A Transactional Flash File System for Microcontrollers

Eran Gal and Sivan Toledo*
School of Computer Science, Tel-Aviv University

Abstract

We present a transactional file system for flash memory devices. The file system is designed for embedded microcontrollers that use an on-chip or on-board NOR flash device as a persistent file store. The file system provides atomicity to arbitrary sequences of file system operations, including reads, writes, file creation and deletion, and so on. The file system supports multiple concurrent transactions. Thanks to a sophisticated data structure, the file system is efficient in terms of read/write-operation counts, flash-storage overhead, and RAM usage. In fact, the file system typically uses several hundreds bytes of RAM (often less than 200) and a bounded stack (or no stack), allowing it to be used on many 16-bit microcontrollers. Flash devices wear out; each block can only be erased a certain number of times. The file system manages the wear of blocks to avoid early wearing out of frequently-used blocks.

1 Introduction

We present TFFS, a transactional file system for flash memories. TFFS is designed for microcontrollers and small embedded systems. It uses extremely small amounts of RAM, performs small reads and writes quickly, supports general concurrent transactions and single atomic operations, and recovers from crashes reliably and quickly. TFFS also ensures long device life by evening out the wear of blocks of flash memory (flash memory blocks wear out after a certain number of erasures).

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). Flash memory is nonvolatile (retains its content without power), so it is used to store files and other persistent objects in workstations and servers (for the BIOS), in handheld computers and mobile phones, in digital cameras, and in portable music players.

The read/write/erase behavior of flash memory is radically different than that of other programmable memories, such as volatile RAM and magnetic disks. Perhaps more importantly, memory cells in a flash device (as well as in other types of EEPROMs) can be written to only a limited number of times, between 10,000 and 1,000,000, after which they wear out and become unreliable.

Flash memories come in three forms: on-chip memories in system-on-a-chip microcontrollers, standalone chips for board-level integration and removable memory devices (USB sticks, SmartMedia cards, CompactFlash cards, and so on). The file system that we present in this paper is designed for system-on-a-chip microcontrollers that include flash memories and for on-board standalone chips. The file system is particularly suited for devices with very little RAM (system-on-a-chip microcontrollers often include only 1-2 KB of RAM).

Flash memories also come in several flavors with respect to how reads and writes are performed (see, e.g. [1, 2]). The two main categories are NOR flash, which behaves much like a conventional EEPROM device, and NAND flash, which behaves like a block device. But even within each category there are many flavors, especially with respect to how writes are performed. Our file system is designed for NOR flash, and in particular, for devices that are *memory mapped and allow reprogramming*. That is, our file system assumes that four flash operations are available:

- Reading data through random-access memory instructions.
- Erasing a block of storage; in flash memories and EEPROM, erasing a block sets all the bits in the block to a logical '1'.
- Clearing one or more bits in a word (usually 1-4 bytes) consisting of all ones. This is called programming.
- Clearing one or more bits in a word with some zero bits. This is called reprogramming the word.

Virtually all NOR devices support the first three operations, and many support all four, but some do not support reprogramming.

Because flash memories, especially NOR flash, have very different performance characteristics than magnetic disks, file systems designed for disks are usually not appropriate for flash. Flash memories are random access devices that do not benefit at all from access patterns with temporal locality (rapid repeated access to the same location). NOR flash memories do not benefit from spatial locality in read and write accesses; some benefit from sequential access to blocks of a few bytes. Spatial locality is important in erasures—performance is best when much of the data in a block becomes obsolete roughly at the same time.

The only disk-oriented file systems that addresses at least some of these issues are log-structured file systems [3, 4], which indeed have been used on flash devices, often with flash-specific adaptations [5, 6, 7, 8]. But even log-structured file systems ignore some of the features of flash devices, such as the ability to quickly write a small chunk of data anywhere in the file system. As we shall demonstrate in this paper, by exploiting flash-specific features we obtain a much more efficient file system.

File systems designed for small embedded systems must contend with another challenge, extreme scarcity of resources, especially RAM. Many of the existing flash file systems need large amounts of RAM, usually at least tens of kilobytes, so they are not suitable for small microcontrollers (see, e.g., [9, 10]; the same appears to be true for TargetFFS, `www.blunkmicro.com`, and for smxFFS, `www.smxinfo.com`). Two recent flash file systems for TinyOS, an experimental operating system for sensor-network nodes, Matchbox [11, 12] and ELF [8], are designed for microcontrollers with up to 4 KB of RAM.

TFFS is a newly-designed file system for flash memories. It is efficient, ensures long device life, and supports general transactions. Section 3 details the design goals of TFFS, Section 4 explains its design, and Section 5 demonstrates, experimentally, that it does meet its quantitative goals. The design of TFFS is unique; in particular, it uses a new data structure, pruned versioned trees, that we developed specifically for flash file systems. Some of our goals are also novel, at least for embedded systems. Supporting general transactions is not a new idea in file systems [13, 14], but it is not common either; but general transactions were never supported in flash-specific file systems. Although transactions may seem like a luxury for small embedded systems, we believe that transaction support by the file system can simplify many applications and contribute to the reliability of embedded systems.

2 Flash Memories

This section provides background information on flash memories. For further information on flash memories, see, for example, [1] or [2]. We also cover the basic principles of flash-storage management, but this section does not survey flash file systems, data structure and algorithms; relevant citations are given throughout the paper. For an exhaustive coverage of these techniques, see our recent survey [15].

Flash memories are a type of electrically-erasable programmable read-only memory (EEPROM). EEPROM devices store information using modified MOSFET transistors with an additional floating gate. This gate is electrically isolated from the rest of the circuit, but it can nonetheless be charged and discharged using a tunneling and/or a hot-electron-injection effect.

Traditional EEPROM devices support three types of operations. The device is memory mapped, so reading is performed using the processor's memory-access instructions, at normal memory-access times (tens of nanoseconds). Writing is performed using a special on-chip controller, not using the processor's memory-access instructions. This operation is usually called *programming*, and takes much longer than reading; usually a millisecond or more. Programming can only clear set bits in a word (flip bits from '1' to '0'), but not vice versa. Traditional EEPROM devices support *reprogramming*, where an already programmed word is programmed again. Reprogramming can only clear additional bits in a word. To set bits, the word must be *erased*, an operation that is also carried out by the on-chip controller. Erasures often take much longer than even programming, often half a second or more. The word size of traditional EEPROM, which controls the program and erase granularities, is usually one byte.

Flash memories, or more precisely, flash-erasable EEPROMs, were invented to circumvent the long erase times of traditional EEPROM, and to achieve denser layouts. Both goals are achieved by replacing the byte-erase feature by a block-erase feature, which operates are roughly the same speed, about 0.5–1.5 seconds. That is, flash memories also erase slowly, but each erasure erases more bits. Block sizes in flash memories can range from as low as 128 bytes to 64 KB. In this paper, we call these erasure blocks *erase units*.

Many flash devices, and in particular the devices for which TFFS is designed, differ from traditional EEPROMs only in the size of erase units. That is, they are memory mapped, they support fine-granularity programming, and they support reprogramming. Many of the flash devices that use the so-called NOR organization support all of these features, but some NOR devices do not support reprogramming. In general, devices that store more than one bit per transistor (MLC devices) rarely support reprogramming, while single-bit per transistor often do, but other factors may also affect reprogrammability.

Flash-device manufacturers offer a large variety of different devices with different features. Some devices support additional programming operations that program several words at a time; some devices have uniform-size erase blocks, but some have blocks of several sizes and/or multiple banks, each with a different block size; devices with NAND organization are essentially block devices—read, write, and erase operations are performed by a controller on fixed-length blocks. A single file-system design is unlikely to suit all of these devices. TFFS is designed for the most type of flash memories that are used in system-on-a-chip microcontrollers and

low-cost standalone flash chips: reprogrammable NOR devices.

Storage cells in EEPROM devices wear out. After a certain number of erase-program cycles, a cell can no longer reliably store information. The number of reliable erase-program cycles is random, but device manufacturers specify a guaranteed lower bound. Due to wear, the life of flash devices is greatly influenced by how it is managed by software: if the software evens out the wear (number of erasures) of different erase units, the device lasts longer until one of the units wears out.

