which all file system structures are consistent and complete. However, a checkpoint does *not* provide such guarantees in ELF. An ELF checkpoint only stores the status of a file. In order to roll-forward to the error free record, ELF tracks both the current operation and a list of past operations on a reliable file. Each operation must be documented in addition to the modified data.

A crash can happen when updating either the metadata or the data contained in the file. It is useful to find the failure point. Calculating the checksum can detect whether the data in a flash page is damaged, but does not reveal any further information. A traditional LFS scheme is insufficient for a flash based file system. In flash, even if a checkpoint is committed in the middle of a page, the write failure *after* the checkpoint in the same page will still result in loss of information. For example, ELF always appends data to the end of the file instead of creating a new node. When such an append fails, all data contained in the same page as the failure point will be seen as invalid. Thus the checkpoint fails to achieve consistency.

ELF implements a best effort crash recovery mechanism for reliable files. For write operations, ELF records a number of snapshots of recent actions. It also keeps a snapshot of the current action. These snapshots are represented by a tuple

$$\{inodeNum, action, cur\_version, highest\_version\}$$

An action may be one of the following:

- Append: includes the starting offset and length of the appended data
- Modify: includes the starting offset and length of modified data
- Rename: includes a version number
- Obsolete: includes a version number

The typical length of a tuple is about 15 bytes. For each operation on a reliable file, ELF needs to write data to the file, modify the RAM representation, update metadata and then store the snapshot in the EEPROM. Before an operation occurs, a snapshot of this action is recorded in the EEPROM as the current operation. If the system crashes during this action, ELF compares the last finished action and the current action during the recovery stage. If they match and the CRC checksum is correct, this file will be treated as a file with confirmed integrity. Otherwise, ELF will roll back checkpoints, attempting to find a checked operation happening in a page different from the current operation. If there is no such snapshot, the file will be considered damaged and marked obsolete.

5. PERFORMANCE EVALUATION

ELF is implemented on TinyOS, which runs on mica2 motes [2]. In this section, we will first evaluate the performance of common file operations in ELF. Then we present ELF's performance with respect to wear levelling and reliability. We also compare ELF with Matchbox since Matchbox is currently the only openly available sensor file system for the motes.

5.1 Sequential Read Performance

The goal of the sequential read performance test was to measure the maximum read throughput for the most common files: sensor data logs. We expect the majority of the read operations of sensor files to be sequential reads since the content of a sensor file is likely to be read sequentially and transmitted through the radio for relay to the base station. In this experiment, we first filled the entire memory with a 500 KB log file and then performed sequential reads on this sensor log file. We gradually increase the number of bytes read each time and compare the performance between ELF and Matchbox. Each run is repeated fifty times, and the average result is plotted in the graph.

Figure 8 illustrates the sequential read performance of Matchbox and ELF on a large continuous file. For sequential reading, the performance of both ELF and Matchbox are stable. In general, ELF achieves better performance than Matchbox by about 5 KB/second. However, this is due to a device driver implementation detail. The driver used in Matchbox will first data into the cache on the flash chip. It then reads the data from this internal cache. ELF improves performance by reading data directly from the flash memory. This reduces the latency for reading data from each page by about 250 microseconds.

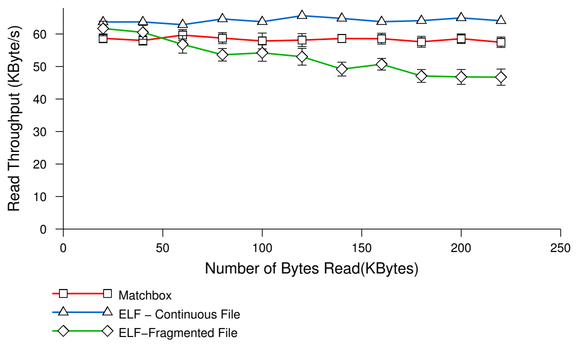

Figure 8: Sequential read performance

5.2 Sequential Write Performance

Write-Append is another major file operation in ELF expected to be used for sensor data. The size of each sensor reading is normally one or two bytes. For example, the Mica weather board used on Great Duck Island for habitat monitoring [9] contains seven different sensors: a photo-resistor, I^2C temperature sensor, barometric pressure, humidity, thermopile and thermistor. Each sensor sample is between 10-16 bits. Even if all sensors are read simultaneously during a sampling period, the total data size is only $7 \times 2 = 14$ bytes.

