Contiki Ring File System for Real-Time Applications

Sebastian Schildt, Wolf-Bastian Pöttner and Lars Wolf
Institute of Operating Systems and Computer Networks
Technische Universität Braunschweig,
Braunschweig, Germany
Email: `[schildt|poettner|wolf]@ibr.cs.tu-bs.de`

***Abstract*—Wireless Sensor Nodes often need to store or cache measurement or operational data into some non-volatile memory. Many applications using low power sensor nodes are based on Contiki, which provides the COFFEE filesystem for data storage. As COFFEE introduces occasional long and hard to predict write delays, it can not be used in applications which have real time demands, such as sensor networks operating in industrial environments.**

In this paper we present an efficient implementation of a flexible Ring File System for Contiki, that can be used in real-time applications. Read and write times are predictable and adherence to Contiki's filesystem API allows it to be used as a drop-in replacement for COFFEE in many applications.

I. Introduction

For cost and energy efficiency reasons wireless sensor networking applications are mostly build using relatively weak hardware platforms. A common sensor node is often powered by an 8 bit microprocessor and has a few kiB of RAM and a few tenths of kiB as flash storage available. If any operating system is used, you will usually find small cooperative multitasking runtime systems such as TinyOS[1] or Contiki[2]. In many applications there is the requirement that the system needs to be able to store data on the sensor node itself. Either the sensor is disconnected for long-term monitoring, and data will be gathered by a ferry [7] later, or you just want to cache sensed data in the case of temporary transient link failures. In addition, some systems may collect data over a longer period (e.g. about radio link quality) and store the readings to flash. Later, the collected raw data can be processed en-bloc producing a short summary that can be sent via the wireless interface [5].

The Contiki operating system already provides the COFFEE filesystem which can be used for these applications. COFFEE, however, can not work in real time environments, as its write times are unpredictable and Contiki does not preempt its tasks. Real-time capability is needed, if your sensor network uses a TDMA scheme for communication, or if your node also controls actors in closed loop control. This is usually the case for sensor networks deployed in industrial settings: For example, the GINSENG project [4] developed a sensor/actor network for industrial applications, where strict reliability and real-time requirements need to be met. This is achieved by using the GinMAC TDMA MAC, which uses a static tree topology where each device gets a fixed number of time slots in a GinMAC "epoch" to communicate with its parent or child nodes. If you want to write or read data in such a scenario, the file read or write operations need to be scheduled according to the MAC's TDMA time schedule, and must be guaranteed to finish before the next active slot for this node arrives.

In this paper we present RFS, a ring file system for Contiki. RFS offers a similar API to COFFEE so it can act as a drop-in replacement for many applications. Files are organized as FIFO buffers of linked flash pages. RFS tries hard to minimize copying of data. For potentially long-term operations such as flash reorganization during garbage collection, RFS allows the user to schedule them at a time when the application can tolerate it. RFS can give an advance warning that a potentially costly operation is pending, giving the application time to schedule it. Due to this mechanism, normal read and write operations in RFS are much more tightly bounded than with COFFEE, making it feasible to deploy RFS in real-time applications.

The remainder of this paper is organized as follows: Section II gives an overview of the state of the art in Contiki and lines out the goals for RFS design, in section III we introduce RFS' architecture and key data structures. Section V shows RFS' performance for various operations. Finally, in section VI we give some concluding remarks and line out future work.

II. RFS Design Goals

Contiki ships with the COFFEE filesystem [9], which is a versatile filesystem supporting common create, read, write and append operations. COFFEE is a log-structured filesystem, which makes sense for flash based storage devices, which only support limited random access capabilities. Another log-structured sensor node filesystem is ELF [1] for TinyOS, which uses a global log, compared to the per-file logs in COFFEE.

COFFEEs complexity also has some disadvantages: When creating a file, its size needs to be reserved beforehand. If you write more than the reserved size COFFEE can extend the file. However, since COFFEE requires files to be stored linearly, this usually encompasses reserving a new bigger file, and copying the contents from the old one. While this is transparent to the programmer, it can consume a lot of time when this happens

[1]http://www.tinyos.net/
[2]http://www.contiki-os.org/

during a write operation, which is hard to predict. Also, COFFEE can get problems with fragmentation, as it needs continuous space for a file, and uses a first-fit strategy for finding a free area. COFFEE tries to alleviate this problem by an active garbage collector that moves data around, in order to create more erasable sectors. COFFEE as well as ELF support random-access semantics, which make them comfortable to use but which also adds quite a bit of complexity to the code dealing with logs and committing transactions to the storage.