Flash devices are used to store data objects. If the size of the data objects matches the size of erase units, then managing the device is fairly simple. A unit is allocated to an object when it is created. When the data object is modified, the modified version is first programmed into an erased unit, and then the previous copy is erased. A mapping structure must be maintained in RAM and/or flash to map application objects to erase units. This organization is used mostly in flash devices that simulate magnetic disks—such devices are often designed with 512-byte erase units that each stores a disk sector.

When data objects are smaller than erase units, a more sophisticated mechanism is required to reclaim space. When an object is modified, the new version is programmed into a not-yet-programmed area in some erase unit. Then the previous version is marked as obsolete. When the system runs out of space, it finds an erase unit with some obsolete objects, copies the still-valid objects in that unit to free space on other units, and erases the unit. The process of moving the valid data to other units, modifying the object mapping structures, and erasing a unit is called *reclamation*. The data objects stored on an erase unit can be of uniform size, or of variable size [16, 17].

3 Design Goals

We designed TFFS to meet the requirements of small embedded systems that need a general-purpose file system for NOR flash devices. Our design goals, roughly in order of importance, were

- Supporting the construction of highly-reliable embedded applications,
- Efficiency in terms of RAM usage, flash storage utilization, speed, and code size,
- High endurance.

Supporting general-purpose file-system semantics, such as the POSIX semantics, was not one of our goals. In particular, we added functionality that POSIX file systems do not support when this functionality served our goals, and we do not support some POSIX features that would have reduced the efficiency of the file system.

The specification of the design goals of TFFS was driven by an industrial partner with significant experience in operating systems for small embedded systems. The industrial partner requested support for explicit concurrent transactions, requested that RAM usage be kept to a minimum, and designed with us the API of TFFS. We claim that this involvment of the industrial partner in the specification of TFFS demonstrates that our design goals serve genuine industrial needs and concerns.

Embedded applications must contend with sudden power loss. In any system consisting of both volatile and nonvolatile storage, loss of power may leave the file system itself in an inconsistent state, or the application's files in an inconsistent state from the application's own viewpoint. TFFS performs all file operations atomically, and the file system always recovers to a consistent state after a crash. Furthermore, TFFS's API supports explicit and concurrent transactions. Without transactions, all but the simplest applications would need to implement an application-specific recovery mechanism to ensure reliability. TFFS takes over that responsibility. The support for concurrent transactions allows multiple concurrent applications on the same system to utilize this recovery mechanism.

Some embedded systems ignore the power loss issue, and as a consequence are simply unreliable. For example, the ECI Telecom B-FOCuS 270/400PR router/ADSL modem presents to the user a dialog box that reads "[The save button] saves the current configuration to the flash memory. Do not turn off the power before the next page is displayed, or else *the unit will be damaged!*". Similarly, the manual of the Olympus C-725 digital camera warns the user that loosing power while the flash-access lamp is blinking could destroy stored pictures.

Efficiency, as always, is a multifaceted issue. Many microcontrollers only have 1–2 KB of RAM, and in such systems, RAM is often the most constrained resource. As in any file system, maximizing the effective storage capacity is important; this usually entails minimizing internal fragmentation and the storage overhead of the file system's data structures. In many NOR flash devices, programming (writing) is much slower than reading, and erasing blocks is even slower. Erasure times of more than half a second are common. Therefore, speed is heavily influenced by the number of erasures, and also by overhead writes (writes other than the data writes indicated by the API). Finally, storage for code is also constrained in embedded systems, so embedded file systems need to fit into small footprints.

In TFFS, RAM usage is the most important efficiency metric. In particular, TFFS never buffers writes, in order

```
TID BeginTransaction();
int CommitTransaction(tid);
int AbortTransaction(tid);
FD  Open(FD parent, uint16 name, tid);
FD  Open(char* long_name, tid);
FD  CreateFile(type, name, long_name[],
        properties, FD parent_dir, tid);
int ReadBinary(file, buffer, length,
        offset, tid);
int WriteBinary(...);
int ReadRecord(file, buffer, length,
        record_number, tid);
int UpdateRecord(...);
int AddRecord(file, buff, length, tid);
int CloseFile(file, tid);
int DeleteFile(file);
```

Figure 1: A slightly simplified version of the API of TFFS. The types `TID` and `FD` stand for transaction identifier and file descriptor (handle), respectively. We do not show the type of arguments when the type is clear from the name (e.g., `char*` for `buffer`, `FD` for `file`, and so on). We also do not show a few utility functions, such as a function to retrieve a file's properties.

not to use buffer space. The RAM usage of TFFS is independent of the number of resources in use, such as open files. This design decision trades off speed for RAM. For small embedded systems, this is usually the right choice. For example, recently-designed sensor-network nodes have only 0.5–4 KB of RAM, so file systems designed for them must contend, like TFFS, with severe RAM constraints [8, 11, 12]. We do not believe that RAM-limited systems will disappear anytime soon, due to power issues and to mass-production costs; even tiny 8-bit microcontrollers are still widely used.

The API of the file system is nonstandard. The API, presented in a slightly simplified form in Figure 1, is designed to meet two main goals: support for transactions, and efficiency. The efficiency concerns are addressed by API features such as support for integer file names and for variable-length record files. In many embedded applications, file names are never presented to the user in a string form; some systems do not have a textual user interface at all, and some do, but present the files as nameless objects, such as appointments, faxes, or messages. Variable-length record files allow applications to efficiently change the length of a portion of a file without rewriting the entire file.

We deliberately excluded some common file-system features that we felt were not essential for embedded system, and which would have complicated the design or would have made the file system less efficient. The most important among these are directory traversals, file truncation, and changing the attributes of a file (e.g., its name). TFFS does not support these features. Directory traversals are helpful for human users; embedded file systems are used by embedded applications, so the file names are embedded in the applications, and the applications know which files exist.

One consequence of the exclusion of these features is that the file system cannot be easily integrated into some operating systems, such as Linux and Windows CE. Even though these operating systems are increasingly used in small embedded systems, such as residential gateways and PDAs, we felt that the penalty in efficiency and code size to support general file-system semantics would be unacceptable for smaller devices.

On the other hand, the support for transactions does allow applications to reliably support features such as long file names. An application that needs long names for files can keep a long-names file with a record per file. This file would maintain the association between the integer name of a file and its long file name, and by creating the file and adding a record to the naming file in a transaction, this application data structure would always remain consistent.

The endurance issue is unique to flash file systems. Since each block can only be erased a limited number of times, uneven wear of the blocks leads to early loss of storage capacity (in systems that can detect and not use worn-out blocks), or to an untimely death of the entire system (if the system cannot function with some bad blocks).

We will show below that TFFS does meet these objectives. We will show that the support for transactions is correct, and we will show experimentally that TFFS is efficient and avoids early wear. Because we developed the API of TFFS with an industrial partner, we believe that the API is appropriate.

4 The Design of the File System

4.1 Logical Pointers and the Structure of Erase Units

The memory space of flash devices is partitioned into *erase units*, which are the smallest blocks of memory that can be erased. TFFS assumes that all erase units have the same size. (In some flash devices, especially devices that are intended to serve as boot devices, some erase units are smaller than others; in some cases the irregularity can be hidden from TFFS by the flash device driver, which can cluster several small units into a single standard-size one, or TFFS can ignore the irregular units.) TFFS reserves one unit for the log, which allows it to perform transactions atomically. The structure of this erase unit is simple: it is treated as an array of fixed-size records, which TFFS

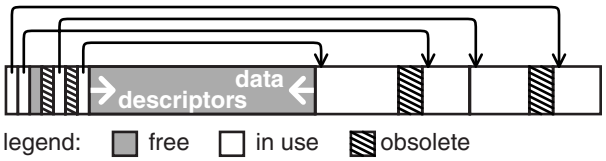

Figure 2: Partitioning an erase unit into variable-length sectors.

always fills in order. The other erase units are all used by TFFS's memory allocator, which uses them to allocate variable-sized blocks of memory that we call *sectors*.