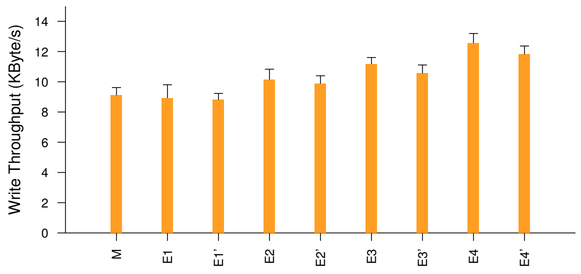

Figure 9: Sequential Write Throughput

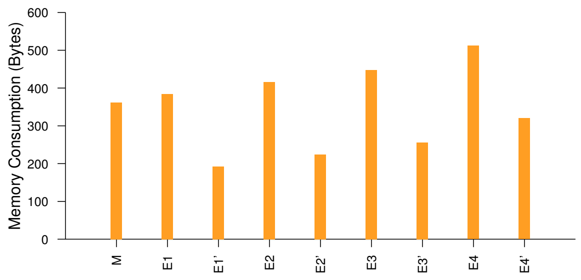

Figure 10: Sequential Write Memory Consumption

In order to examine the performance of ELF's log appending operations, a file is first created in the flash memory. Then 16 bytes of sensor data are appended to the file in each write until the whole flash is consumed, making a total of 512KB of data. Figure 9 plots the average write throughput while Figure 10 presents the memory consumption. Experiment M corresponds to the maximum write throughput or memory consumption of the Matchbox system.

In our experiments, several different ELF file system configurations are examined by varying the amount of RAM used. Experiment $E1$ corresponds to the append performance of ELF without RAM buffering of append operations, and with the entire free space bitmap stored in RAM (256 bytes). Experiment $E1'$ is identical to $E1$ except that only a quarter of the bitmap, 64 bytes, is resident in RAM, with the whole bitmap being stored in EEPROM. After all pages in the 64 bytes have been allocated, the next 64-byte portion of the bitmap is loaded into the RAM. We then varied the amount of buffering by ELF of appends in RAM: $E1$ and $E1'$ have 0 bytes of buffering; $E2$ and $E2'$ have 32 bytes of buffer; $E3$ and $E3'$ have 64 bytes; and $E4$ and $E4'$ have 128 bytes in each trial. The experiment is repeated fifty times for each different setting and the average is shown.

When there is no buffer assigned to ELF, both ELF $E1$ and Matchbox M exhibit similar performance in these raw write operations. This is because ELF falls in the same category as Matchbox by directly appending the data to the end of the file in the flash memory. Matchbox writes at about 9.21 KB/second while the throughput of ELF is 9.03 KB/sec for $E1$.

Comparing variation across bitmap storage in RAM, $E1$'s 9.03 KB/sec is marginally higher than the 8.86 KB/sec throughput of $E1'$. This is because in $E1'$ ELF needs to write the used up bitmap in memory back to EEPROM and read in the next 64 bytes of the whole bitmap. This swapping time affects the performance when a large continuous sequence of writes occurs. This difference in write throughput is reflected for all cases. However, the memory consumption savings of the quarter-bitmap approach of $E1'$ compared to $E1$ is significant, as shown in Figure 10.

As the data buffer size used to aggregate small writes increases, the performance of ELF improves significantly, and exceeds the performance of Matchbox. This is because a Write-Append involves several different steps. The target page is first loaded into the internal cache in flash chip in order to prevent data loss. By aggregating multiple small writes, ELF avoids loading the page repeatedly, which also includes time for erasing and rewriting. A drawback of this scheme is reduced reliability. A power failure occuring before the buffer is dumped to the flash memory results in a loss of the data in the buffer.