While the overall performance of COFFEE is reasonably good, the unpredictable delays due to reorganizing the flash are a problem for real-time applications. In a cooperative multitasking system such as Contiki, which has only limited abilities to preempt a task, this can quickly become a problem. Basically, using a system like COFFEE precludes the usage of a TDMA based MAC protocol. The problem is not that operations might take longer, but that it is hard to predict *when* an operation will take longer. For example, take GinMAC [6], a TDMA MAC protocol for industrial applications used in the GINSENG project. GinMAC already has a designated "pre-transmission processing time" in each TDMA cycle, which is 2.44 ms in the default configuration. When performing file operations during this time, it must be guaranteed that those functions always return within this time to prevent disrupting the TDMA schedule. Furthermore, GinMAC also offers specific processing slots in each "epoch" which could be used for long-term operations that are pending.

While it is hard to guarantee those limits in full-featured file systems such as COFFEE, which have a lot of book keeping to do, we argue, that for many tasks on sensor network nodes a random access capable file system is not needed: If your data is highly volatile, such as routing tables, they are better kept in RAM anyway, as performance would always be limited by the relatively slow random access times of storage commonly used on network nodes (internal flash, EEPROM, SD cards). On the other hand, for logging tasks just a simple log or ring buffer that new data only gets appended to is enough. As most sensor nodes run more than one task that potentially wants to store data in flash, we do however think, that an abstraction from the underlying flash, i.e. a file system is still desirable.

Therefore we developed the Contiki Ring File System (RFS) keeping the following requirements in mind:

- RFS supports different files for different applications.
- An RFS file is an abstraction of a FIFO: Writes append data, reads consume data. Therefore an RFS filesystem can also be seen as a collection of variably sized ring buffers.
- RFS is efficient, even if file sizes and access patterns are not known beforehand.
- RFS can be deployed in applications requiring real-time guarantees .
- RFS avoids copying user data around when not absolutely necessary.
- RFS provides adequate wear levelling for flash memories.
- RFS implements CFS, the general Contiki interface for filesystems (without random access functionalities).

III. RFS ARCHITECTURE

This section describes RFS' on-flash as well as in-memory data structures. In RFS files are operating like FIFO buffers: Writing appends data to the FIFO and reading consumes the data. There is no need to specify the size of a file beforehand: Files grow as needed, as long as there is space in the storage area. Files are constituted of linked areas of flash storage (RFS blocks). Because usually flash memories do not allow to change arbitrary bytes, the data structures pointing from an RFS block to the next can not be changed, even when garbage collection and wear levelling needs to move data around. Therefore, all addresses to flash pages in RFS are *virtual* addresses, assuming a sequential, continuous area for data storage. RFS maps these virtual page addresses to the current physical address of the data as needed. The details of this mapping are explained in section III-D.

A. Flash Organization

RFS is designed to work directly on flash (or eeprom) storage without an intermediate controller performing any mapping between RFS and the flash. Flash memories are usually organized into pages and sectors. Only full pages can be programmed and only full sectors can be erased. Usually, an erased sector contains only 0s, and subsequent programming of any of the pages can set bits from 0 to 1, but can not reset any bits to 0 without erasing the whole sector first. Depending on hardware and driver the logic might be inverted, i.e. an erased sector might contain only 1s and 0s can be programmed. RFS uses the convention, that erased blocks are all 0, and 1s can be programmed.

RFS' flash organization can bee seen in figure 1. RFS combines a configurable amount of pages into an RFS Block. An RFS file consists of linked RFS blocks. The whole storage area is organized into sectors, where each sector can hold a certain amount of RFS blocks. Sector information in the beginning of each data sector stores information, which RFS blocks in the current sector contain obsolete data, which are actively used and which blocks are still available. One sector is reserved for metadata information about files. This speeds up the opening of files, as only the metadata sector needs to be searched for the file header. RFS will always keep one completely erased sector in reserve for garbage collection (see section IV).

As the garbage collection might change the location of the data, header and temporary sector, RFS uses the concept of virtual sectors: Each data sector has a virtual sector number. When garbage collection copies data from one sector to another physical sector, the new sector will inherit the virtual sector number of the source.

B. API and Data Structures

RFS implements the CFS API from Contiki, which means it can be used as a drop-in replacement for other Contiki filesystems. However, there are some limitations: Currently RFS does not support directories and the `cfs_seek` call is not supported.

