An Embedded Storage Framework Abstracting Each Raw Flash Device as An MTD

Wei Wang Deng Zhou
San Diego State University
{wang, zhoud}@rohan.sdsu.edu

Tao Xie
San Diego State University
txie@mail.sdsu.edu

Abstract

Existing embedded flash storage systems are built based on a single MTD (Memory Technology Device) architecture no matter how many raw flash devices exist under a flash controller. The single-MTD architecture impedes exploiting device-level parallelism to further improve the performance of a storage system. In this paper, we design and implement a new embedded flash storage framework called MA (MTD-array), which abstracts each underlying raw flash device as an independent MTD device to boost performance. To verify its effectiveness, we implement a new flash file system called MA-UBIFS by incorporating UBIFS, one of the best contemporary flash file systems, into MA in Ubuntu 13.04 with 3.8.0 kernel. Simulation results from real-world applications show that MA-UBIFS outperforms UBIFS in mean response time by up to 71.6%. Further, we build an FPGA evaluation platform. Results from the hardware platform show that on average MA-UBIFS improves write and read throughput in a 2-MTD scenario by 55.2% and 84%, respectively.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Design studies; D.4.2 [*Operating Systems*]: Storage Management

Keywords Embedded system, NAND, file system, MTD.

1. Introduction

There are two options to utilize NAND flash memory devices: a flash translation layer (FTL) in combination with a traditional block-oriented file system (Lu et al. 2013) and a log-structured embedded flash file systems with an MTD (David 2003). An MTD is a Linux's software abstraction dedicated for non-volatile memory (Corbet et al. 2005).

SYSTOR '15, May 26–28, 2015, Haifa, Israel.

http://dx.doi.org/10.1145/2757667.2757685

To utilize flash memory devices in a traditional block-oriented file system, an FTL is used in a solid-state drive (SSD) to hide the limitations of flash memory (see Section 2.1) and expose to its upper layer as a virtual block device like a hard disk drive (HDD). An SSD is organized hierarchically, which offers multi-level parallelism (Hu et al. 2011). The FTL that runs on a powerful controller manages multiple flash memory devices in a parallel way so that its performance is maximized (Hu et al. 2011). In addition to address mapping, an FTL also performs garbage collection and wear-leveling (Shin et al. 2009). Still, an FTL could be inefficient because the block-oriented file system is inherently unaware of the unique characteristics of flash memory. Meanwhile, an FTL does not have a complete knowledge of the file system's structures (Engel and Mertens 2005).

To remove the inherent restrictions of conventional block-oriented file systems and maintain a lower cost, developing a dedicated flash file system that can operate directly on raw flash devices becomes a better solution for embedded systems (Engel and Mertens 2005). Currently, the flash controller and its underlying multiple raw flash devices are abstracted as one MTD above which a flash file system is attached. Compared with the FTL-base SSD option, a flash file system is fully aware of the characteristics of flash memory. Thus, it can efficiently perform garbage collection and wear-leveling (Shin et al. 2009).

An MTD-based flash file system is a typical choice for embedded applications because: (1) Since the functionalities of an FTL is moved to the file system layer, the flash controller is much simpler than that of an FTL-based SSD. A simpler flash controller incurs a lower hardware cost, which is essential to a resource-limited and cost-sensitive embedded system (Jung et al. 2008; Kim et al. 2009); (2) Flash file systems can address the inherent constraints of flash (e.g., the *wear out* issue) more efficiently as they directly control raw flash memory devices (Manning 2004; Woodhouse 2001); (3) Users can have a complete control as they can revise file systems whenever needed (Hunter 2008). On the other hand, since FTLs are normally manufacturers' secrets, users can do little on them when requirements cannot be met.

