Enabling Large-Scale Storage in Sensor Networks with the Coffee File System

Nicolas Tsiftes, Adam Dunkels, Zhitao He, Thiemo Voigt
Swedish Institute of Computer Science
{nvt,adam,zhitao,thiemo}@sics.se

ABSTRACT

Persistent storage offers multiple advantages for sensor networks, yet the available storage systems have been unwieldy because of their complexity and device-specific designs. We present the Coffee file system for flash-based sensor devices. Coffee provides a programming interface for building efficient and portable storage abstractions. Unlike previous flash file systems, Coffee uses a small and constant RAM footprint per file, making it scale elegantly with workloads consisting of large files or many files. In addition, the performance overhead of Coffee is low: the throughput is at least 92% of the achievable direct flash driver throughput. We show that network layer components such as routing tables and packet queues can be implemented on top of Coffee, leading to increased performance and reduced memory requirements for routing and transport protocols.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Storage Management; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Sensor networks, storage-centric, file systems, storage abstractions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IPSN'09, April 13–16, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-371-6/09/04 ...$5.00.

1. INTRODUCTION

An emerging class of sensor networks uses flash memory for sensor data logging [3], object databases [15], software modules [4], and as a virtual memory backend [10]. Existing storage systems for sensor networks typically access the flash memory directly and ad hoc, or they use too much RAM to handle large flash memories. A common storage layer ensures that boundaries between files are protected, that it is easy to switch to other storage devices, and that difficult device semantics are hidden from the programmer. Storage abstractions become more portable and simpler to develop when the focus can be on the higher level design, instead of on the low level flash memory management.

Log-structuring [23] has gained foothold as the most common flash file system design. Flash memory has a physical restriction that no data can be overwritten without erasing a large amount of data first. By recording the file changes in log records instead of writing them in place, the log is a natural fit for flash memory semantics. Although a log-structured design is typically memory intensive, the sensornet community has learned how to adapt it to small sensor devices. Still, existing implementations have high memory and code complexity because a considerable share of the file metadata must be cached to gain an acceptable performance. Since in-RAM metadata typically grows linearly [2] with the file sizes, this technique is unwieldy for coexistence with other complex system components.

We design the Coffee File System to meet the need for a generic, high-speed, flash-based file system that is feasible for a wide range of sensor devices. In contrast with earlier fully functional flash file systems, Coffee has a constant RAM footprint per open file. A memory complexity of $O(1)$ per file is indeed particularly important for sensor devices, where the non-volatile storage can be several orders of magnitude larger than the RAM. Coffee serves as a thin file system layer that facilitates platform-independent storage abstractions through an expressive programming interface. In its default setup, Coffee requires 5kb ROM for the code, and 0.5kb RAM

at runtime. Although our premier intention is to reduce the complexity of storage systems, the file system is also efficient to support demanding storage-centric sensor applications and networking components. The performance of the most used file operations–read and append–is over 92% of the direct flash driver speed.

Conventional log-structured flash file systems force all files to share a page-based log that spans over the flash memory. Thus, their performance and use of flash space is far from optimal when modifications are scattered and smaller than the page size. Like log-structured flash file systems, Coffee stores new data in a log, yet our design is a significant departure from the conventional method. Append-only files are stored in the simplest way as contiguous group of pages. Once a file is modified, Coffee creates an accompanying micro log structure and links it with the file. Prior to creating the log, the software using the file may configure the log size and the log record granularity. Abstract storage implementations can fine-tune their micro logs according to the expected access pattern to increase the performance significantly. As evidenced by our experiments with micro logging, we have enabled a performance optimization unavailable in previous flash file systems, while reducing the memory complexity to $O(1)$ per open file.

Our scientific contributions are threefold. First, we show that a structured node-local storage system can provide a generic feature set using $O(1)$ RAM per file. Second, we introduce micro logs and quantify the effects of tuning the parameters for arbitrary workloads. Third, we evaluate the Coffee file system in a networking perspective by implementing storage abstractions for the networking stack. We show that Coffee's high throughput makes it a suitable application-managed virtual memory back-end even for intensive workloads.

The paper proceeds as follows. We introduce flash memories and their uses in storage-centric sensor networks in Section 2. In Section 3 we present the design of Coffee and describe the algorithms beneath the programming interface. Thereafter we discuss implementation aspects in Section 4. We divide the experimental evaluation into two parts: Section 5 quantifies Coffee's low-level operations, whereas Section 6 evaluates how Coffee performs in real applications by studying two storage abstractions for networking. After covering related work in Section 7, we conclude the paper in Section 8.

2. STORAGE IN SENSOR NETWORKS

The presence of on-board flash storage devices on state-of-the-art sensor network hardware platforms has inspired recent work in storage-centric sensor networks and sensor network virtual memory. The availability and low cost of flash memory devices have made them a popular choice for embedded systems. Platforms having on-board flash memory storage include the Tmote Sky [22], the Intel Mote, and the MicaZ. Flash memory is suitable for sensor networks because it is energy-efficient, shock-resistant, small, and cheap. Sensor nodes are typically equipped with flash memory in the range between 100 kb and 1 Mb, but non-volatile memories in the gigabyte class, such as SD cards, are emerging [25].

2.1 Storage Centricity

Wireless sensor networks were in the beginning viewed as communication-centric; the primary objective of wireless sensor networks was to communicate sensor data from the sensor nodes towards one or more base stations, possibly aggregating or processing the data within the network. Recently, however, the monetary cost and energy consumption of on-board storage has been reduced, leading to a reconsideration of the communication centric view of sensor networks. This development leads to the possibility that sensor networks should be viewed as storage-centric rather than communication-centric [3,7,15]. In a storage-centric sensor network, the primary objective of the network is to store the sensed data. The data can later be collected from the network when needed.

Recent work has shown that batching data may improve energy efficiency [7, 16]. By batching the sensed data instead of immediately sending it, we save more energy since the radio duty cycle can be significantly reduced. Several recent sensor network deployments use data batching; examples include volcano monitoring [26] and bridge health monitoring [12].