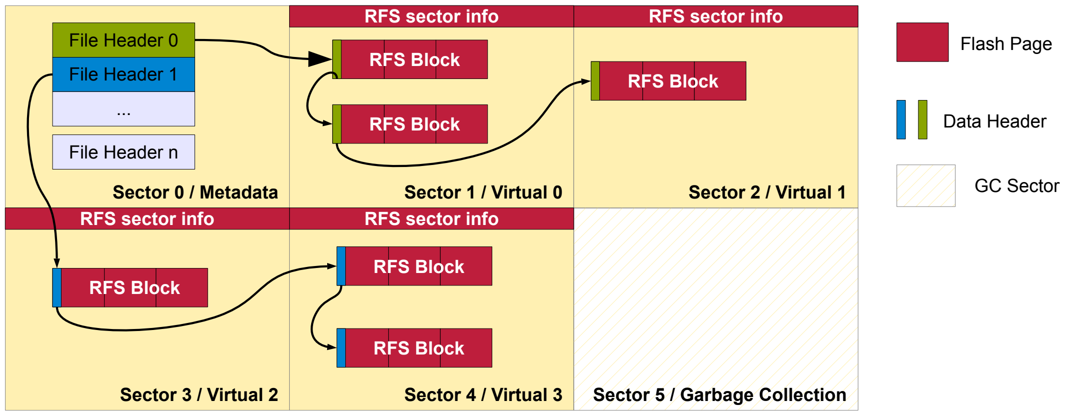


Figure 1. RFS flash organization

```
struct file_header {
   uint16_t firstBlock;
   uint16_t readPtr;
   uint16_t writePtr;
   uint8_t flags;
   uint16_t header;
   char name[RFS_NAME_LENGTH];
};
```

Listing 1. RFS on-flash file header

C. File header

The on-flash structure of a file header is shown in listing 1. Files are identified by name, which is stored in the `name` field. The `firstBlock` field contains the virtual page address of the first data page of the first RFS Block for this file. The read pointer is persisted to indicate how much bytes starting from `firstBlock` have already been consumed when the file was last opened. The write pointer also denotes current file size. Both pointers count bytes since the beginning of the file. The `flags` field marks the file header as either *active* or *obsolete*.

File headers are stored sequentially in the metadata sector. The position of this file header in the file header list in metadata sector 0 is stored in the `header` field for efficiency reasons: When a file header becomes invalid, because the file was changed or deleted, the corresponding header can be marked invalid. If needed, an updated file header will be appended to the end of the header list.

When an file is opened for reading or writing, the data structure is expanded in memory as shown in listing 2. The `currentReadBlock` and `currentWriteBlock` store the virtual page address of the RFS Block which currently contains the read- or write pointer. This makes reading and writing operations faster, as it precludes the need to traverse the linked list of RFS Blocks beginning with `firstBlock` on every read or write operation.

Each RFS file consists of a number of RFS blocks. Each RFS block consists of a number of data pages for user data and begins with a data header as seen in listing 3. For the data header the `nextPage` entry points to the next RFS block .

```
struct file_header {
   uint16_t firstBlock;
   uint16_t currentReadBlock;
   uint16_t currentWriteBlock;

   uint32_t readPtr;
   uint32_t writePtr;

   uint16_t header;
   uint8_t references;

   char name[RFS_NAME_LENGTH];
}
```

Listing 2. RFS Ram file descriptor

```
struct data_header {
         uint16_t nextPage;
         uint8_t flags;
};
```

Listing 3. RFS data block header

D. Virtual Sector Mapping

When the garbage collection (see section IV) moves data between physical sectors, the page addresses in the `firstBlock` field of the file header and the `nextPage` field of data headers already written to flash can not be modified. Therefore, all page addresses used in RFS data structures

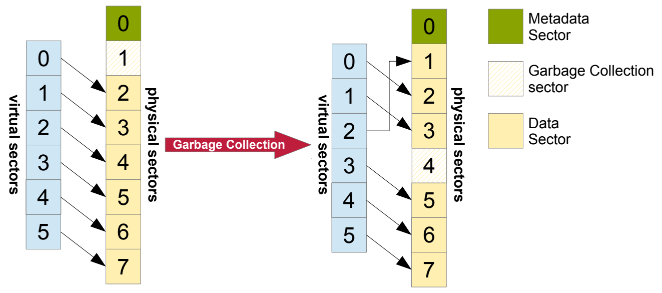


Figure 2. RFS virtual sector mapping

```
struct sector_info{
  int8_t type;
  int8_t virtualSecNum;
  enum page_state state[RFS_BLOCKS_PERSECTOR];
};
```

Listing 4. RFS on-flash sector information

```
struct sec_summary {
    uint16_t used;
    uint16_t active;
};
struct sec_summary sec_sum[RFS_SECTOR_COUNT];
```