The on-flash data structure that the memory allocator uses is designed to achieve one primary goal. Suppose that an erase unit that still contains valid data is selected for erasure, perhaps because it contains the largest amount of obsolete data. The valid data must be copied to another unit prior to the unit's erasure. If there are pointers to the physical location of the valid data, these pointers must be updated to reflect the new location of the data. Pointer modification poses two problems. First, the pointers to be modified must be found. Second, if these pointers are themselves stored on the flash device, they cannot be modified in place, so the sectors that contain them must be rewritten elsewhere, and pointers to them must now be modified as well. The memory allocator's data structure is designed so that pointer modification is never needed.

TFFS avoids pointer modification by using *logical pointers* to sectors rather than physical pointers; pointers to addresses within sectors are not stored at all. A logical pointer is an unsigned integer (usually a 16-bit integer) consisting of two bit fields: a logical erase unit number and a sector number. When valid data in a unit is moved to another unit prior to erasure, the new unit receives the logical number of the unit to be erased, and each valid sector retains its sector number in the new unit.

A table, indexed by logical erase-unit number, stores logical-to-physical erase-unit mapping. In our implementation the table is stored in RAM, but it can also be stored in a sector on the flash device itself, to save RAM.

Erase units that contain sectors (rather than the log) are divided into four parts, as shown in Figure 2. The top of the unit (lowest addresses) stores a small header, which is immediately followed by an array of sector descriptors. The bottom of the unit contains sectors, which are stored contiguously. The area between the last sector descriptor and the last sector is free. The sector area grows upwards, and the array of sector descriptors grows downwards, but they never collide. Area is not reserved for sectors or descriptors; when sectors are small more area is used by the descriptors than when sectors are large. A sector descriptor contains the erase-unit offset of first address in the sector, as well as a valid bit and an obsolete

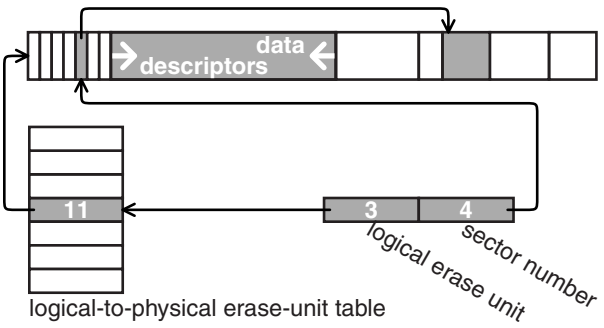

Figure 3: Translating a logical pointer into a physical address. The logical pointer consists of a logical erase unit (3 in the figure) and a sector number (4 in the figure). The logical-to-physical erase-unit table maps logical unit 3 to physical unit 11, which is the erase unit shown in the figure. The pointer stored in descriptor number 4 in erase unit 11 points to the address represented by the logical pointer.

bit. Clearing the valid bit indicates that the offset field has been written completely (this ensures that sector descriptors are created atomically). Clearing the obsolete bit indicates that the sector that the descriptor refers to is now obsolete.

A logical pointer is translated to a physical pointer as shown in Figure 3. The logical erase-unit number within the logical pointer is used as an index into the erase-unit table. This provides the physical unit that contains the sector. Then the sector number within the logical pointer is used to index into the sector-descriptors array on that physical unit. This returns a sector descriptor. The offset in that descriptor is added to the address of the physical erase unit to yield the physical address of the sector.

Before an erase unit is erased, the valid sectors on it are copied to another unit. Logical pointers to these sectors remain valid only if the sectors retain their sector number on the new unit. For example, sector number 6, which is referred to by the seventh sector descriptor in the sector-descriptors array, must be referred to by the seventh sector descriptor in the new unit. The offset of the sector within the erase unit can change when it is copied to a new unit, but the sector descriptor must retain its position. Because all the valid sectors in the unit to be erased must be copied to the same unit, and since specific sector numbers must be available in that unit, TFFS always copies sectors to a fresh unit that is completely empty prior to erasure of another unit. Also, TFFS always compacts the sectors that it copies in order to create a large contiguous free area in the new unit.

TFFS allocates a new sector in two steps. First, it finds an erase unit with a large-enough free area to accommodate the size of the new sector. Our implementation uses

a unit-selection policy that combines a limited-search best-fit approach with a classification of sectors into frequently and infrequently changed ones. The policy attempts to cluster infrequently-modified sectors together in order to improve the efficiency of erase-unit reclamation (the fraction of the obsolete data on the unit just prior to erasure). Next, TFFS finds on the selected unit an empty sector descriptor to refer to the sector. Empty descriptors are represented by a bit pattern of all 1's, the erased state of the flash. If all the descriptors are used, TFFS allocates a new descriptor at the bottom of the descriptors array. (TFFS knows whether all the descriptors are used in a unit; if they are, the best-fit search ensures that the selected unit has space for both the new sector and for a new descriptor).

The size of the sector-descriptors array of a unit is not represented explicitly. When a unit is selected for erasure, TFFS determines the size using a linear downwards traversal of the array, while maintaining the minimal sector offset that a descriptor refers too. When the traversal reaches that location, the traversal is terminated. The size of sectors is not represented explicitly, either, but it is needed in order to copy valid sectors to the new unit during reclamations. The same downwards traversal is also used by TFFS to determine the size of each sector. The traversal algorithm exploits the following invariant properties of the erase-unit structure. Sectors and their descriptors belong to two categories: *reclaimed sectors*, which are copied into the unit during the reclamation of another unit, and *new sectors*, allocated later. *Within each category, sectors with consecutive descriptors are adjacent to each other*. That is, if descriptors i and $j > i$ are both reclaimed or both new, and if descriptors $i + 1, \ldots, j - 1$ all belong to the other category, then sector i immidiately precedes sector j. This important invariant holds because (1) we copy reclaimed sectors from lower-numbered descriptors to higher numbered ones, (2) we always allocated the lowest-numbered free descriptor in a unit for a new sector, and (3) we allocate the sectors themselves from the top down (from right to left in Figure 2). The algorithm keeps track of two descriptor indices, ℓ_r the reclaimed descriptor, and ℓ_n, the last new descriptor. When the algorithm examines a new descriptor i, it first determines whether it is free (all 1's), new or reclaimed. If it is free, the algorithm proceeds to the next descriptor. Otherwise, if the sector lies to the right of the last-reclaimed mark stored in the unit's header, it is reclaimed, otherwise new. Suppose that i is new; sector i starts at the address given by its header, and it ends at the last address before ℓ_n, or the the end of the unit if i is the first new sector encountered so far. The case of reclaimed sectors is completely symmetric. Note that the traversal processes both valid and obsolete sectors.

As mentioned above, each erase unit starts with a header. The header indicates whether the unit is free, used for the log, or for storing sectors. The header contains the logical unit that the physical unit represents (this field is not used in the log unit), and an erase counter. The header also stores the highest (leftmost) sector offset of sectors copied as part of another unit's reclamation process; this field allows us to determine the size of sectors efficiently. Finally, the header indicates whether the unit is used for storing frequently- or infrequently-modified data; this helps cluster related data to improve the efficiency of reclamation. In a file system that uses n physical units of m bytes each, and with an erase counter bounded by g, the size of the erase-unit header in bits is $3+\lceil\log_2 n\rceil+\lceil\log_2 m\rceil+\lceil\log_2 g\rceil$. Flash devices are typically guaranteed for up to one million erasures per unit (and often less, around 100,000), so an erase counter of 24 bits allows accurate counting even if the actual endurance is 16 million erasures. This implies that the size of the header is roughly $27 + \log_2(nm)$, which is approximately 46 bits for a 512 KB device and 56 bits for a 512 MB device.

The erase-unit headers represent an on-flash storage overhead that is proportional to the number of units. The size of the logical-to-physical erase-unit mapping table is also proportional to the number of units. Therefore, a large number of units causes a large storage overhead. In devices with small erase units, it may be advantageous to use a flash device driver that aggregates several physical units into larger ones, so that TFFS uses a smaller number of larger units.