In storage-centric sensor networks sensed and collected data must be delay-tolerant. For delay-sensitive data, a communication-centric approach is better. Examples of delay-tolerant data are temperature logs, camera images, and structural health monitoring data. Examples of delay-sensitive data are intrusion detection data, fire alarms, and industrial control.

Storage-centric sensor networks require storage facilities on the nodes and a mechanism for retrieving the data from the nodes. In this paper, we focus on the node-level storage and note that protocols for batch data transfer exist [11].

2.2 Using Storage as Virtual Memory

There are several recent proposals for extending the limited RAM of sensor nodes by using the on-board flash memory. Examples that use the flash as a swap area include interpreted virtual machines [1], compile-time mechanisms [13], and run-time hybrid interpretation models such as the t-kernel [10].

Because memory accesses in a virtual memory setting may be frequent, using on-board storage for virtual

memory requires a fast underlying storage system. In the absence of this, virtual memory mechanisms generally access the on-board flash device directly. Accessing the flash device directly, though, requires that the application manages wear levelling, garbage collection, and space allocation. With this paper, we present a storage abstraction that provides these services while having a throughput close to what is achievable by directly using the underlying flash device. As shown through our experiments, mechanisms such as virtual memory can be implemented efficiently without requiring the application to manage wear levelling, garbage collection, and storage allocation.

2.3 Flash Memory Semantics

Flash memories have three characteristic operations: read, program, and erase. Unlike magnetic disks, a part of the flash memory must be erased before overwriting data. These parts are called erase sectors and are typically several kilobytes large. All bits in the erase sector are set to 1 initially, but a subset of these can be programmed to 0 in one operation. The read operation, in contrast, can be done an arbitrary amount of times, either on page or byte granularity depending on the flash memory type.

Flash memories are classified as NOR flash or NAND flash. NOR flash is generally memory-mappable and permits random access I/O, but the erase sectors can be large. NAND flash requires page-based I/O and is not memory-mappable, though the erase sectors can be considerably smaller than those of NOR flash. NAND flash memories are generally limited to sequential writes within an erase sector, but the erase sector is often significantly smaller than on a NOR flash memory of similar size.

2.4 Flash Semantics Affect File System Design

File system designs for magnetic disks are not applicable in sensor network hardware for two reasons. First, the memory constraints of sensor devices make it unfeasible to hold large buffer caches or metadata structures in RAM. Hence, the flash memory should store the majority of the metadata to suppress the effect having very little memory available. Second, as described above, flash memories have different write semantics than magnetic disks. Flash memory bits can be programmed with a low overhead, but resetting programmed bits requires an expensive sector erase. The erase takes time and causes sectors to wear out after erasing many times. Consequently, fine-grained modifications are more complex to handle when using flash memory.

Flash file systems usually follow the conventional log structure design to mitigate flash memory semantics. Essentially, the log structure is a a circular log where every record belongs to a certain file. Records of different files are intertwined, and are typically distinguished by a file id, a region in the file, and a record age. Log structures generally require linked lists of regions in RAM, since the performance would be degraded severely if the file would be traversed in flash. The conventional log structure does not scale well with the flash storage size, which can be several orders of magnitude larger than the RAM in sensor devices. Moreover, the garbage collection procedure is further affected by the overhead of moving active pages from the oldest sector towards the end of the log.

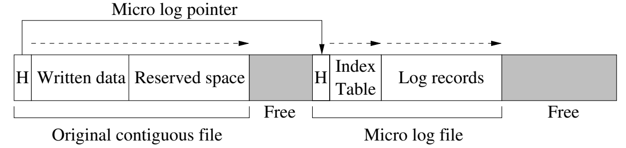

Figure 1: The file layout in the flash memory. Files consist of reserved consecutive pages and start with a header (H). Modified files are linked with a micro log file containing the most recent data.

3. COFFEE

Coffee is a portable, high-speed file system for sensor devices equipped with flash memories. While using a simplistic sequential page structure for each file, we introduce the concept of micro logs to handle file modifications without imposing a spanning log structure. Conventional log structures typically require considerable memory for caching file metadata. Micro logs allow us to configure the logs of individual files with a tunable trade-off between space and speed. In essence, Coffee has an extensive interface for high level storage abstractions, while using a small memory footprint: each open file uses $O(1)$ RAM. As sensor networks perform a vast range of sensing and networking tasks with different requirements, the storage layer should be optimizable for many kinds of abstractions.

3.1 Design Principles

The principles that we design Coffee on concern both flash memory semantics and relevant parameters for sensor network hardware and applications. The file system must firstly commensurate with the memory and code size constraints in sensor devices. The micro controllers of sensor devices typically have RAM in the range of 1-10 kb, and code memory in the range of 10-100 kb. Storage systems using large buffer caches and data structures to increase the speed of file and directory operations are unsuitable for general purpose use in sensor networks. It is desirable to have a small memory footprints–regardless of the file sizes.

The large number of sensor devices and storage types motivates a flexible design that does not rely on stringent assumptions of a specific flash memory device. Nevertheless, we assume that the storage device is not in violation of the following semantics, though Coffee adapts well to memory types with more relaxed rules.

1. The flash memory is divided into erase sectors of uniform size.
2. Erasing sets all bits in an erase sector to 1.
3. An erase sector consists of a set of pages which can be programmed and read in random order.
4. Programming switches a subset of the bits in a page from 1 to 0.

These conditions are satisfied by a large variety of non-volatile memories, including NOR flash, mixed semantics flash (e.g., AT45DB in the Mica2), EEPROM, and SD memory cards. Nevertheless, if the flash memory requires page-based I/O, Coffee must emulate random access I/O within a page. Because NAND requires sequential page writes within erase sectors, it is less suited for fast random writing, unless large metadata structures can be accommodated in RAM. Coffee provides a limited functionality without random writes in the subset of NAND flash memories whose sectors are considerably larger than their pages are.

Performance is also important; both in terms of latency and throughput. Long-running file operations may affect time-critical operations. For instance, the MAC protocol timing and scheduled sensor readings are negatively affected by slow file operations. Similarly, motes that store large quantities of data, such as camera images, require a file system with high-speed append operations. The sequential write trait of both periodic sensing applications and high-rate streaming applications motivates a file system that supports append operations with low execution and I/O overhead.