Listing 5. RFS RAM sector summary

assume that all data sectors are numbered sequentially from 0 up to he number of data sectors in the current configuration. RFS maintains a mapping from virtual to physical sectors and uses this information to convert a virtual page address to the current physical page address. To convert the virtual page address v_page, first this page's virtual sector is determined:

$$v\_sec = \left\lfloor \frac{v\_page}{RFS\_PAGES\_PER\_SECTOR} \right\rfloor$$

The real sector r_sec corresponding to the virtual sector v_sec is looked up in the current mapping. Then the physical page address p_page can be determined by

$$\begin{aligned} p\_page &= v\_page \\ +(r\_sec - v\_sec) &\cdot RFS\_PAGES\_PER\_SECTOR \end{aligned}$$

For an example look at figure 2: The left side shows the mapping between virtual and physical sectors as it might be on a freshly formatted RFS filesystem. As the metadata and garbage collection sector do not have a virtual sector number, in the beginning the mapping simply realizes an offset of two: Virtual sector 0 maps to physical sector 2, virtual 1 maps to physical 3 etc. When the garbage collection decides to move data in sector 4 (virtual sector 2) to sector 1, the mapping from virtual sector 2 is adapted accordingly to reflect this. The virtual page address conversion makes sure, that now virtual page addresses in virtual sector 2 map to physical sector 1.

E. Sector Metadata

Each sector has an associated data structure that keeps track of used and free blocks. Listing 4 shows the sector information. A sector's type can be header sector, data sector or garbage collection sector. The `virtualSecNum` stores this sector's virtual sector number, which is used for mapping virtual to real sectors. The `state` field can indicate a *free*, *active* or *obsolete* block. Initially all blocks are free. When a block is appended to a file, it transitions to the active state. As RFS files represent FIFO buffers, once the read pointer leaves a block, that block is considered obsolete. For performance reasons and to avoid writing to flash too much, the sector information in flash will only be updated when a file is closed, or before a garbage collection is performed.

For normal operations a small summary of sector info as seen in listing 5 is stored in RAM. When a new RFS Block is reserved in a sector, the `used` and `active` counts in the corresponding sector info will be incremented. When an RFS Block has been read completely, i.e. the block has been *obsoleted*, the `active` count of the virtual sector's corresponding sector summary will be decreased.

IV. Garbage Collection and Wear Leveling

As flash storage cells have only a limited number of erase-cycles and there is no intermediate controller between the file system and the flash (in contrast to SD-Cards or SSDs), the file system should make sure, that writes are distributed to all flash cells evenly. A ring file system, which is primarily designed to *append* data to existing files is already a good start to provide adequate wear levelling.

A. Metadata Sector

Special care needs to be taken with the metadata information: A file's read- and write pointers change with every write or read operation. In RFS, once a file is opened this data will be kept in RAM. Once a file is closed, the old file header in the metadata sector is invalidated, and the updated file header will be written to the end of the file header list. Once the metadata sector is filled up, the live entries will be copied to the garbage collection sector, which becomes the new metadata sector. The old metadata sector will be deleted and becomes the new garbage collection sector.

B. Data Sectors

A simplified version of the garbage collection algorithm for data sectors is shown in algorithm 2. When a write request needs to add another RFS block to a file, the sector information in RAM will be scanned sequentially to find a free RFS block. If such a block is not found, depending on the configuration the write fails or the garbage collection is initiated.

The garbage collection first truncates all files. Obsoleted blocks will be marked as obsolete in their corresponding sector info structures. The algorithm for truncating files is shown in algorithm 1. The number of obsolete blocks in a file is determined by looking at how many RFS blocks the file's read pointer has already traversed completely (line 4). The sector information of the sector containing the files's first RFS block is loaded, and if there is at least one obsolete block, the first will be marked obsolete in the sector information. Then the linked structure of RFS blocks is followed for the amount of obsoleted blocks, marking each obsolete in the sector information. Whenever the next block is located in another sector than the previous one, the sector information for the last sector will be written to flash and the information for the new sector will be loaded instead (lines 11-15). There is a small probability that this leads to a situation that for truncating a given file the information for one sector needs to be updated more than once, if due to fragmentation the linked