4.2 Efficient Pruned Versioned Search Trees

TFFS uses a novel data structure that we call *efficient versioned search trees* to support efficient atomic file-system operations. This data structure is a derivative of persistent search trees [18, 19], but it is specifically tailored to the needs of file systems. In TFFS, each node of a tree is stored in a variable-sized sector.

Trees are widely-used in file systems. For example, Unix file systems use a tree of indirect blocks, whose root is the inode, to represent files, and many file systems use search trees to represent directories. When the file system changes from one state to another, a tree may need to change. One way to implement atomic operations is to use a *versioned tree*. Abstractly, the versioned tree is a sequence of versions of the tree. Queries specify the version that they need to search. Operations that modify the tree always operate on the most recent version. When a sequence of modifications is complete, an explicit commit operation freezes the most recent version, which becomes read-only, and creates a new read-write

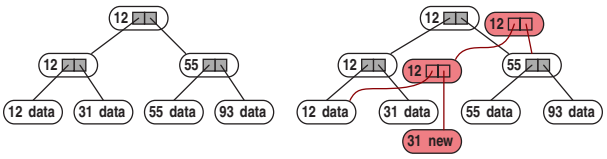

Figure 4: Path copying. Replacing the data associated with the leaf whose key is 31 in a binary tree (left) creates a new leaf (right). The leaf replacement propagates up to the root. The new root represents the new version of the tree. Data structure items that are created as a result of the leaf modification are shown in red.

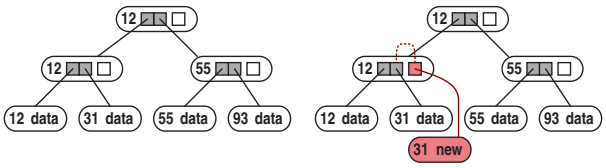

Figure 5: Node copying. Internal tree nodes contain a spare pointer, initially unused (white). Replacing a leaf creates a new leaf and sets the spare pointer in its parent. The spare pointer points to the new leaf, and it also indicates the child pointer that it replaced (dotted line).

version.

When versioned trees are used to implement file systems, usually only the read-write version (the last one) and the read-only version that precedes it are accessed. The read-write version represents the state of the tree while processing a transaction, and the read-only version represents the most-recently committed version. The read-only versions satisfy all the data-structure invariants; the read-write version may not. Because old read-only versions are not used, they can be pruned from the tree, thereby saving space. We call versioned trees that restrict read access to the most recently-committed version *pruned versioned trees*.

The simplest technique to implement versioned trees is called *path copying* [18, 19], illustrated in Figure 4. When a tree node is modified, the modified version cannot overwrite the existing node, because the existing node participates in the last committed version. Instead, it is written elsewhere in memory. This requires a modification in the parent as well, to point to the new node, so a new copy of the parent is created as well. This always continues until the root. If a node is modified twice or more before the new version is committed, it can be modified in place, or a new copy can be created in each modification. If the new node is stored in RAM, it is usually modified in place, but when it is stored on a difficult to modify memory, such as flash, a new copy is created. The log-structured file system [3, 4], for example, represents each file as tree whose root is an inode, and uses this algorithm to modify files atomically. WAFL [20], a file system that supports snapshots, represents the entire file-system as a single tree, which is modified in discrete write episodes; WAFL maintains several read-only versions of the file-system tree to provide users with access to historical states of the file system.

A technique called *node copying* can often prevent the copying of the path from a node to the root when the node is modified, as shown in Figure 5. This technique relies on *spare pointers* in tree nodes, and on nodes that can be physically modified in place. To implement node copying, nodes are allocated with one or more spare pointers, which are initially empty. When a child pointer in a node needs to be updated, the system determines whether the node still contains an empty spare pointer. If it does, the spare pointer is modified instead. The modified spare pointer points to the new child, and it contains an indication of which original pointer it replaces.

Each spare pointer also includes a commit bit, to indicate whether it has been created in the current read-write version or in a previous version. If the commit bit is set, then tree accesses in both the read-only version and the read-write version should traverse the spare pointer, not the original pointer that it replaces. If the commit bit is not yet set, then tree access in the read-write version should traverse the spare pointer but tree access in the read-only version should traverse the original pointer. Spare pointers also have an abort bit; if set, then the spare pointer is simply ignored.

In B-trees, in which nodes have a variable number of child pointers, the spare pointer can also be used to add a new child pointer. This serves two purposes. First, it allows us to allocate variable-size nodes, containing only enough child pointers for the number of children the node has at creation time. Second, it allows us to store original child pointers without commit bits.

In principle, using a number of spare pointers can further reduce the number of node creations, at the expense of more storage overhead. However, even with only one spare, it can be shown that the amortized cost of a single tree update/insert/delete operation is constant. Therefore, our file system always uses only one spare per node.

4.3 Tree Traversals with a Bounded Stack

Small embedded systems often have very limited stack space. Some systems do not use a stack at all: static compiler analysis maps automatic variables to static RAM locations. To support systems with a bounded or no stack, TFFS never uses explicit recursion. We do use recursive algorithms, but the recursion uses a statically-allocated

stack and its depth is configured at compile time. We omit further details.

4.4 Mapping Files and File Names

TFFS uses pruned versioned trees for mapping files and file names. Most of the trees represent files and directories, one tree per file/directory. These trees are versioned.

In record files, each record is stored in a separate sector, and the file's tree maps record numbers to the logical addresses of sectors. In binary files, extents of contiguous data are stored on individual sectors, and the file's tree maps file offsets to sectors. The extents of binary files are created when data is appended to the file. Currently, TFFS does not change this initial partitioning.

TFFS supports two naming mechanisms for the `open` system call. One mechanism is a hierarchical name space of directories and files, as in most file systems. However, in TFFS directory entries are short unsigned integers, not strings, in order to avoid string comparisons in directory searches. The second mechanism is a flat namespace consisting of unique strings. A file or directory can be part of one name space or both. In the future, we may merge these two mechanisms, as explained below. Currently, however, the hierarchical name space does not allow long names.

Therefore, directory trees are indexed by the short integer entry names. The leaves of directory trees are the metadata records of the files. The metadata record contains the internal file identifier (GUID) of the directory entry, as well as the file type (record/binary/directory), the optional long name, permissions, and so on. In TFFS, the metadata is immutable.

TFFS assumes that long names are globally unique. We use a hash function to map these string names to 16-bit integers, which are perhaps not unique. TFFS maps a directory name to its metadata record using a search tree indexed by hash values. The leaves of this tree are either the metadata records themselves (if a hash value maps into a single directory name), or arrays of logical pointers to metadata records (if the names of several directories map into the same hash value).

In the future, we may replace the indexing in directory trees from the explicit integer file names to hash values of long file names.

A second search tree maps GUIDs to file/directory trees. This tree is indexed by GUID and its leaves are logical pointers to the roots of file/directory trees.

The `open` system call comes in two versions: one returns a GUID given a directory name, and the other returns a GUID given a directory GUID and a 16-bit file identifier within that directory. The first computes the hash value of the given name and uses it to search the directory-names tree. When it reaches a leaf, it verifies the directory name if the leaf is the metadata of a directory, or searches for a metadata record with the appropriate name if the leaf is an array of pointers to metadata records. The second variant of the system call searches the GUID tree for the given GUID of the directory. The leaf that this search returns is a logical pointer to the root of the directory tree. The system call then searches this directory tree for the file with the given identifier; the leaf that is found is a logical pointer to the metadata record of the sought-after file. That metadata record contains the GUID of the file.

In file-access system calls, the file is specified by a GUID. These system calls find the root of the file's tree using the GUID tree.

4.5 Transactions on Pruned Versioned Trees

The main data structures of TFFS are pruned versioned trees. We now explain how transactions interact with these trees. By transactions we mean not only explicit user-level transactions, but also implicit transactions that perform a single file-system modification atomically.

Each transaction receives a transaction identifier (TID). These identifiers are integers that are allocated in order when the transaction starts, so they also represent discrete time stamps. A transaction with a log TID time stamp started before a transaction with a higher TID. The file system can commit transactions out of order, but the linearization order of the transactions always corresponds to their TID: a transaction with TID t can observe all the side effects of committed transactions $t-1$ and lower, and cannot observe any of the side effects of transactions $t+1$ and higher.