3.2 Page Structure

Coffee divides the flash memory into logical pages whose sizes typically match those of the underlying flash memory pages. A file is stored as a contiguous group of pages–much like file system extents. Extents are used in numerous file systems to reduce fragmentation, but we draw advantage from this structure to reduce the file system complexity instead. As illustrated in Figure 1, a file consists of a header, a data area, and possibly some free space up to the file boundary. File pages are allocated either implicitly by opening a new file, or explicitly by calling the file reservation function. The page allocation algorithm uses a first-fit policy.

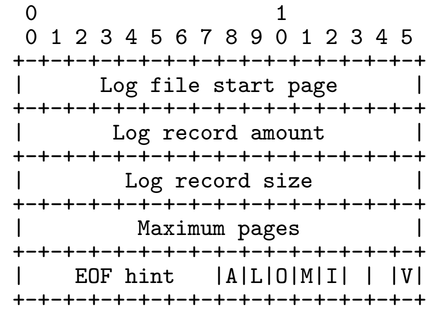

Figure 2: The file metadata contains micro log information and file status indicators.

Coffee allocates a pre-defined amount of pages if the file size is not known beforehand. Later on, if the reserved size turns out to be insufficient, Coffee creates a new larger file and copies the old file data into it. In our experience, however, the file size is often known. For instance, before accepting a file from a data dissemination protocol, an initial protocol message typically contains the file size. Another case is that an application may want to allocate a certain space to make sure that there is enough space to log sensor data, and leave the rest of the space for secondary uses.

3.3 Minimizing Metadata in the RAM

Because of the contiguous page structure, file metadata uses only a small and constant memory footprint for each file. Coffee stores metadata in a header in the first page of a file. To increase the performance of reopening files, we store a small metadata cache of 8 entries by default in RAM. The header is designed for flash semantics that restrict bits to one switch after each erase. Coffee inverts all read and written bits to have all data initialized to zero.

As depicted in Figure 2, the file header consists of seven fields. *log_page* points to the first page of the micro log. If the log is configured, *log_records* denotes the number of records that the log can hold, and it is multiplied with the *log_record_size* determine the log file size. If the log configuration fields are zero, then Coffee uses the default values for the hardware platform. The *max_pages* field specifies the amount of pages that have been reserved for the file. The *eof_hint* points to the part of the file containing the last written byte.

A page transits over three states: free, active, and obsolete. The flag field tells us the current state. The A flag denotes a file in use. Conversely, the current page and all remaining pages up to the next sector boundary are free if this flag is not set. When the file is deleted, the O flag is marked to indicate that the reserved pages are obsolete. The L flag shows that the file has been modified, and that a related log file exists.

The I flag identifies an isolated page. Isolated pages are processed one at a time by all Coffee algorithms, and are treated the same way as obsolete files. To discover garbled headers–typically caused by a system reboot during a header write operation–the V (valid) flag helps by marking that the header data is complete. Flags have a precedence order in which the valid flag is the most significant, followed by the isolated flag, the obsolete flag, the log flag, and lastly the allocated flag.

3.4 Locating Files

Directly after starting the system, Coffee is unaware of where files are located. Once a request is made to open a file, Coffee checks the file cache for its location. The file cache is filled with information from prior successful file search operations. A cache miss implies that we must scan the flash memory sequentially for the file.

As specified in Algorithm 1, Coffee uses a quick skip algorithm to find uncached files. The search algorithm reads one page at a time, but is typically able to skip many pages before reading the next page. If an obsolete file is encountered, the algorithm jumps over the number of pages that have been reserved for it. Otherwise, the algorithm checks whether the file is active and if the file name matches the name of the file being opened. If these conditions are fulfilled, we have found the first page of the file. If not, the reserved pages for the non-matching file are skipped.

Coffee accelerates the search further by skipping all pages of the current sector if a free page is encountered, starting from the page in question. The first fit page allocation algorithm guarantees that if a page is free, then the following pages in that sector are also free.

Algorithm 1 File search

Input: *name*, a null terminated string.
Output: The file's first page if found, -1 otherwise.

```
page ← 0
repeat
  hdr ← read_header(page)
  if page_allocated(hdr) ∧ ¬page_obsolete(hdr) then
    if filename(hdr) = name then
      return page
    else
      page ← page + max_pages(hdr)
    end if
  else if page_obsolete(hdr) then
    page ← page + max_pages(hdr)
  else
    page ← sector_to_page(next_sector(page))
  end if
until page ≥ MAX_PAGE
return −1
```

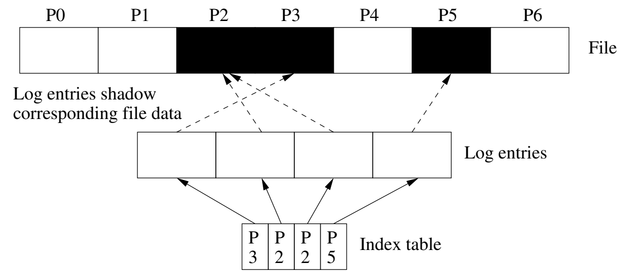

Figure 3: A modified Coffee file consists of data stored both in the original file and in a micro log. Writes that do not append data to the original file are added as records in the micro log files.

3.5 Determining the File Length

Since the length cannot be stored and updated in the metadata, we determine the length by using a hint of one byte in the file header. As soon as a file is closed, the hint byte is updated if the file length has increased. Each bit in the hint byte represents $\frac{1}{8}$ of the reserved file size. The least significant bit that is set determines where in the file that the last written byte exists. The indicated area is then scanned sequentially from the back toward the front to find the first byte whose bits are not all ones.

3.6 Tunable Micro Logs

Flash semantics require a more complex treatment of partial file overwrites compared to the treatment of file appends. Like most flash file systems, Coffee enables overwrites by logging the changes to files. To enable log optimization and at the same time reduce the complexity of the file system, we introduce a new approach to logging. Coffee creates micro log structures on demand and links them to files. The memory overhead of log structuring is thereby reduced to only two bytes per file descriptor that are used for storing an address to the last record in the log. Log files are created in the same manner as ordinary files, but they are invisible in the directory and inaccessible through the file system programming interface.