Algorithm 1 Truncate file

```
1: procedure TRUNCATE_FILE($file$)
2:     $currSec$ =SECTORFORPAGE($file.firstBlock$)
3:     $currSecInfo$ =READSECTORINFO($currSec$)
4:     $obsoleteCount = \lfloor (file.readPtr)/(RFS\_BLOCK\_SIZE) \rfloor$
5:     $block\_page = file.firstBlock$
6:     while $obsoleteCount > 0$ do
7:         $blk\_num = (block\_page \,\%\, RFS\_Pages\_PER\_SECTOR)/RFS\_BLOCK\_SIZE$
8:         $currSecInfo[blk\_num] = OBSOLETE$
9:         $obsoleteCount = obsoleteCount - 1$
10:        $block\_page =$ DATA_HEADER($block\_page$).$nextHdr$
11:        if SECTORFORPAGE($block\_page$) $\neq currSec$ then
12:            WRITESECTORINFO($currSecInfo$,$currSec$)
13:            $currSec$ =SECTORFORPAGE($block\_page$)
14:            READSECTORINFO($currSec$)
15:        end if
16:    end while
17: end procedure
```

list of RFS blocks leaves sector i for one block, but comes back to it for a newer block. This is a design trade-off we made to preclude having to load the sector information of all sectors into RAM all at once.

In the next step the sector summary's `active` counts are scanned to find the sector which has the least live data (algorithm 2, lines 6-13). Truncating files and updating sector information in the previous step was necessary, because while the `active` counts can show which sectors contains the least active data, they cannot show which specific RFS blocks are obsolete.

In the final step the live data from the chosen block is copied to the garbage collection sector (lines 15-25) and the selected sector will be erased becoming the new garbage collection sector. This operation can take some time, if too much data is alive in a sector, i.e. when the storage space is highly fragmented. To enable tighter bounds in a real-time system, RFS can also keep track of the number of free RFS blocks and not initiate garbage collection automatically. It is then the responsibility of the application to monitor the number of free blocks and trigger a garbage collection manually at a convenient time.

V. EVALUATION

The evaluation was performed on a Maxfor MTM-CM5000-MSP which is a TmoteSky [3] clone. The node features an MSP430 processor with 10 kiB of integrated RAM and 48 kiB of integrated flash storage. RFS and COFFEE use the M25P80 SPI dataflash [8] available on the node. The M25P80 uses a page size of 256 bytes, where 256 pages make up one erase sector. The chip contains 4096 pages in 16 sectors.

A. COFFEE

As a baseline test, we did a very basic COFFEE evaluation: We created a COFFEE file with default parameters and wrote 50 kiB data to it. We choose to write data in 8 bytes steps, as small writes are common when logging sensor data. The results can be seen in figure 3(a).

We found, that most writes take around 0.25 ms, however every once in a while writing takes significantly longer (more than 700 ms in this test case). This is due to the fact that we exceed the maximum amount of reserved blocks for that file. In this case COFFEE will create a new bigger file, and copy the complete data from the old file to the new file. Once the new file's size is exceeded again, the data is duplicated once more, which will take longer each time the file is extended, as there is more data to copy. Now, for this simple test case COFFEE could probably be modified to extend a file, if free blocks are available at the end of the file. However, since COFFEE requires files to be stored continuously in flash, as soon as more tasks use the file system some amount of copying will be needed.

It is not a problem, that occasionally a file system needs to perform some management and clean-up tasks. The main problem is, that even in this simple case COFFE occasionally needs a time which is more than two magnitudes higher than the median writing time to execute a write due to these tasks. So, to make your system real-time capable with COFFEE, you need to schedule a time that is 3 orders of magnitude larger than the average execution time for file system operations. While in COFFEE this problem can be alleviated by setting the "expected" file size before, so that extension never occurs, we found, that for many tasks the maximum amount of storage needed is hard to know beforehand. Over provisioning is not a solution for resource constrained devices.

B. Ring File System Write Times

We performed the same test as in section V-A with RFS. The results can be seen in figure 3(b). The figure shows, that for RFS maximum write times are more tightly bounded compared to COFFEE.