5.3 Random Read Performance

Random read access is designed to address the case when there is a request to read a portion of the data in the middle of the log. For example, in an application such as TinyDB, a query may be sent out from the base station to the sensor nodes to retrieve sensor data collected during certain time periods. Another such case is dynamic reprogramming. Allowing nodes to retransmit specific portions of a program image requires random reads from a file and receiving packets (potentially out of order due to dropped packets) requires random writes. This experiment is designed to evaluate the performance of ELF in such cases. Figure 11 illustrates the performance of ELF in two different scenarios. In the first case, there is a large continuous log file stored in the flash. This file is composed of a single physical node. In the second case, four modifications have been applied to this file, which results in the creation of a fragmented file with an additional four physical nodes. The experiment is repeated fifty times with different randomly generated fragmented files created each time. The averages are plotted. In each experiment, the file pointer is set to several different offsets. The number of bytes read from each offset is fixed at 20 KB.

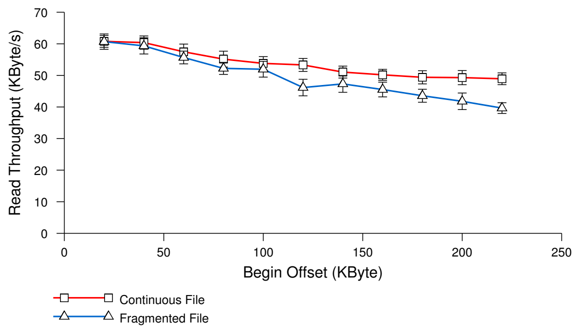

Figure 11: Random Read Performance

As is illustrated in Figure 11, the performance of random reads in ELF is worse for the fragmented file than for the continuous file. This is because a read might run across several different physical nodes. Matchbox is not shown for comparison because it does not support random reads. For the case of the fragmented file, large offsets can again span several different physical nodes. ELF's read performance also decreases modestly when the data offset increases because the seek time is also counted.

5.4 Random Write Performance

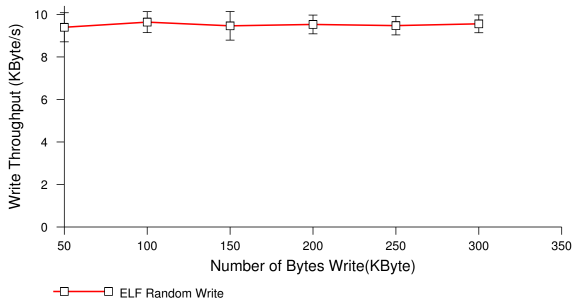

Figure 12: Random write performance

This experiment is designed to evaluate the case when there is a request to overwrite existing data in a file, *i.e.* a Write-Modify operation. For example, dynamic reprogramming might require the update of a local configuration file. In this experiment, we first write a 100 KB log file to the flash. After its creation, a portion of the file is randomly selected and overwritten with $0 - 300$KB of new data. No data buffer is used in this experiment. Figure 12 illustrates the random write throughput as a function of cumulative written data. As expected, ELF's performance for overwriting or modifying data doesn't significantly vary depending on the amount of overwritten data. This is because each write results in the creation of a new physical node after a simple comparison between the begin offset of new data and the file length.

5.5 Wear Levelling Performance

We use a simulator to examine the wear levelling performance of ELF. Wear levelling is a primary goal of an effective flash file system. In our experiment, four files are created:

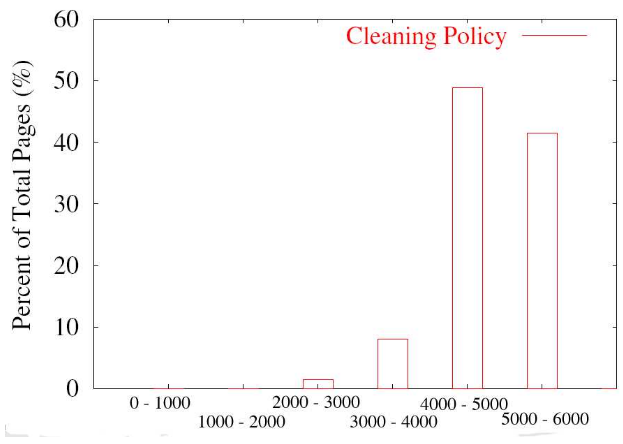

Figure 13: Wear Levelling Performance

- One 4 KB file is created and rarely updated. This represents sensor node configuration information.
- Two log files are created to evaluate wear levelling when logging data from two different sensors. 64 bytes of data are appended to each file alternately.
- For every 10,000 writes, a file of 20KB is created and then erased. This is designed to simulate basic dynamic reprogramming behavior.