As shown in Figure 3, the log file contains an index table that consists of a fixed set of pointers representing each log record. The index key refers to a non-overlapping range in the file. This range starts at the file offset $pointer \times log_record_size$ and is as long as the specified record size. When reading from a file, the current file offset determines which index key to search for in the index table. The table is scanned from the last record towards the beginning. If no record is found, the file system concludes that the most recent data for that particular file offset range is located in the original file.

We exploit the individual micro log structures of files to introduce an interface for optimizing each micro log. This allows application programmers to set the log record size granularity and log file size according to the expected usage pattern of a specific file. Although this is limited to storage devices that permit variable block sizes for write operations, it reduces the space usage and the I/O time on the flash memories that have this capability. This is typical for NOR flash memories, such as the Tmote Sky's ST M25P80.

A series of file modifications eventually causes the log to fill up. This requires that the log and the original file are merged. Coffee carries out this operation by reading the file in chunks of log record size, and writing them to a new file without a log. Finally, the old file is deleted and thereby available for garbage collection. All file descriptors that pointed to the old file are redirected to the new file.

3.7 Garbage Collection

Coffee includes a garbage collector that reclaims obsolete pages when a file reservation request cannot be satisfied. We depict how the garbage collector iterates over all sectors in Algorithm 2. Page statistics are gathered for each sector to decide whether to erase the sector in question. The erase condition requires that the sector contains at least one obsolete page and no active pages. An obsolete file may span over more than one sector, which means that the sector status function must remember where the file ends in order to correctly obtain page states in subsequent sectors. Obsolete files that partly belong to a sector set for erasure and partly to a sector having active files must be split. The remaining pages in such a file become isolated.

Unlike log-structured flash file systems, Coffee erases any sector that satisfies the condition above, instead of just the least recently written sector. Active pages do not need to be copied to the end of the log structure. Furthermore, log-structured file systems must call the garbage collector every time the tail reaches the head, causing potential delays in the middle of file I/O. Coffee triggers the garbage collector only when creating files.

Irregular file creation and deletion patterns can induce a mixture between active and deleted files that makes a number of sectors irreclaimable for a long time. To alleviate this problem, a cleaning process periodically tries to move files from sectors having a majority of obsolete pages to sectors where most pages are free. Thereafter the obsoleted sector will be eligible for erasure. The cleaning process contributes to wear levelling by ensuring that files do not occupy sectors for too long. The cleaning process runs when removing a file, if a specified amount of time has passed since its previous sweep over the flash.

Algorithm 2 Garbage collection

```
sectors_in_row ← 0
longest_in_row ← 0
for all sector ∈ SECTORS do
  s ← get_sector_status(sector)
  if |active(s)| = 0 ∧ |obsolete(s)| > 0 then
    erase(sector)
    sectors_in_row ← sectors_in_row + 1
    if sectors_in_row > longest_row then
      longest_row ← sectors_in_row
    end if
  else
    sectors_in_row ← 0
  end if
end for
return longest_row
```

3.8 Wear Levelling Policy

Flash memory is prone to wear: every time a page is erased increases the chance of memory corruption. With current flash memories, a flash page is guaranteed to last for about 100,000 erasures. With an expected lifetime of 10 years, memory corruption may occur if pages are erased too often. The purpose of wear levelling is to spread sector erases evenly to minimize the risk of damaging some sectors much earlier than others.

Wear levelling is irrelevant for the main type of storage-centric application that simply logs sensor data. In this case, it is highly unlikely that the number of erase cycles per sectors will come close to the guaranteed amount of the vendor. In a traffic-intensive network, however, where routing nodes use packet queues, wear levelling is important if the flash memory must endure for years.

Coffee achieves automatic wear levelling as part of the garbage collection policy. Coffee delays garbage collection until a space reservation request cannot be fulfilled. Page allocations are thereby rotated over the full flash memory to a high degree, but sectors having old files will be exempted from garbage collection until a background process concatenates old files from different sectors into one file.

3.9 Fault Recovery

A sensor network storage system must be resistant to crashes because nodes generally have no memory protection that can isolate malfunctioning software. Malfunctioning software can thus influence the file system negatively if memory pointers are garbled. Moreover, watchdog timers may be enabled to restart a node if an operation does not terminate within a bounded time. Even when restarting during a file operation, the file system should prevent file inconsistencies after the system has been restarted. Furthermore, sensor measure-

ment data that has been collected for a long time must be easily extractable if the system fails.

The fault recovery mechanism in Coffee addresses the concerns above in two ways. First, sequential page allocations simplify file reconstruction since they do not rely on complex file structure that may be partially destroyed. Second, metadata updates are limited to one consecutive area of the flash in a file header, and therefore do not cause erroneous directory structures if they are stopped in the middle of their operation.

One of four different file modification operations can be active during a crash: a header modification, a log index entry addition, a log data addition, or a file append. Each operation is self-contained and sequential within its own designated area in a file. Consequently, a possible crash in the midst of a file operation will only affect the file in question, and the older contents of the file will be recoverable because of the sequential page structures in both original files and micro logs.

4. IMPLEMENTATION

We have implemented Coffee in the Contiki operating system [5] but the implementation is not Contiki-specific and can be ported to other systems and flash devices. The implementation is written in C and consists of approximately 1200 lines of code, including comments. To validate Coffee's portability, we have ported Coffee to an EEPROM device by changing fewer than ten configuration parameters and by changing the device-dependent implementations of the write, read, and erase abstractions used by the file system.

The file system programming interface conforms with the Contiki File System (CFS) API which provides a set of basic file and directory manipulation and access functions. As in the Unix world, files are accessed through file descriptors. Coffee's two most complex functions, garbage collection and log merging, are called implicitly, and the details are hidden from application developers. In addition, we augment the CFS API with two functions: coffee_reserve, and coffee_configure_log. The reserve function allocates space for a file before opening it for the first time. The coffee_configure_log function offers an optimization opportunity for the micro log file prior to creating it on demand.