Numerous embedded flash file systems (Hunter 2008; Jung et al. 2008; Kim et al. 2009; Manning 2004; Woodhouse 2001) have been proposed in the literature. However, only JFFS2 (Journaling Flash File System version 2) (Woodhouse 2001), YAFFS2 (Yet Another Flash File System version 2) (Manning 2004), and UBIFS (Unsorted Block Image File System) (Hunter 2008) have been widely used (Kang and Miller 2009). Although existing embedded flash storage systems are effective for traditional embedded applications, they are becoming increasingly inadequate for emerging data-intensive embedded applications due to their incompetent performance and inadequate level of reliability. For example, the read and write throughput of UBIFS are around 10 MB/s on a typical embedded platform (Homma 2009). However, the data volume created by emerging mobile applications per second is far more than that. Take live sports broadcast (Postley 2012) for example. Each pair of 3-D cameras generates several hundreds megabytes video and audio data per second (Postley 2012). One might think that the collected data could be shelved on a cloud via the Internet. However, a reliable wireless networking connection cannot be always guaranteed, and thus, storing and processing a large amount data locally are still demanded. Similarly, wireless healthcare also requires a reliable and high-performance local data processing capability because it monitors patients in real-time (Smith 2011). After comprehensively examining the existing flash file systems, we found that none of them is fundamentally designed to utilize the multi-level parallelisms presented by flash memory devices. To the best of our knowledge, all existing embedded flash file systems can support only one MTD, which communicates with one or multiple underlying flash memory devices. The single-MTD architecture has its advantages. It exposes an unified logical address space to a flash file system while hiding multiple raw devices so that the flash file system design can be simplified. Nevertheless, it also has some disadvantages. Since only after the current request is completed can next one be issued to the single MTD, the multiple flash memory devices under one MTD cannot work in parallel. Besides, unlike an FTL, an MTD has no intelligence to utilize the parallelism provided by a flash controller (Hu et al. 2011). To exploit the device-level parallelism in an embedded flash storage system, one obvious approach is introducing an FTL-like software module running on top of a powerful controller. However, this approach will increase hardware cost, which may not be feasible for cost-sensitive embedded applications.

In order to maintain a low hardware cost while utilizing device-level parallelism, in this paper we propose an MTD-array based embedded flash storage system framework called MA (MTD array). It abstracts each raw flash device as an independent MTD to form an array of MTDs. The MA framework enables multiple MTD devices to communicate with a flash file system concurrently so that the device-level parallelism can be exploited at the flash file system layer. Besides, it provides an opportunity for these MTD devices to protect each other's data through a data redundancy scheme (e.g., parity data, etc.). Consequently, data reliability could be further improved. Although MA increases the complexity of a file system, our experimental results show that it can improve performance without introducing any extra hardware cost. The performance gains justify the additional software complexity. We integrate UBIFS into the MA framework in Ubuntu 13.04 with 3.8.0 Linux kernel. The MA-enabled UBIFS flash file system is called MA-UBIFS. Next, we comprehensively evaluate the performance of MA-UBIFS with both micro-benchmarks and real-world traces. Experimental results show that when 4 flash devices are abstracted as 4 MTDs, MA-UBIFS improves I/O performance by up to 1.9X while its mounting time is less than 120 ms and RAM usage is smaller than 2.5 MB. For data-intensive workloads like *CameraVideo*, MA-UBIFS reduces mean response time by up to 71.6%. Hardware evaluation results show that on average it improves write and read throughput in a 2-MTD case by 55.2% and 84%, respectively.

The remainder of this paper is organized as follows. Background is discussed in the next section. Section 3 describes the design of the MA framework and the implementation of MA-UBIFS. Section 4 evaluates MA-UBIFS with micro-benchmarks. Experiments results with five real-world applications are illustrated in Section 5. Section 6 evaluates MA-UBIFS on a hardware platform. Section 7 concludes the paper with a summary and future direction.