Whenever the flash memory is full, one of the sensor logs is randomly deleted. The occupied space is reclaimed by the cleaner. A new sensor log file will be created after the erasure.

This experiment continues running until the total number of erasures exceeds 10,240,000. Each of the 2000 256-byte pages in the 512 KB flash are expected to be written exactly 5000 times if absolutely uniform wear levelling is achieved. Figure 13 shows a binned distribution function of the wear on each page following this experiment. ELF manages to achieve close to uniform wear levelling. Most pages' write counts are close to 5000. The number of pages with a write count fewer than 4000 is less than 10% of the total pages. We conclude that ELF's proposed mechanisms, though simple, achieve reasonable wear levelling. The pages that were significantly younger than the other pages belonged to the simulated sensor configuration files. Since these files are unlikely to be touched in a long time, there is little opportunity for wear levelling.

5.6 Energy Consumption

Here we summarize energy consumption in ELF. The major approach followed by ELF is through write aggregation. According to Table 1, it takes 4-10 mA to read the flash and 15-35 mA to program the flash. A whole flash page is erased and rewritten even for 1-byte change. Thus, aggregating small writes on the same page will help to reduce the energy consumed. This can be illustrated by a simple calculation. Assuming 16 bytes are appended to a log file each time in logging applications, it takes 16 writes to fill a 256-byte flash page. By allocating a 32-byte buffer to cache the writes, the power consumption can be roughly cut by half since there will be only 8 writes. Table 2 provides the detail for the EEPROM power consumption. In ELF, EEPROM read/write mainly takes place at the start/shutdown of the system and at the read/write operation on reliable files such as a program image. As these are not frequent practices for sensor networks, the operations on EEPROM are not expected to be a major source of energy consumption.

6. DISCUSSION AND FUTURE WORK

ELF seeks to meet the need for a more advanced file system for sensor networks. Several previous works on distributed storage in sensor networks have employed the circular buffer as the lowest level medium management approach in a distributed storage architecture. [32, 33]. With the evolution of sensor networks, this simple management technique is no longer able to satisfy the needs of various applications, e.g. the Mate virtual machine. Moreover, applications such as TinyDB also require more flexible and reliable low level storage systems. ELF is designed to meet this emerging desire for complexity. The ELF file system provides a complete file system uses atomic write operations for operating with file metadata, and provides support for simple garbage collection and best effort crash recovery. ELF also provides convenient APIs similar to the unix system call interface for applications.

ELF is designed with efficiency and sensor network characteristics in mind. Although it's quite different from the traditional LFS in both design and motivation, ELF is still a log-structured file system in the sense that new data is always written to new space, which achieves wear levelling for sensor flash-based systems. For sensor database-like applications, there will be the need to retrieve data from the middle of the file in facing a query. ELF enables file read random access for such applications. ELF also allows the creation of a simple directory in order to preserve the flexibility of categorizing files. Considering the characteristics of sensor networks, ELF is not designed to handle the frequent occurrence of operations such as rebooting, file opening and creation. These issues will be addressed in the future if they prove desirable.

Buffering is important to both the performance and efficiency of ELF. The performance evaluation section has illustrated the improvement of write throughput by using modest buffers to aggregate writes. Furthermore, caching the writes also increases the flash's memory lifetime and saves energy by effectively reducing the number of actual writes to the physical flash medium. However, such performance enhancement is obtained at the cost of the loss of reliability, since a system crash before the actual write operations will result in the data loss of the cached writes.