Coffee enables directory listings using the cfs_opendir, cfs_readdir, and cfs_closedir functions. By iterating over the complete flash memory with cfs_readdir, which uses Coffee's skip algorithm, we obtain the name and size of each file. To simplify directory storage, the file system hierarchy is flat, but this is generally not a limitation in sensor devices since the number of files is typically small. Nevertheless, Coffee could be extended to support a deep hierarchy by using the conventional method of storing directory data in files.

Table 1: Micro benchmark for the Coffee operations

Operation	Avg. time (ms)	Std.dev.
open (uncached)	131.06	36.60
modify (initial)	51.76	24.28
open (cached)	26.73	0.17
modify (subsequent)	9.52	0
read (log)	7.32	0.51
append	7.08	0.05
read (original data)	6.35	0
close	0.73	0.11
seek	0.61	0.17

5. EXPERIMENTAL EVALUATION

We evaluate Coffee's operations through a series of benchmarks, and also measure its memory footprint and energy consumption. We have chosen a set of metrics that are justified by the needs in various real sensor network application scenarios. As the storage system is node-local, we conduct the experiments on a single Tmote Sky node [22]. The Tmote Sky has an MSP430F1611 processor, 10kb RAM, 48kb internal flash memory, and an ST M25P80 external flash module of 1Mb. The external flash memory is accessed through a 75MHz serial peripheral interface (SPI) bus. The page write operation takes 1.5 ms and the erase operation takes 2 s. Sectors are 64kb large and consist of 256 pages. Although the M25P80 uses the notion of pages, single bytes can be read and written randomly.

5.1 Micro Benchmark

To quantify the execution time of each of the main Coffee operations, we have devised a micro benchmark. After running the benchmark 100 times, we calculate the average time and standard deviation of the operations and round all values to two decimals. The read, append, and modify operations use page-sized buffers, and the file size is the default 4kb. As shown in Table 1, the operations can be divided into three groups; search and allocations, read and writes, and processor-based operations.

The most used group of functions is in the middle of the table. Being based on a small amount of I/O operations, they are more deterministic because no expensive search is needed. Instead, a fixed amount of bytes are read from or written to the flash memory at a known location. Hence, only small factors such as intermittent interrupt handling contribute to the variance.

A consequence of Coffee's design is that the search and allocation functions exhibit a significantly longer execution time and standard deviation than the other functions. The relatively high standard deviations of

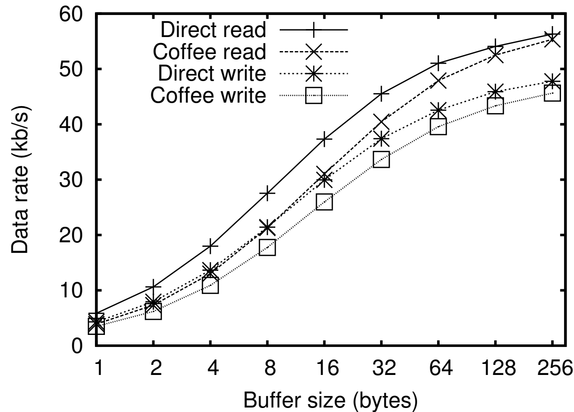

Figure 4: The performance overhead of sequential writes compared to direct flash access is negligible. Small buffers cause a higher processor execution cost per byte. Plotted on a lin-log scale.

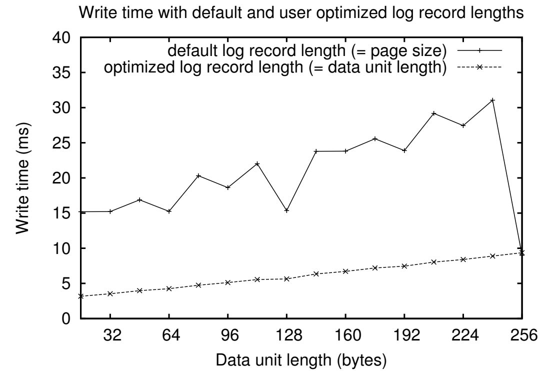

Figure 5: Writing to a user optimized log is faster than to a default log, because the former induces no read-modify overhead and no cross-page writing overhead.

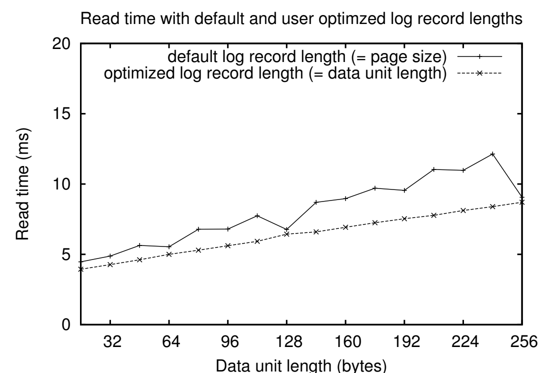

Figure 6: Reading from a user optimized log is slightly faster than from a default log because of elimination of cross-page reading.

these functions stem from the uncertainty of where the files are located in the flash memory. The initial file modification generates a log file allocation which also involves searching for available space. The search algorithm for finding available space is similar to that of finding a file, except for the condition to terminate the search. These two operations are uncommon, however, since the search results are cached.

The third and final group is the functions that are done mostly without touching the flash memory. The seek operation is entirely done in the CPU by doing a boundary check and then setting an offset in a file descriptor structure. The close operation writes a byte to the file metadata if the end of the file has advanced sufficiently to increase the EOF hint value.

5.2 Sequential I/O Performance

We evaluate Coffee's sequential read and write performance by reading and writing files to Coffee and vary the size of the files. The results, as plotted in Figure 4, show that the sequential write performance is 47.1 kb/s and the sequential read performance is 56.9 kb/s. Like log-structured file systems such as ELF [2], Coffee achieves a high sequential throughput. Coffee's sequential write throughput is at most 8% slower than direct flash driver access when using 256 byte chunks. We have verified that the performance is sustained as the files grow.