2. Background and Related Work

2.1 Flash Memory

According to the number of bits stored in a memory cell, flash memory is categorized into single-level cell (SLC), multi-level cell (MLC), and triple-level cell (TLC) flash (Agrawal et al. 2008). While each cell of an SLC stores only one bit of information, an MLC cell stores 2 bits of data to increase the capacity while lowering the cost per gigabyte. Each TLC cell stores 3 bits (Agrawal et al. 2008). Flash memory offers three basic operations: write, read, and erase (Agrawal et al. 2008). Read and write are carried out at the page level, whereas erase can be conducted only at block granularity. One block is composed of an array of pages (e.g., 256 pages) with each page being of a particular size (e.g., 4 KB). Erase is a time-consuming process (e.g., 2 ms) compared with read (e.g., 20 μs) and write (e.g., 900 μs) (Grupp et al. 2009). Flash memory possesses a few "awkward" characteristics that an HDD does not have. First of all, it does not allow in-place updates as a write operation can only change bits from 1 to 0. An erase operation changes all bits of a block to 1 and a block must be erased before being programmed (Wang et al. 2014). The out-of-place update enforces flash memory to utilize a garbage collection mechanism to reclaim invalid pages within a block. Next, the lifetime of a block is limited by the number of P/E (program/erase) cycles on it. The P/E

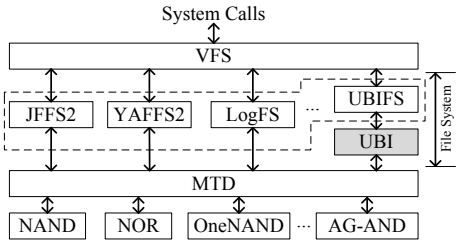

Figure 1. Current MTD subsystem in Linux kernel.

cycle limitation brings the need for a wear-leveling scheme, which ensures that all blocks in a flash memory device are worn out evenly in order to prolong device's lifetime.

2.2 Linux Flash File Systems

Linux is one of the most popular operating systems for an embedded system. It supports three types of devices: block devices, character devices and network devices. Since flash memory do not match the description of any of them, a special device type that exhibits flash characteristics is created: MTD. It is a new type of device file in Linux for interacting with flash memory (David 2003). Figure 1 illustrates the architecture of existing Linux based embedded flash storage system. MTD abstracts the hardware and hides different characteristics of various types of flash memory devices. Above MTD, a flash file system such as JFFS2 is built to manage data and provide interfaces to virtual file system (VFS) layer (see Figure 1). Although multiple raw flash devices may exist under a flash controller, present flash file systems can only see one MTD (David 2003).

Among the four representative flash file systems shown in Figure 1, UBIFS delivers the best performance in terms of I/O throughput and mounting time. Besides, adding an UBI (Unsorted Block Image) layer can largely reduce the complexity of a flash file system because it takes care of volume management. Therefore, we choose UBIFS as an example file system to demonstrate the strength of MA. UBIFS employs two copies of master node and journal replay to enhance its power failure tolerance (Hunter 2008). When it is mounted, UBIFS assumes that the on-flash index is always out-of-date (e.g., the index may be broken due to a power failure). In order to bring it to up-to-date, UBIFS reads all the leaf nodes in the journal and get them reindexed. MA-UBIFS directly borrows the journal replay method in UBIFS to tolerate a power failure.

3. The MA Framework

In this section, we first present an MTD-array based framework. Next, we incorporate UBIFS into the MA framework to design and implement a new flash file system MA-UBIFS.

3.1 MTD-Array Based Framework

After analyzing several prevalent open-source embedded flash storage systems, we propose two MTD-array based

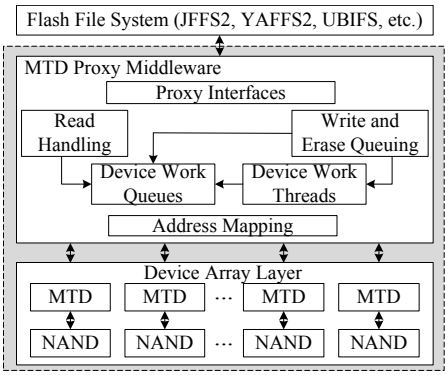

Figure 2. MTD proxy middleware.