When a transaction modifies a tree, it creates a new version of the tree. That version remains active, in read-write mode, until the transaction either commits or aborts. If the transaction commits, all the spare pointers that it created are marked as committed. In addition, if the transaction created a new root for the file, the new root becomes active (the pointer to the tree's root, somewhere else in the file system, is updated). If the transaction aborts, all the spare pointers that the transaction created are marked as aborted by a special bit. Aborted spare pointers are not valid and are never dereferenced.

Therefore, a tree can be in one of two states: either with uncommitted and unaborted spare pointers (or an uncommitted root), or with none. A tree in the first state is being modified by a transaction that is not yet committed or aborted. Suppose that a tree is being modified by transaction t, and that the last committed transaction that modified it is s. The read-only version of the tree, consisting of all the original child pointers and all the committed spare pointers, represents the state of the tree

in discrete times r for $s \leq r < t$. We do not know what the state of the tree was at times smaller than s: perhaps some of the committed spares represent changes made earlier, but we cannot determine when, so we do not know whether to follow them or not. Also, some of the nodes that existed at times before s may cease to exist at time s. The read-write version of the tree represents the state of the tree at time t, but only if transaction t will commit. If transaction t will abort, then the state of the tree at time t is the same state as at time s. If transaction t will commit, we still do not know what the state of the tree will be at time $t + 1$, because transaction t may continue to modify it. Hence, the file system allows transactions with TID r, for $s < r < t$, access to the read-only version of the tree, and to transaction t access to the read-write version. All other access attempts cause the accessing transaction to abort. In principle, instead of aborting transactions later than t, TFFS could block them, but we assume that the operating system's scheduler cannot block a request.

If a tree is in the second state, with only committed or aborted spares, we must keep track not only of its last modification time s, but also of the latest transaction $u \geq s$ that read it. The file system admits read requests from any transaction r for $r > s$, and write requests from a transaction $t \geq u$. As before, read requests from a transaction $r < s$ causes r to abort. A write request from a transaction $t < u$ causes t to abort, because it might affect the state of the tree that u already observed.

To enforce these access rules, we associate three TIDs with each versioned tree: the last committed modification TID, the last read-access TID, and the TID that currently modifies the tree, if any. These TIDs are kept in a search tree, indexed by the internal identifier of the versioned tree. The file system never accesses the read-only version of the TID tree. Therefore, although it is implemented as a pruned versioned tree, the file system treats it as a normal mutable search tree. The next section presents an optimization that allows the file system not to store the TIDs associated with a tree.

4.6 Using Bounded Transaction Identifiers

To allow TFFS to represent TIDs in a bounded number of bits, and also to save RAM, the file system represents TIDs modulo a small number. In essence, this allows the file system to store information only on transactions in a small window of time. Older transactions than this window permits must be either committed or aborted.

The TID allocator consists of three simple data structures that are kept in RAM: next TID to allocate, the oldest TID in the TID tree, and a bit vector with one bit per TID within the current TID window. The bit vector stores, for each TID that might be represented in the system, whether it is still active or whether it has already been aborted or committed. When a new TID needs to be allocated, the allocator first determines whether the next TID represents an active transaction. If it does, the allocation simply fails. No new transactions can be started until the oldest one in the system either commits or aborts. If the next TID is not active and not in the TID tree, it is allocated and the next-TID variable is incremented (modulo the window size). If the next TID is not active but it is in the TID tree, the TID tree is first cleaned, and then the TID is allocated.

Before cleaning the TID tree, the file system determines how many TIDs can be cleaned. Cleaning is expensive, so the file system cleans on demand, and when it cleans, it cleans as many TIDs as possible. The number of TIDs that can be cleaned is the number of consecutive inactive TIDs in the oldest part of the TID window. After determining this number, the file system traverses the entire TID tree and invalidates the appropriate TIDs. An invalid TID in the tree represents a time before the current window; transactions that old can never abort a transaction, so the exact TID is irrelevant. We cannot search for TIDs to be invalidated because the TID tree is indexed by the identifiers of the trees, not by TID.

The file system can often avoid cleaning the TID tree. Whenever no transaction is active, the file system deletes the entire TID tree. Therefore, if long chains of concurrent transactions are rare, tree cleanup is rare or not performed at all. The cost of TID cleanups can also be reduced by using a large TID window size, at the expense of slight storage inefficiency.

4.7 Atomic Non-transactional Operations

To improve performance and to avoid running out of TIDs, the file system supports non-transactional operations. Most requests to the file system specify a TID as an argument. If no TID is passed to a system call (the TID argument is 0), the requested operation is performed atomically, but without any serializability guarantees. That is, the operation will either success completely, or will fail completely, but it may break the serializability of concurrent transactions.

The file system allows an atomic operation to modify a file or directory only if no outstanding transaction has already modified the file's tree. But this still does not guarantee serializability. Consider a sequence of operations in which a transaction reads a file, which is subsequently modified by an atomic operation, and then read or modified again by the transaction. It is not possible to serialize the atomic operation and the transaction.

Therefore, it is best to use atomic operations only on files/directories that do not participate in outstanding transactions. An easy way to ensure this is to access a

particular file either only in transactions or only in atomic operations.

Atomic operations are more efficient than single-operation transactions in two ways. First, during an atomic operation the TID tree is read, to ensure that the file is not being modified by an outstanding transaction, but the TID tree is not modified. Second, a large number of small transactions can cause the file system to run out of TIDs, if an old transaction remains outstanding; atomic operations avoid this possibility, because they do not use TIDs at all.

4.8 The Log

TFFS uses a log to implement transactions, atomic operations, and atomic maintenance operations. As explained above, the log is stored on a singe erase unit as an array of fixed-size records that grows downwards. The erase unit containing the log is marked as such in its header. Each log entry contains up to four items: a valid/obsolete bit, an entry-type identifier, a transaction identifier, and a logical pointer. The first two are present in each log entry; the last two remain unused in some entry types.

TFFS uses the following log-record types:

- New Sector and New Tree Node. These record types allow the system to undo a sector allocation by marking the pointed-to sector as obsolete. The New-Sector record is ignored when a transaction is committed, but a The New-Tree-Node record causes the file system to mark the spare pointer in the node, if used, as committed. This ensures that a node that is created in a transaction and modified in the same transaction is marked correctly.
- Obsolete Sector. Sectors are marked as obsolete only when the transaction that obsoleted them is committed. This node is ignored at abort time, and clears the obsolete bit of the sector at commit time.
- Modified Spare Pointer. Points to a node whose spare pointer has been set. Clears the spare's commit bit at commit time or its abort bit at abort time.
- New File. Points to the root of a file tree that was created in a transaction. At commit time, this record causes the file to be added to the GUID tree and to the containing directory. Ignored at abort time.
- File Root. Points to the root of a file tree, if the transaction created a new root. At commit time, the record is used to modify the file's entry in the GUID tree. Ignored at abort time.
- Commit Marker. Ensures that the transaction is redone at boot time.
- Erase Marker. Signifies that an erase unit is about to be erased. The record contains a physical erase-unit number and an erase count, but does not contain a sector pointer or a TID. This record typically uses two fixed-size record slots. If the log contains a non-obsolete erase-marker record at boot time, the physical unit is erased again; this completes an interrupted erasure.
- GUID-Tree Pointer, TID-Tree Pointer, and Directory-Hash-Tree Pointer. These records are written to the log when the root of one of these trees moves, to allow the file system to find them at boot time.

File trees are modified during transactions and so does the TID tree. The GUID and directory-hash trees, and directory trees, however, are only modified during commits. We cannot modify them during transactions because our versioned trees only support one outstanding transaction. Delaying the tree modification to commit time allows multiple outstanding transactions to modify a single directory, and allows multiple transactions to create files and directories (these operations affect the GUID and the directory-hash trees). TFFS does not allow file and directory deletions to be part of explicit transactions because that would have complicated the file/directory creation system calls.

The delayed operations are logged but not actually performed on the trees. After the commit system call is invoked, but before the commit marker is written to the log, the delayed operations are performed.