5.3 Micro Log Optimization

We evaluate and compare the reading and writing performance between the default log configuration and a user-optimized log configuration. The latter leverages the flexibility of Coffee's micro logs to set the log records just large enough to hold individual user data units. Assuming that users access and update data units randomly, we expect that the I/O speed can be accelerated by using a customized log record length.

In our experiments, we construct two files: one with the default log configuration and the other with the optimized log configuration. Each file consists of 10 data units of equal sizes. We loop through both files for 5 times, modifying every data unit for 5 times and verifying each modification with a read. We iterate the test for a range of data unit lengths from 16 bytes to 256 bytes. After measuring the time for each write and each read, we plot the performance of the two log configurations based on the average times.

Figure 5 shows that optimized log records reduce the write times substantially because Coffee simply fills a new record with data. There is no need to carry out a normal read-modify-update sequence since a complete record is overshadowed by the new data. If the write buffer differs in size from the log record, Coffee must read the most recent data from the file to merge the

contents with the write buffer into a new log record. This explains the sudden fall when the write buffer size finally matches the default log record size.

Figure 6 shows that the read performance of the configured log records and the default log records are comparable. The read log function stops reading from the flash as soon as the read buffer has been filled, which explains why the difference is not as large as when writing. Depending on the starting offset and buffer size, the default log read may break the read over a page and then read the log of the next page. Hence, the default log becomes slightly slower on average and the probability that the read must be broken is proportional to the write size.

5.4 Memory Footprint

Since miniaturization and cost reduction are driving the development of economical and practical sensor networks of scale, memory remains to be a scarce resource. One of the objectives of Coffee is to reduce the memory footprints significantly compared with earlier file systems for flash memories and sensor networks. We measure the memory footprints of temporary operations and static data.

Figure 7 shows the different parts that form Coffee's memory footprint. The file metadata and file descriptors have a constant size, whereas the I/O operations and garbage collection allocate most of their memory on the stack. Reading and writing to the log requires an order of magnitude more memory than other operations use; the log index table is preferably processed in large portions to improve the performance. Log writing also requires that up to a complete log record is copied into a buffer before partly overwriting the data in the buffer and writing the updated record into the log.

We compare the ROM and RAM footprints of Coffee with existing systems in Figure 8. Coffee's ROM footprint is approximately one third of the footprints of Capsule [15] and the Matchbox file system. The static RAM footprint of Coffee is smaller than 200 bytes, which is approximately one fifth of the RAM footprint of Matchbox and one eighth of that of Capsule. Unlike Capsule and Matchbox, Coffee does not need more RAM when the number of files or the sizes of files increase. An open file in Coffee requires only 15 bytes of memory when using a 32-bit file offset type.

A general figure of the ELF memory consumption cannot be determined because it depends on the amount of files in use and their sizes. Furthermore, ELF keeps track of free pages by using a bitmap. The bitmap requires 256 bytes to represent a 512kb flash memory, but it becomes cumbersome with larger flash memories. Coffee puts no data in the RAM for this purpose since free pages are contiguous in every sector.

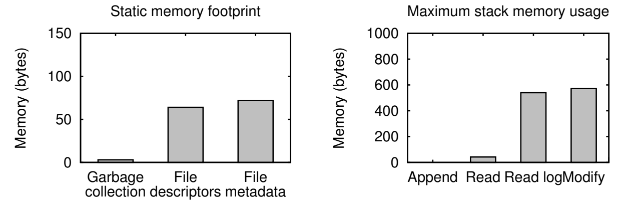

Figure 7: The total static memory footprint of Coffee is less than 200 bytes and is not affected by the number of files in the file system. The maximum stack memory usage for the most common operations, append and read, is less than 50 bytes each.

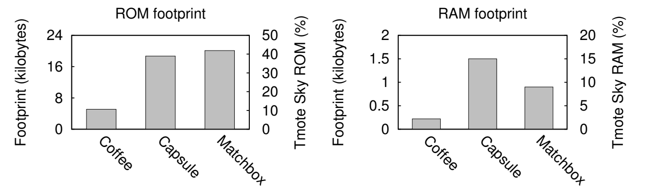

Figure 8: The memory footprint of a storage system must be small to leave room for applications, protocols, and other parts of the system. The Coffee code occupies approximately 10% of the total code ROM on a Tmote Sky and uses less than 5% of the RAM.

5.5 Energy Consumption

Although idle listening still dominates the energy consumption profile of most sensor devices, MAC protocols are becoming efficient to the degree that the radio listens less than 1% of the time in an idle network [18]. As storage-centric sensor networks are mostly idle by nature, the energy consumption of the storage system becomes more interesting since its relative share increases.

We look at the energy profiles of file reads and file writes to determine the added energy cost of Coffee. The operations use a file of 1kb, and we vary the chunk size to see how the overhead is amortized over more bytes as the chunk size increases. We quantify the energy consumption by using Contiki's software-based online energy estimation method [6].

The results are in Figure 9. When the chunk size is 1 byte, the overhead in terms of CPU energy is large, but the cost of the flash writes is also large per byte. The reason is that Coffee's processing of the operation is constant in time, regardless of chunk size, whereas the flash energy has a linear relation with the chunk size. When the chunk size increases towards 256 bytes–the flash page size–the overhead is negligible. The case of flash reads is slightly different: the CPU processing energy is larger when the chunk size is small, but the energy savings in flash reads flatten out quickly and be-

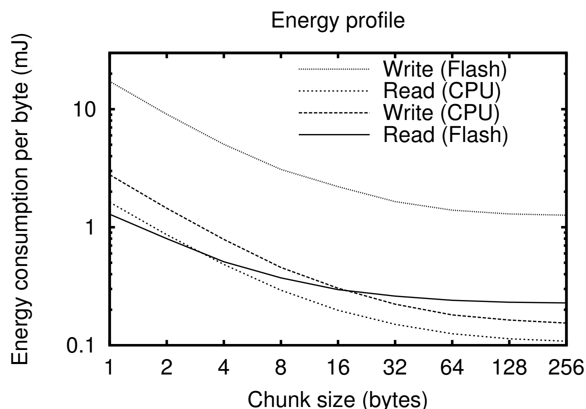

Figure 9: The energy overhead of using Coffee, and initializing the flash for the operations, is amortized as the chunk size increases. The axis use log scales.