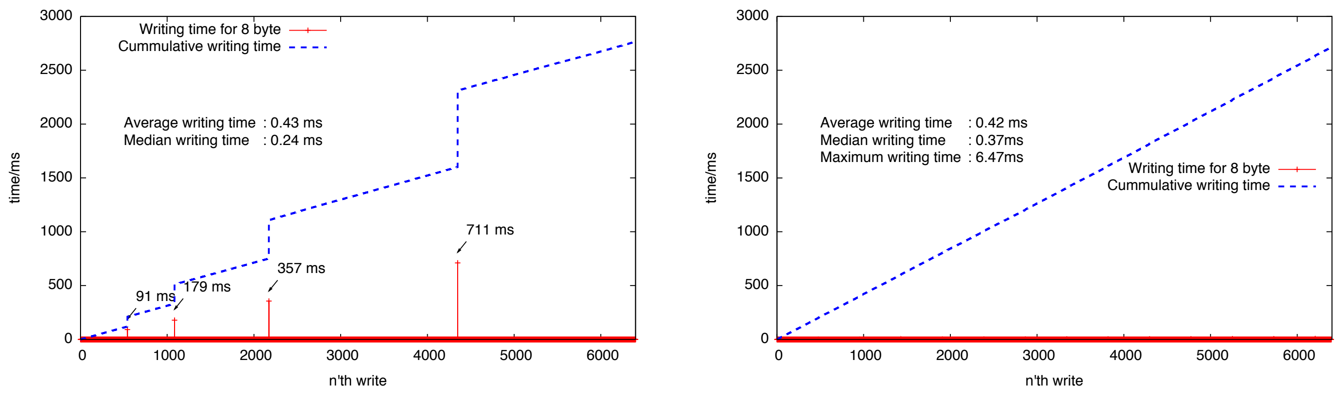
It can be seen that the variance is much smaller, and there

Algorithm 2 Simplified garbage collection

```
procedure GARBAGE_COLLECT
    for $file$ in $open\_files$ do                                   ▷ Truncate files
        TRUNCATE_FILE($file$)
    end for

    $minActive = sec\_info\_ram[0].active$                            ▷ Find least populated sector
    $src\_sec = 0$
    for $v\_sec$ in $data\_sectors$ do
        if $sec\_info\_ram[v\_sec].active < minActive$ then
            $minActive = sec\_info\_ram[v\_sec].active$
            $src\_sec = v\_sec$
        end if
    end for

    $src\_secinfo$ =READSECTORINFO($src\_sec$)                         ▷ Copy active blocks to new sector
    $tgt\_secinfo$ =EMPTYSECTORINFO( )
    $target\_sec = gc\_sec$
    $count = 0$
    for $block$ from 1 to $BLOCKS\_PER\_SECTOR$ do
        if $src\_secinfo[block].state == ACTIVE$ then
            COPY_BLOCK($collect$,$block$,$target\_sec$,$count$)
            $tgt\_sec\_info[count] = ACTIVE$
        end if
        $count = count + 1$
    end for
    WRITESECTORINFO($tgt\_sec\_info$,$target\_sec$)
    $sec\_info\_ram[target\_sec].active = sec\_info\_ram[target\_sec].used = sec\_info\_ram[source\_sec].active$
    ERASESECTOR($src\_sector$)
    $gc\_sec = src\_sec$
end procedure
```



(a) COFFEE

(b) RFS

Figure 3. Writing to a file

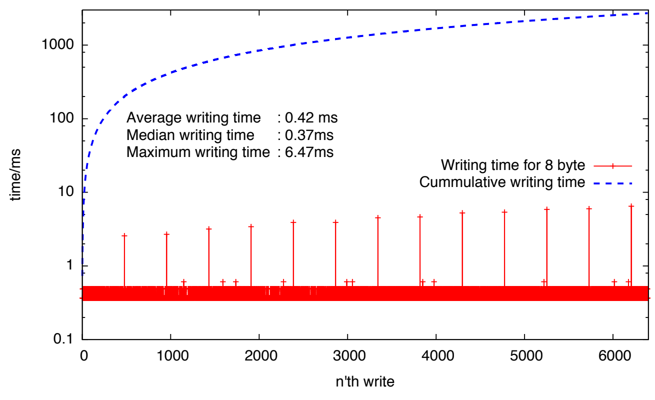


Figure 4. RFS file extension

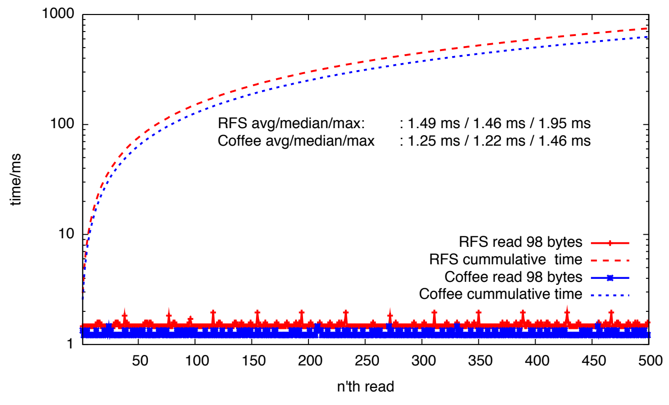


Figure 5. Reading 98 byte blocks

are less extreme outliers. The maximal writing time to full fill the request is 6.47 ms compared to 711 ms for COFFEE. The average writing time is almost similar, with the slight advantage for RFS due to COFFEE's outliers. In total, RFS can finish writing the total 50 kiB approximately 50 ms earlier. In this configuration the maximum writing time occurs every 3820 bytes, when RFS has to reserve a new RFS block. This behaviour can be seen in figure 4. In RFS the user has the option to disable the automatic extension of files. In this case the maxima in figure 4 can be avoided. Without automatic file extension it is the applications responsibility to extend a file at a convenient time before all free space in the current RFS block of a file is used. RFS provides the application with the amount of bytes still available in the current block to allow it to make an informed decision.