A more detailed study of the reliability of ELF to a host of failure conditions is needed. At present, our study of ELF's resilience to crashes has been rudimentary. Further study is needed to test the capability of ELF to withstand failure at any point in the open, read, write, close process, in addition to garbage collection cleaning. This paper also has not explored the energy cost or latency cost of ensuring reliability for a subset of the files on a micro sensor node.

Data compression, although incurring a computational cost, can be very useful to the deployed sensor network since it effectively increases the storage capacity on each micro sensor node. Typical lossless data compression techniques include Huffman, arithmetic encodings, the Lempel-Ziv coder, etc. Studies shows that these techniques could increase the space usage by 2-4 times. [9] A future direction of ELF includes implementing a suitable compression algorithm for sensor data in order to increase the storage capacity on flash.

7. CONCLUSION

ELF is a full, reliable and efficient file system designed and implemented for micro sensor nodes such as the mote. ELF is tailored to the resource constraints of sensor nodes, e.g. limited RAM and energy. ELF is also tailored to the behavior of sensor networks, i.e. ELF is optimized for write-append operations characteristic of logging sensor data to a file. ELF is further adapted to the storage medium of choice on micro sensor nodes, namely non-volatile flash memory. In particular, the log-structured nature of ELF enables wear-levelling across flash, so that writes are evenly distributed across flash pages with limited write lifetimes. ELF also meets a variety of other sensor networking needs, such as crash recovery for reliable configuration files and binary images. To our best knowledge, ELF is the first log-structured file system for sensor networks that provides garbage collection and a best effort recovery mechanism.

8. ACKNOWLEDGMENTS

Many thanks to Charles Morrey, Brian Shucker, Anmol Sheth and Jing Deng for their suggestions concerning ELF.

9. REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, Aug 2002.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *ACM Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.

[3] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Mobile Computing and Networking*, pages 32–43, 2000.

[4] N. B. Priyantha, A. K. L. Miu, H. Balakrishnan, and S. J. Teller. The cricket compass for context-aware mobile applications. In *Mobile Computing and Networking*, pages 1–14, 2001.

[5] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[6] The smart-its project, http://www.smart-its.org/.

[7] The eyes project, http://eyes.eu.org/.

[8] M. Leopold, M. Dydensborg, and P. Bonnet. Bluetooth and sensor networks: a reality check. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 103–113. ACM Press, 2003.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, Atlanta, GA, September 2002.

[10] James reserve extensible sensing system, http://www.cens.ucla.edu/ eoster/tinydiff/.

[11] New computing frontiers - the wireless vineyard, http://www.intel.com/labs/features/rs01031.htm.

[12] Crossbow motes, http://www.xbow.com/.

[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI Conference*, December 2002.

[14] S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. 2002.

[15] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[16] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA*, pages 60–67. ACM Press, 2003.

[17] Crossbow in network programming, http://www.xbow.com/support/support_pdf_files/motetraining/xnp.pdf.

[18] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[19] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15. ACM Press, 1991.

[20] M. I. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[21] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *ASPLOS*, pages 86–97. ACM Press, 1994.

[22] Flash memory, intel corporation, 1994.

[23] F. Douglis, R. Caceres, M. Frans Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Proceedings of the First Symposium on Operating Design and Implementation (OSDI)*, November 1994.

[24] D. Woodhouse. Jffs : The journalling flash file system.

[25] S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, S. Shenker, L. Yin, and F. Yu. Data-centric storage in sensornets. 2002.

[26] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob. Netw. Appl.*, 8(4):427–442, 2003.

[27] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution storage for sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 89–102. ACM Press, 2003.

[28] D. Ganesan, D. Estrin, and J. Heidemann. Dimensions: Why do we need a new data handling architecture for sensor networks. In *First Workshop on Hot Topics in Networks (Hotnets-I)*, 2002.

[29] 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 sensornets. In *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[30] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to network embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[31] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI*, 2004.

[32] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[33] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution storage for sensor networks. In *Proceedings of the ACM SenSys Conference*, pages 89–102, Los Angeles, California, USA, November 2003. ACM.