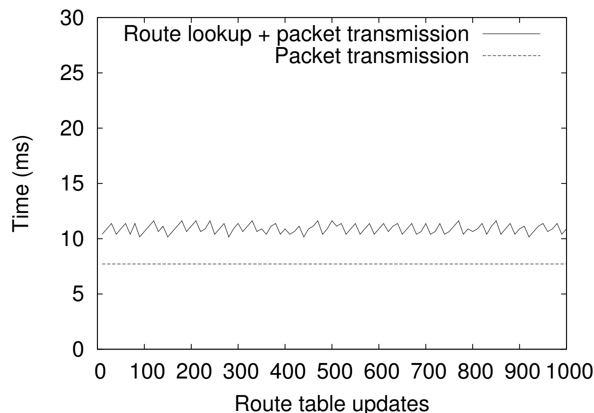

Figure 10: The lookup time for a routing entry depends on how many updates have been made to the routing table because updates are stored in the log. The sawtooth pattern is caused by automatic log merges.

come more costly from a 4 byte chunk size. The flash operations require initial setup such as turning off interrupts and transmitting a few bytes that tells the flash where to write or read the data.

6. A NETWORKING PERSPECTIVE

We look at Coffee in a networking perspective to answer the question if flash storage is suitable for components in communication stacks. To our knowledge, we are the first to perform such a study in a sensor network context. Further, this perspective serves as a demonstration that Coffee can be used to implement various services that are desirable in sensor systems. The components that use the file system can be viewed as an application-managed virtual memory, enabling us to use extended data structures that were earlier restricted to a portion of the RAM.

To quantify Coffee's capability and performance for storing and retrieving high-level objects, we have implemented a routing table that stores the route entries in Coffee, as well as a packet queuing module that uses Coffee to store packets. We measure the performance and implementation complexity of our mechanisms and show that the use of configured micro logs results in high performance.

6.1 Storing Routing Tables in Coffee

The trend toward IPv6-enabled sensor networks draws new attention to the problem of storing large data structures for routing and forwarding information. Neighbor discovery cache entries can include an IPv6 address, a link layer address, state information, timers, and possibly also a queued packet. Due to the limited RAM on most sensor devices, the neighbor discovery module typically keeps information about few neighbors. In IPv6, the limited neighbor table causes neighbor entry replacements and expensive protocol exchanges if a sensor node has a large neighborhood.

The sensor network performance in general may also benefit from having virtually unrestricted routing tables. By implementing the routing table on top of Coffee, we are able to store considerably more routes in it and at the same time free up valuable RAM. Additionally, by having a persistent routing table, the node can quickly resume operations after an eventual restart. When evaluating the Collection Tree Protocol, Fonseca et al. [8] show that the average number of transmissions drops significantly when allowing the sensor nodes to accommodate as many routes as they want.

Our experiment consists of a basic routing table that uses a Coffee file as a storage back-end. The routing table entries are 10 bytes large and consist of a node id, a routing metric, a next hop id, and a timestamp. Routes are added, deleted, modified, or retrieved through our routing API that uses file operations underneath it. The implementation consists of 53 lines of C code and uses 2 bytes of static memory and a maximum of 4 bytes on the stack.

Initially, we insert 1000 routes with random routing metrics. We change the routing metrics of 10 random destinations every second. The routing table file is 1kb, whereas the micro log is 512 bytes large and has 10 byte log records to reflect the routing entry. Coffee automatically merges the log after approximately every 30 route entry updates, since the file header and the index table also require space in the log file.

Figure 10 shows that the route lookup time depends on the number of records in the log and the log size. The send time constitutes both the copying of the buffer to the radio and the actual transmission of the data. The

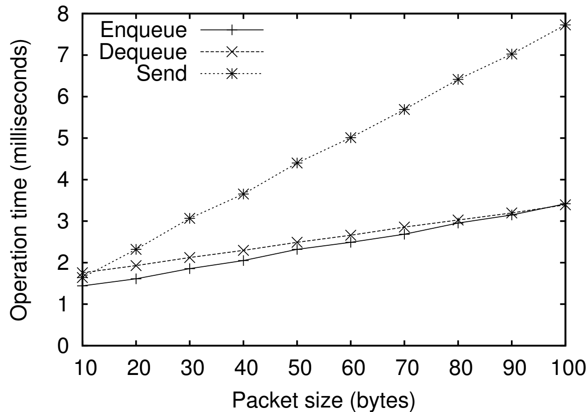

Figure 11: Enqueuing and dequeuing packets is fast: Less than four milliseconds for a 100 byte packet. In comparison, sending a 100-byte packet with the CC2420 radio device of the Tmote Sky takes eight milliseconds.

route lookups in the storage backed table are faster than packet transmission, and the lookup time is independent of the number changes in the routing table.

6.2 Queuing Packets through Coffee

Packet queuing tests Coffee in different way than routing tables: packets must be queued and dequeued quickly to adhere to the timing constraints of MAC protocols. Furthermore, queued packets are likely larger than routing table entries, since packets can be over 100 octets large in Wireless HART and 6LoWPAN [17]. Nevertheless, in comparison with the routing table, the number of entries is typically rather small in our experience. Examples of when packet queuing is used are data aggregation, delay-tolerant data mules, and mobile sensor networks with temporary connectivity loss.

We implement the packet queue by using common file system operations on a single file. The packet queue uses two pointers: one to the next packet in line for removal and one to the end of the queue where packets are added. Both pointers are incremented by the packet size modulo the file size.

The performance of the enqueue and dequeue operations is shown in Figure 11. The execution time increases linearly with the packet size. Both enqueuing and dequeuing is quick when compared with sending one packet over the radio: approximately 66% as fast as sending a packet. We do not expect that the time required for packet enqueuing and dequeuing to affect the system performance significantly, because most sensor network applications are not delay-sensitive. For high-throughput applications, recent work has shown that it is possible to achieve high throughput even with expensive node-local copying operations [21].