When a transaction accesses a tree whose modification by the same transaction may have been delayed, the tree access must scan the log to determine the actual state of the tree, from the viewpoint of that transaction. Many of these log scans are performed in order to find the roots of files that were created by the transaction or whose root was moved by the transaction. To locate the roots of these file trees more quickly, the file system keeps a cache of file roots that were modified by the transaction. If a file that is accesses is marked in the TID tree as being modified by the transaction, the access routine first checks this cache. If the cache contains a pointer to the file's root, the search in the log is avoided; otherwise, the log is scanned for a non-obsolete file-root record.

5 Implementation and Performance

This section describes the implementation of the file system and its performance. The performance evaluation is based on detailed simulations that we performed using several simulated workloads. The simulations measure the performance of the file system, its storage overheads, its endurance, and the cost of leveling the device's wear.

5.1 Experimental Setup

This section describes the experimental setup that we used for the simulations.

Devices. We performed the experiments using simulations of two real-world flash devices. The first is an 8 MB stand-alone flash-memory chip, the M29DW640D from STMicroelectronics. This device consists of 126 erase units of 64 KB each (and several smaller ones, which our file system does not use), read access times of about 90 ns, program times of about 10 us, and block-erase times of about 0.8 seconds.

The second device is a 16-bit microcontroller with on-chip flash memory, the ST10F280, also from STMicroelectronics. This chip comes with two banks of RAM, one containing 2 KB and the other 16 KB, and 512 KB of flash memory. The flash memory contains 7 erase units of 64 KB each (again, with several smaller units that we do not use). The flash access times are 50 ns for reads, 16 us for writes, and 1.5 seconds erases. The small number of erase units in this chip hurts TFFS's performance; to measure the effect, we also ran simulations using this device but with smaller erase units ranging from 2 to 32 KB.

Both devices are guaranteed for 100,000 erase cycles per erase unit.

File-System Configuration. We configured the non-hardware-related parameters of the file system as follows. The file system is configured to support up to 32 concurrent transactions, B-tree nodes have either 2–4 children or 7–14 children, 10 simulated recursive-call levels, and a RAM cache of 3 file roots. This configuration requires 466 bytes of RAM for the 8 MB flash and 109 bytes for the 0.5 MB flash.

Workloads. We used 3 workloads typical of flash-containing embedded systems to evaluate the file system. The first workload simulates a fax machine. This workload is typical not only of fax machines, but of other devices that store fairly large files, such as answering machines, dictating devices, music players, and so on. The workload also exercises the transactions capability of the file system. This workload contains:

- A parameter file with 30 variable length records, ranging from 4 to 32 bytes (representing the fax's configuration). This file is created, filled, and is never touched again.
- A phonebook file with 50 fixed-size records, 32 bytes each. This file is also created and filled but never accessed again.
- Two history files consisting of 200 cyclic fixed-size records each. They record the last 200 faxes sent and 200 last faxes received. They are changed whenever a fax page is sent or received.
- Each arriving fax consists of 4 pages, 51,300 bytes each. Each page is stored in a separate file and the pages of each fax are kept in a separate directory that is created when the fax arrives. The arrival of a fax triggers a transaction that creates a new record in the history file and creates a new directory for the file. The arrival of every new fax page adds changes the fax's record in the history file and creates a new file. Data is written to fax-page files in blocks of 1024 bytes.
- The simulation does not include sending faxes.

We also ran experiments without transactions under this workload, in order to assess the extra cost of transactions. We did not detect any significant differences when no transactions were used, so we do not present these results in the paper.

The second workload simulates a cellular phone. This simulation represents workloads that mostly store small files or small records, such as beepers, text-messaging devices, and so on. This workload consists of the following files and activities:

- Three 20-record cyclic files with 15-byte records, one for the last dialed numbers, one for received calls, and one for sent calls.
- Two SMS files, one for incoming messages and one for outgoing messages. Each variable-length record in these files stores one message.
- An appointments file, consisting of variable-length records.
- An address book file, consisting of variable-length records.
- The simulation starts by adding to the phone 150 appointments and 50 address book entries.
- During the simulation, the phone receives and sends 3 SMS messages per day (3 in each direction), receives 10 and dials 10 calls, and misses 5 calls, adds 5 new appointments and deletes the oldest 5 appointments.

The third workload simulates an event recorder, such as a security or automotive "black box", a disconnected remote sensor, and so on. The simulation represents workloads with a few event-log files, some of which record frequent events and some of which record rare events (or perhaps just the extreme events from a high-frequency event stream). This simulation consists of three files:

- One file records every event. This is a cyclic file with 32-byte records.

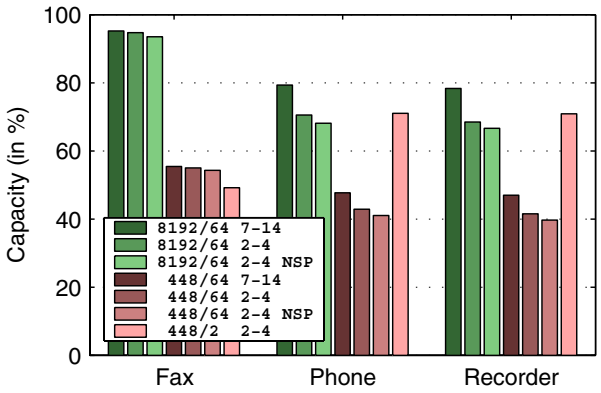

Figure 6: The capacity of TFFS. For each workload, the graph shows 7 bars: 3 for an 8 MB flash with 64 KB erase units (denoted by 8192/64) and 4 bars for a 448 KB flash with either 64 or 2 KB erase units (448/64 and 448/2). Two bars are for file systems whose B-trees have 7–14 children, and the rest for B-trees with 2–4 children. The scenarios denoted NSP describe a file system which does not use spare pointers.

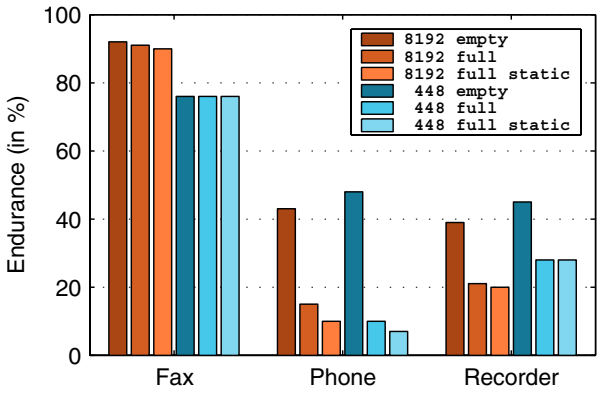

Figure 7: Endurance under different contents scenarios. For each flash size, the graph shows the endurance of a file system that is always almost empty, for a file system that is always almost full and half its data is static, and for a full file system with almost only static data.

- The other file records one event per 10 full cycles through the other files. This file too is cyclic with 32-byte records.
- A configuration file with 30 variable-size records ranging from 4 to 32 bytes. These files are filled when the simulation starts and never accessed again.

5.2 Capacity Experiments

Figure 6 presents the results of experiments intended to measure the storage overhead of TFFS. In these simulations, we initialize the file system and then add data until the file system runs out of storage. In the fax workload, we add 4-page faxes to the file system until it fills. In the phone workload, we do not erase SMS messages. In the event-recording simulation, we replace the cyclic files by non-cyclic files.

The graph shows the amount of user data written to the file system before it ran out of flash storage, as a percentage of the total capacity of the device. For example, if 129,432 bytes of data were written to a flash file system that uses a 266,144 bytes flash, the capacity is 49%.

The groups of bars in the graph represent different device and file-system configurations: an 8 MB device with 64 KB erase units, a 448/64 KB device, and a 448/2 KB device; file systems with 2–4 children per tree node and file systems with 7–14 children; file systems with spare pointers and file systems with no spare pointers.

Clearly, storing large extents, as in the fax workload, reduces storage overheads compared to storing small records or extents. Wider tree nodes reduce overheads when the leaves are small. The performance- and endurance-oriented experiments that we present later, however, indicate that wider nodes degrade performance and endurance. A small number of erase units leads to high overheads. Small erase units reduce overheads except in the fax workload, in which the 1 KB extents fragment the 2 KB erase units.