C. RFS Read Times

We also performed a read test to show the real-time fitness of this operation. We choose to read in blocks of 98 bytes, as this is the maximum size of payload in a GinMAC frame. The results for COFFEE and RFS are shown in figure 5. It can be seen that in this test neither COFFEE nor RFS should have a problem meeting real-time requirements. The overall performance of RFS is slightly worse than COFFEE. The reason is RFS' support for having non contiguous files, which is an important factor in enabling real-time support when writing files. Due to this feature, RFS needs to perform a mapping from virtual to pyhsical addresses (see section III-D) for each read. Also, just as in the writing test, a read operation might need to change from one RFS block to the next which takes some additional, but well bounded, time.

D. Garbage Collection Performance

All flash based file systems need to perform garbage collection at some time. This includes erasing sectors that do not contain any useful data anymore, so they can be filled with new data. This is a lengthy process, as deleting a sector takes a much longer time compared to programming or reading. The M25P80 has a "typical" erase time of 2 s and a specified maximum of 3 s [8]. In practice, this value depends on the

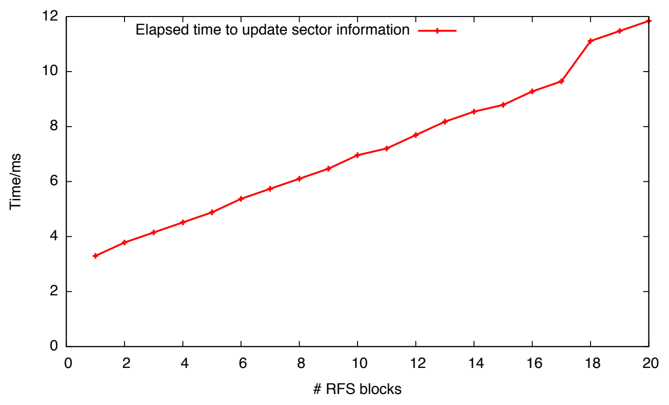


Figure 6. Garbage collection: Truncating files

individual chip, but in any case it is some orders of magnitude larger than a typical program or read cycle (the typical duration of a page program cycle is 1.5 ms for the M25P80), which is the reason why in RFS the user has the option to schedule garbage collection when appropriate.

As explained in section IV, in the first step the garbage collection will truncate all open files, adding obsolete and active block information to the sector information stored in the first page of each sector. For the truncate operation the linked list of RFS blocks of a file needs to be traversed, so the time needed to update the sector information depends on the number of RFS blocks in a file. The times for truncating a file can be seen in figure 6. It can be seen, that the time rises linearly with the number of blocks, until 17 blocks. Obsoleting the 18th block takes more time. The reason is, that each test was performed with only one file. Files which span more than 17 blocks fill a full sector and need to be extended to the next sector. That means, when switching to RFS block 18, the garbage collection needs to write the sector info for the sector containing block 17 back to flash and load the sector information for the sector containing block 18 (see algorithm 1, lines 11-15). This additional delay occurs, when the next linked RFS block resides in another sector than the previous one.

Before erasing the old sector, the garbage collection needs to

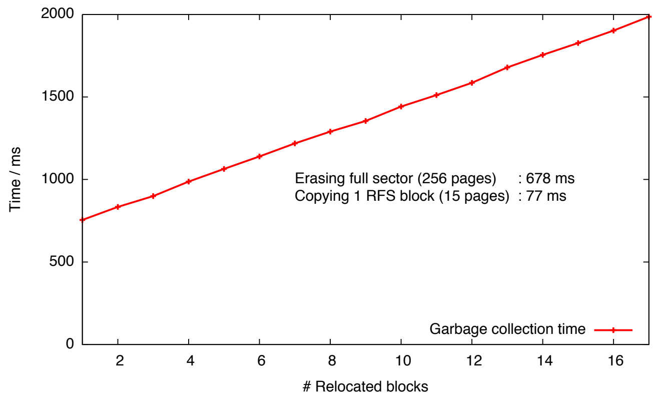


Figure 7. Garbage collection: sector copy

	RFS	COFFEE
ROM (.text)	5432	5184
RAM (.data & .bss)	262	144

Table I
RFS MEMORY FOOTPRINT

copy the live data to the garbage collection sector. Obviously, the time needed for this operation depends on the amount of live data that needs to be copied from one sector to the new sector. The results of copying different amounts of live data from one sector to another are shown in figure 7. The figure plots the average of three runs. For the device measured, the average time for erasing one sector was 678 ms, the average time to copy one RFS block (15 pages) from one sector to another was 77 ms.