7. RELATED WORK

There is a considerable body of work on storage systems for flash memories that has influenced us. The ELF file system [2] is the work closest to ours. Like Coffee, ELF provides an extensive feature set including partial file overwrites, directories, and garbage collection. ELF is built on a log structure, which makes it a suitable alternative for random writing to NAND flash memories. Much like the conventional method, a file is represented in RAM as a linked list of small page groups, or *nodes*. Consequently, ELF allocates more memory as the files grow. ELF alleviates the problem of large memory requirements by using an additional EEPROM to store metadata, but is difficult to port to systems without such an EEPROM. ELF's predecessor in TinyOS, MatchBox, is more limited than ELF and Coffee since it does not support partial file overwrites.

Capsule is another comprehensive storage system providing storage abstractions for stacks, queues, streams, and indices. A simple file system is implemented by using a combination of these abstractions. While providing useful storage abstractions, Capsule requires over 25kb for storing its code when all of its abstractions are linked into the system. Hence, Capsule requires more than half of the code memory of the Tmote Sky.

TFFS [9] is a transactional flash file system designed for memory-constrained embedded systems. It is based on a search structure called pruned version trees. TFFS maps logical sector numbers to real sector numbers in RAM and stores B-tree nodes there as well. TFFS memory footprint matches that of Coffee on smaller non-volatile memories, but uses significantly more RAM on larger non-volatile memories. As sensor devices are increasingly being equipped with SD cards of 1Gb or more, we emphasize that it is important support large storage devices with very limited RAM.

In addition to file systems, various efforts have been directed towards databases and storage abstractions for sensor devices, including FlashDB [20], DALi [24], and MicroHash [27]. Nath and Gibbons propose a new abstraction for flash memory called B-FILE [19]. The B-FILE abstraction has "semi-random" write semantics in NAND flash, without having the large memory overhead of log structures. The design is mainly aimed at large sequential logs of samples and self-expiring items.

In-Page Logging (IPL) [14] is the database design that is in a sense most similar to Coffee's type of log structure. Both Coffee and IPL distinguish original data and log records. IPL divides each erase unit into a large data area and a small log region. Once the log region is full, the data area is merged with the log region into a new erase unit. Coffee, in contrast, can store multiple logs in the same erase unit on demand and adapts more easily to the workload since the log size and log record gran-

ularity are configurable. Coffee does not offer an explicit indexing service or a database query engine, but provides an interface upon which such systems can be implemented portably.

8. CONCLUSIONS

We have presented and evaluated the Coffee file system for flash-based sensor platforms. Our experience with Coffee has shown that it is small enough for inclusion by default in a sensor network OS, while also being easily portable to a wide range of platforms. We think that the most important aspect of Coffee is that files are represented with a small and constant RAM footprint. Furthermore, the tunable micro logs that we introduce are a strong point since multiple flash memory devices support random access I/O. In a networking perspective, we have shown that Coffee's high throughput and low latency make it a suitable underlying layer for storage abstractions in the networking stack.

Acknowledgments

This work was financed by VINNOVA, the Swedish Agency for Innovation Systems. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

9. REFERENCES

[1] R. Balani, C. Han, R. Kumar Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. In *Proceedings of ACM & IEEE EMSOFT*, pages 112–121, Seoul, Korea, 2006.

[2] H. Dai, Michael N., and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of ACM SenSys'04*, Baltimore, MD, USA, November 2004.

[3] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Re-thinking data management for storage-centric sensor networks. In *Proceedings of CIDR'07*, Asilomar, CA, USA, January 2007.

[4] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of ACM SenSys'06*, Boulder, USA, November 2006.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of Emnets I*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of Emnets IV*, Cork, Ireland, June 2007.

[7] P. Dutta, D. Culler, and S. Shenker. Procrastination might lead to a longer and more useful life. In *Proceedings of HotNets-VI*, Atlanta, GA, USA, November 2007.

[8] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Four-bit wireless link estimation. In *ACM HotNets-VI*, Atlanta, Georgia, USA, November 2007.

[9] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005.

[10] L. Gu and J. Stankovic. t-kernel: providing reliable OS support to wireless sensor networks. In *Proceedings of ACM SenSys'06*, Boulder, Colorado, USA, November 2006.

[11] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *Proceedings of ACM SenSys'07*, Sydney, Australia, November 2007.

[12] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of IPSN'07*, pages 254–263, 2007.

[13] A. Lachenmann, P. Marrón, M. Gauger, D. Minder, O. Saukh, and K. Rothermel. Removing the memory limitations of sensor networks with flash-based virtual memory. *SIGOPS Oper. Syst. Rev.*, 41(3):131–144, 2007.

[14] S-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of SIGMOD '07*, 2007.

[15] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of ACM SenSys'06*, Boulder, Colorado, USA, November 2006.

[16] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor networks. In *Proceedings of IPSN/SPOTS'06*, Nashville TN, USA, April 2006.

[17] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. Internet proposed standard RFC 4944, September 2007.

[18] R. Musaloiu-E., C-J. M. Liang, and A. Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *Proceedings of IPSN '08*, 2008.

[19] S. Nath and P. Gibbons. Online maintenance of very large random samples on flash storage. In *Proceeings of PVLDB '08*, 2008.

[20] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of IPSN'07*, Cambridge, Massachusetts, USA, April 2007.

[21] F. Österlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *Proceedings of ACM HotEmNets'08*, June 2008.

[22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[23] M. Rosenblum and J. Ousterhout. The design and implementation of a log structured file system. In *Proceedings of ACM SOSP'91*, 1991.

[24] C. Sadler and M. Martonosi. DALi: a communication-centric data abstraction layer for energy-constrained devices in mobile sensor networks. In *Proceedings of ACM MobiSys'07*, San Juan, Puerto Rico, June 2007.

[25] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. Luster: wireless sensor network for environmental research. In *Proceedings of ACM SenSys '07*, 2007.

[26] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of NSDI'06*, Seattle, November 2006.

[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'05*, San Francisco, California, USA, 2005.