5.3 Endurance Experiments

The next set of experiments measures both endurance, which we present here, and performance, which we present in the next section. All of these experiments run until one of the erase units reaches an erasure count of 100,000; at that point, we consider the device worn out. We measure endurance by the amount of user data written to the file system as a percentage of the theoretical endurance limit of the device. For example, a value of 68 means that the file system was able to write 68% of the data that can be written on the device if wear is completely even and if only user data is written to the device.

We performed two groups of experiment. The first assesses the impact of file-system fullness and data life spans on TFFS's behavior. In particular, we wanted to understand how TFFS copes with a file system that is almost full and with a file system that contains a significant amount of static data. This group consists of three scenarios: one scenario in which the file system remains mostly empty; one in which is it mostly full, half the data is never deleted or updated, and the other half is updated cyclically; and one in which the file system is mostly full, most of the data is never updated, and a small portion is updated cyclically. The results of these endurance exper-

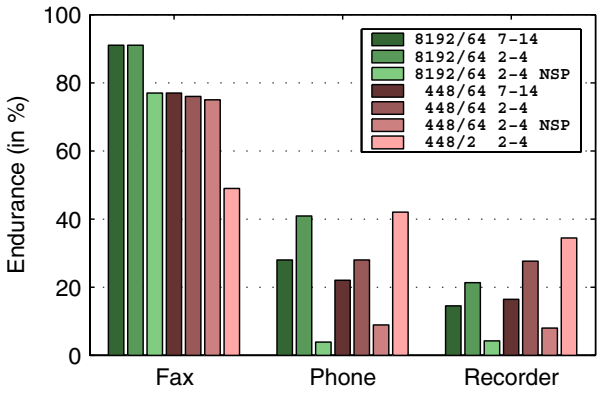

Figure 8: Endurance under different device and file-system configurations.

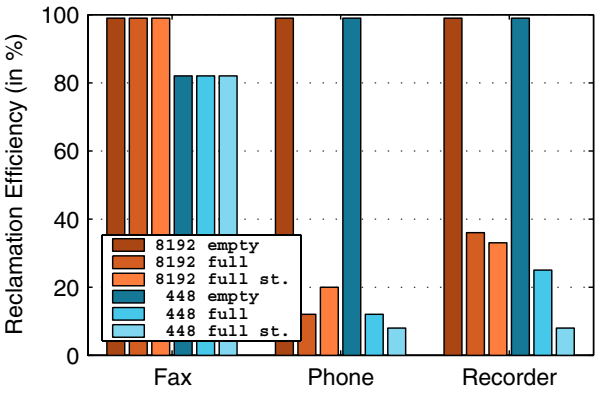

Figure 9: Reclamation efficiency under different content scenarios.

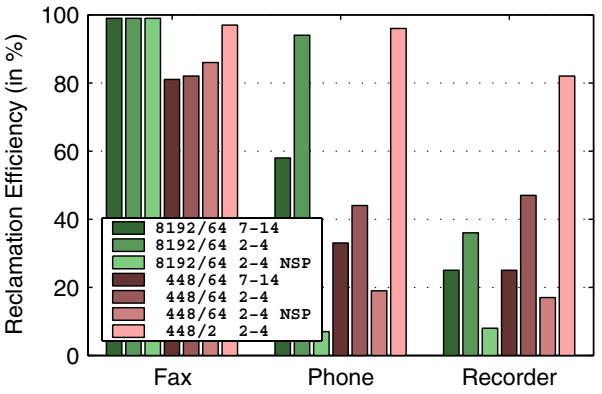

Figure 10: Reclamation efficiency under different device/file-system scenarios.

iments are shown in Figure 7.

The other group of experiments assesses the impact of device characteristics and file-system configuration on TFFS's performance. This group includes the same device/file-system configurations as in the capacity experiments, but the devices were kept roughly two-thirds full, with half of the data static and the other half changing cyclically. The results of this group of endurance experiments are shown in Figure 8.

The graphs show that on the fax workload, endurance is good, almost always above 75% and sometimes above 90%. On the two other workloads endurance is not as good, never reaching 50%. This is caused not by early wear of a particular block, but by a large amount of file-system structures written to the device (because writes are performed in small chunks). The endurance of the fax workload on the device with 2 KB erase units is relatively poor because fragmentation forces TFFS to erase units that are almost half empty. The other significant fact that emerges from the graphs is that the use of spare pointers significantly improves endurance (and performance, as we shall see below).

5.4 Performance Experiments

The next set of experiments is designed to measure the performance of TFFS. We measured several performance metrics under the different content scenarios (empty, full-half-static, and full-mosly-static file systems) and the different device/file-system configuration scenarios.

The first metric we measured was the average number of erasures per unit of user-data written. That is, on a device with 64 KB erase units, the number of erasures per 64 KB of user data written. The results were almost exactly the inverse of the endurance ratios (to within 0.5%). This implies that the TFFS wears out the devices almost completely evenly. When the file system performs few erases per unit of user data written, both performance and endurance are good. When the file system erases many units per unit of user data written, both metrics degrade. Furthermore, we have observed no cases where uneven wear leads to low endurance; low endurance is always correlated with many erasures per unit of user data written.

The second metric we measured was the efficiency of reclamations. We define this metric as the ratio of user data to the total amount of data written in block-write operations. The total amount includes writing of data to sectors, both when a sector is first created and when it is copied during reclamation, and copying of valid log entries during reclamation of the log. The denominator does not include writing of sector descriptors, erase-unit headers, and modifications of fields within sectors (fields such as spare pointers). A ratio close to 100% implies that little data is copied during reclamations, whereas a low ratio indicates that a lot of valid data is copied during reclamation. The two graphs presenting this metric,

Figure 9 and Figure 10, show that the factors that affect reclamation efficiency are primarily fullness of the file system, the amount of static data, and the size of user data items. The results again show that spare pointers contribute significantly to high performance.

We also measured two other metrics: the number of programming operations per system call and the number of flash-read instructions per system call. These metrics do not count programming and read operations performed in the context of copying blocks; these are counted by the reclamation-efficiency metric. These metrics did not reveal any interesting behavior rather than to show (again) that spare pointers improve performance. Spare pointers improve these metrics by more than a factor of 2.

6 Summary

We have presented several contributions to the area of embedded file system, especially file systems for memory-mapped flash devices.

Some of our design goals are novel. We have argued that even embedded file systems need to be recoverable (journaled), and that they should support general transactions, in order to help programmers construct reliable applications. We have also argued that in many cases, embedded file systems do not need to offer full POSIX or Windows semantics and that supporting these general semantics is expensive. The other design goals that we have attempted to achieve, such as high performance and endurance, are, of course, not novel.

We have shown that supporting recoverability and transactions is not particularly expensive. Only time will tell whether these features will actually lead to more reliable systems, but we believe that the anecdotal evidence that we presented in Section 3 (the unreliable modem/router) is typical and that due to lack of file-system support for atomicity, many embedded systems are unreliable. We have not measured the cost of supporting more general semantics.

Another area of significant contribution is the design of the data structures that TFFS uses, especially the design of the pruned versioned B-trees. This design borrows ideas from both research on persistent data structures [18, 19] and from earlier flash file systems. We have adapted the previously-proposed persistent search trees to our needs: our trees can cluster many operations on a single tree into a single version, and our algorithms prune old versions from the trees. Spare pointers are related to replacement pointers that were used in the notoriously-inefficient Microsoft Flash File System [16, 21, 22, 23, 24, 25], and to replacement block maps in the Flash Translation Layer [26, 27, 28]. But again, we have adapted these ideas: in Microsoft's FFS, paths of replacement pointers grew and grew; in TFFS, spare pointers never increase the length of paths. The replacement blocks in the Flash Translation Layer are designed for patching elements in a table, whereas our replacement pointes are designed for a pointer-based data structure.