E. Memory Footprint

To give an impression about RFS resource usage, we measured RAM and ROM footprint on the TMoteSky node. RFS does not use any dynamic RAM, so it can also be used on platforms where dynamic RAM allocation is not available. Table I shows RFS's and COFFEE's memory footprint in their respective default configurations. RFS uses slightly more space in ROM.

In its default configuration RFS uses more RAM on the TMote Sky compared to COFFEE. We do want to note however, that 262 bytes are still only around 2.6% of the RAM available to the TMote Sky. In RFS each file that can be opened needs 36 bytes metadata in RAM, which will be allocated statically even if the file is not opened. In the default configuration RFS and COFFEE support 6 concurrently opened files. With 3 open files RFS's RAM usage drops to 152 bytes (COFFEE 104 bytes).

VI. CONCLUSIONS

We presented RFS, a ring file system for sensor nodes that can be used in real-time environments. RFS can achieve real-time compliance by separating tasks with a small and known runtime such as normal reads and writes, from potentially longer running tasks such as extending a file or performing garbage collection. RFS can give an advance warning when such a time consuming operation might be necessary to allow the file system to accept writes in the future. This gives applications the chance to schedule these maintenance operations. In contrast to COFFEE, which offers more functionalities and can be quite effective when the file size and file access patterns are known beforehand by tuning the per-file log structure, RFS is designed to operate efficiently even when these details are not known beforehand. RFS is flash aware and performs basic wear-levelling, which makes it possible to use it directly on flash memory. As RFS adheres to the Contiki CFS API it can be used as a drop-in replacement for applications that do not rely on random access abilities.

Currently RFS is focussed on buffering tasks: Reading from a file invalidates the read data. Many applications might want to read data more than once. Examples are configuration data or maybe a new system image from an over-the-air flashing system that needs to be checked for consistency before it will be flashed. It is easy to extend the RFS file descriptor with a flag preventing invalidation of already read data. For read access it will even be possible to provide random access with minimal effort. Adapting RFS to other platforms supported by Contiki should be easy. We are currently working on porting RFS to the INGA sensor node [2].

ACKNOWLEDGEMENTS

This work has been supported by the European Commission under the contract FP7-ICT-224282 (GINSENG) and by the NTH School for IT Ecosystems.

REFERENCES

[1] H. Dai, M. Neufeld, and R. Han. ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes. In *Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys '04*, page 176, New York, New York, USA, Nov. 2004. ACM Press.
[2] Felix Büsching, Ulf Kulau, and Lars Wolf. INGA - An Inexpensive Node for General Applications. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, page 2, Seattle, 2011. ACM.
[3] Moteiv Corporation. Tmote Sky Datasheet. http://www.snm.ethz.ch/pub/uploads/Projects/tmote_sky_datasheet.pdf, 2006.
[4] T. O'Donovan, J. Brown, U. Roedig, C. Sreenan, J. DoO, A. Dunkles, A. Klein, J. S. Silva, V. Vassiliou, and L. Wolf. GINSENG: Performance Control in Wireless Sensor Networks. In *7th Annual IEEE Communications Society Conference on Sensor Mesh and Ad Hoc Communications and Networks (SECON)*, July 2010.
[5] T. O'Donovan, C. J. Sreenan, N. Tsiftes, Z. He, and T. Voigt. Poster abstract: Storage-centric debugging of performance problems in sensor networks. *7th European Conference on Wireless Sensor Networks, 17-19 Feb 2010, Coimbra, Portugal.*, 2010.
[6] J. B. P. Suriyachai and U. Roedig. Poster Abstract: A MAC Protocol for Industrial Process Automation and Control. In *In the European Conference on Wireless Sensor Networks (EWSN 2010)*. IEEE, February 2010.
[7] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: modeling and analysis of a three-tier architecture for sparse sensor networks. *Ad Hoc Networks*, 1(2-3):215–233, Sept. 2003.
[8] ST Microelectronics. *M25P80 8 Mbit , Low Voltage , Serial Flash Memory With 25 MHz SPI Bus Interface*, 2002.
[9] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, San Francisco, USA, Apr. 2009.