The performance metrics that we used to evaluate the file system are also innovative. To the best of our knowledge, most of them have never been used in the literature. The characteristics of flash are different from those of other persistent storage devices, such as traditional EEPROM and magnetic disks. Therefore, flash-specific metrics are required in order to assess and compare flash file systems. The metrics that we have introduced, such as endurance metrics and metrics that emphasize writes over reads, allow both users and file-system implementers to assess and compare file systems. We expect that additional research will utilize these metrics, perhaps even to quantitatively show that future file systems are better than TFFS.

Finally, our performance results show that TFFS does achieve its design goals, but they also point out to weaknesses. On devices with many erase units, TFFS performs very well, but it does not perform well on devices with very few erase units. This issue can be addressed either by avoiding devices with few units in TFFS-based systems, or by improving TFFS to better exploit such devices. Also, like other flash management software that manages small chunks of data on large erase units, TFFS performs poorly when the devices is nearly full most of the time and contains a lot of static data.

We conjecture that some of the weaknesses in TFFS can be addressed by better file-system policies, perhaps coupled with a re-clustering mechanism. In particular, is is likely that a better allocation policy (on which erase unit to allocate sectors) and a better reclamation policy (which erase unit to reclaim) would improve endurance and performance. A re-clustering mechanism would allow TFFS to copy sectors from an erase unit being reclaimed into two or more other units; currently, the structure of logical pointers does not allow re-clustering. The allocation, reclamation, and re-clustering policies that were developed in two contexts might be applicable to TFFS. Such policies have been investigated for management of fixed-sized blocks on flash memories [5, 29, 30, 31], and for log-structured file system [3, 32, 33, 34]. Clearly not all of these techniques are applicable to flash file systems, but some of them might be. This remains an area for future research.

The design of TFFS brings up an important issue: file systems for NOR flash can write very efficiently small amounts of data, even if these writes are performed atomically and committed immediately. Writes to NOR flash are not performed in large blocks, so the time to per-

form a write operation to the file system can be roughly proportional to the amount of data written, even for small amounts. This observation is not entirely new: proportional write mechanisms have been used in Microsoft's FFS2 [16, 22, 23, 24, 25] and in other linked-list based flash file systems [35]. But these file systems suffered from performance problems and were not widely used [21]. More recent file systems tend to be block based, both in order to borrow ideas from disk file systems and in order to support NAND flash, in which writes are performed in blocks. TFFS shows that in NOR flash, it is possible to benefit from cheap small writes, without incurring the performance penalties of linked-list based designs. The same observation also led other researchers to propose flash-based application-specific persistent data structures [36, 37].

It is interesting to compare TFFS to Matchbox [11, 12] and ELF [8], two file systems for sensor-network nodes. Both are designed for the same hardware, sensor nodes with a NAND (page-mode) flash and up to 4 KB of RAM. Matchbox offers limited functionality (sequential reads and writes, a flat directory structure) and is not completely reliable. ELF offers more functionality, including random access and hierarchical directories. It appears to be more reliable than Matchbox, but still not completely reliable. ELF uses an in-RAM linked list to represent open files; these lists can be quite long if the file is long or has been updated repeatedly. It seems that some of the reliability and performance issues in ELF result from the use of NAND flash; we believe that NOR flash is more appropriate for file systems for such small embedded systems.

Acknowledgments Thanks to Yossi Shacham, Eli Luski, and Shay Izen for helpful discussion regarding flash hardware characteristics. Thanks to Andrea Arpaci-Dusseau of the USENIX program committee and to the three anonymous referees for numerous helpful comments and suggestions.

References

[1] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash Memories*. Kluwer, 1999.

[2] N. A. Takahiro Ohnakado and, "Review of device technologies of flash memories," *IEICE Transactions on Electronics*, vol. E84-C, no. 6, pp. 724–733, 2001.

[3] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[4] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *USENIX Winter 1993 Conference Proceedings*, San Diego, California, Jan. 1993, pp. 307–326. [Online]. Available: citeseer.ist.psu.edu/seltzer93implementation.html

[5] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, Jan. 1995, pp. 155–164. [Online]. Available: http://www.usenix.org/publications/library/proceedings/neworl/kawaguchi.html

[6] H.-J. Kim and S.-G. Lee, "An effective flash memory manager for reliable flash memory space management," *IEICE Transactions on Information and Systems*, vol. E85-D, no. 6, pp. 950–964, 2002.

[7] K. M. J. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM systems," U.S. Patent 6 081 447, June 27, 2000.

[8] H. Dai, M. Neufeld, and R. Han, "ELF: An efficient log-structured flash file system for wireless micro sensor nodes," in *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004, pp. 176–187.

[9] D. Woodhouse, "JFFS: The journaling flash file system," July 2001, presented in the Ottawa Linux Symposium, July 2001 (no proceedings); a 12-page article available online from http://sources.redhat.com/jffs2/jffs2.pdf.

[10] (2002) YAFFS: Yet another flash filing system. Aleph One. Cambridge, UK. [Online]. Available: http://www.aleph1.co.uk/yaffs/index.html

[11] D. Gay, "Design of matchbox, the simple filing system for motes (version 1.0)," Aug. 2003, available online from http://www.tinyos.net/tinyos-1.x/doc/.

[12] ——, "Matchbox: A simple filing system for motes (version 1.0)," Aug. 2003, available online from http://www.tinyos.net/tinyos-1.x/doc/.

[13] B. Liskov and R. Rodrigues, "Transactional file systems can be fast," in *11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[14] M. Seltzer and M. Stonebraker, "Transaction support in read optimized and write optimized file systems," in *Proceedings of the 16th Conference on Very Large Databases*, Brisbane, 1990.

[15] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," July 2004, submitted to the *ACM Computing Surveys*.

[16] P. Torelli, "The Microsoft flash file system," *Dr. Dobb's Journal*, pp. 62–72, Feb. 1995. [Online]. Available: http://www.ddj.com

[17] S. Wells, R. N. Hasbun, and K. Robinson, "Sector-based storage device emulator having variable-sized sector," U.S. Patent 5 822 781, Oct. 13, 1998.

[18] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.

[19] H. Kaplan, "Persistent data structures," in *Handbook of Data Structures and Applications*, D. P. Mehta and S. Sahni, Eds. CRC Press, 2004.

[20] D. Hitz, J. Lau, , and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of the 1994 Winter USENIX Technical Conference*, San Francisco, CA, Jan. 1994, pp. 235–245.

[21] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," in *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, Nov. 1994, pp. 25–37. [Online]. Available: citeseer.ist.psu.edu/douglis94storage.html

[22] P. L. Barrett, S. D. Quinn, and R. A. Lipe, "System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers," U.S. Patent 5 392 427, Feb. 21, 1995.

[23] W. J. Krueger and S. Rajagopalan, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 634 050, May 27, 1999.

[24] ——, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 898 868, Apr. 27, 1999.

[25] ——, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 6 256 642, July 3, 2001.

[26] A. Ban, "Flash file system," U.S. Patent 5 404 485, Apr. 4, 1995.

[27] ——, "Flash file system optimized for page-mode flash technologies," U.S. Patent 5 937 425, Aug. 10, 1999.

[28] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Application Note 648, 1998.

[29] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *The Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[30] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software—Practice and Experience*, vol. 29, no. 3, 1999.

[31] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," in *Proceedings of the 6th international conference on Architectural support for programming languages and operating systems*. ACM Press, 1994, pp. 86–97.

[32] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995, pp. 277–288.

[33] J. Wang and Y. Hu, "A novel reordering write buffer to improve write performance of log-structured file systems," *IEEE Transactions on Computers*, vol. 52, no. 12, pp. 1559–1572, Dec. 2003.

[34] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," in *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, Oct. 1997, pp. 238–251.

[35] N. Daberko, "Operating system including improved file management for use in devices utilizing flash memory as main memory," U.S. Patent 5 787 445, July 28, 1998.

[36] C.-H. Wu, L.-P. Chang, and T.-W. Kuo, "An efficient B-tree layer for flash-memory storage systems," in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Tainan City,Taiwan, Feb. 2003, 20 pages.

[37] ——, "An efficient R-tree implementation over flash-memory storage systems," in *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems*. ACM Press, 2003, pp. 17–24.