flash storage system framework design schemes: MTD proxy middleware and MTD-array storage hierarchy. Both schemes require reconstructions of hardware platforms and software stacks. In terms of hardware, they all need a new flash controller, which can communicate with multiple flash memory devices in parallel. Unlike existing flash controllers, the controller should be able to expose its underlying multiple flash devices (or multiple dies in a single device) to upper layers so that they can be abstracted as an array of MTDs. In our design, each flash memory device is abstracted as one MTD. Multiple identical raw flash memory devices are organized in an array format. From the perspective of Linux kernel, multiple MTD devices with each having the same configuration are presented (see Figure 2 and 3).

In the first scheme, a software interface layer called MTD proxy middleware is designed, which sits between an MTD-array and a traditional flash file system as shown in Figure 2. The MTD proxy middleware enables an existing flash file system to communicate with an array of MTD devices. The MTD proxy middleware has the following components: (1) an address translation mechanism called *address mapping*, which can translate an aggregate device address to an underlying MTD device address; (2) an *operation queuing* method that chooses an underlying MTD device, and then inserts requests into its work queue; (3) multiple *work threads* (each attaching to a particular MTD device), each of which reads requests out of the work queue of the corresponding device in a first-in-first-out manner, and then calls the MTD functions (i.e., read, write, and erase) of the device to serve the requests; and (4) a *proxy interface* module, which emulates the interface of an MTD so that a flash file system can directly work on it. The MTD proxy middleware exploits the parallelism among multiple MTD devices by splitting a request into multiple sub-requests, which can then be served by different MTD devices concurrently. The middleware is independent of existing flash file systems. Therefore, it incurs no modifications of existing flash file systems, which is an attractive feature of this design scheme. However, it also possesses an obvious disadvantage. Existing flash file systems are designed on top of one MTD by default, and

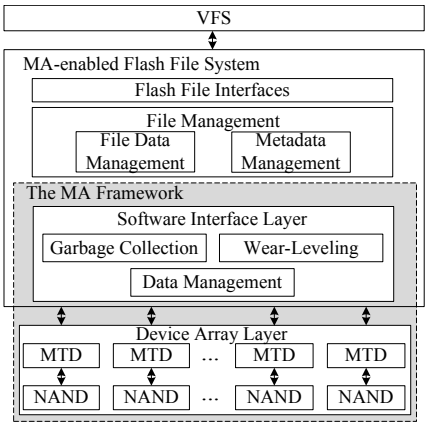

Figure 3. MTD-array storage hierarchy.

thus, can only process requests sequentially. Only after the response of the current request has been returned can next request be issued. Therefore, they cannot fully exploit the parallelism among multiple MTD devices.

The second scheme is to redesign an MTD-array storage system in a hierarchical way. Since a flash file system has a full control on data management and flash devices, the only feasible way to fully utilize the underlying MTD array is to redesign software stacks as shown in Figure 3. The second design scheme named MTD-array storage hierarchy can deliver a potentially higher performance compared with the first scheme as the flash file system is redesigned to employ parallelism. Figure 3 illustrates the major components of this scheme. From bottom to top, each raw flash memory device is connected to the new flash file system through an MTD. A software interface module provides an interface to communicate with the MTD array. Besides, the serial working flows in original file system are revised to cooperate with the multiple MTD devices. Compared with the first scheme, it can fully employ the parallelism among multiple MTD devices not only by splitting requests but also by issuing multiple requests concurrently. Since this scheme can fully control the MTD array, we adopt it as the MA framework. The software interface layer of MA contains three major components: (1) an *array-oriented wear-leveling* scheme; (2) a *centralized garbage collection* method; and (3) a new *data management module* for the multiple MTD devices. In the file management module, only device related working flows are revised. The original data structures and management schemes are kept to simplify the complexity of MA.

3.2 MA-UBIFS

To demonstrate the effectiveness of MA, we integrate UBIFS into it to develop a new MA-enabled flash file system called MA-UBIFS. Note that other flash file systems like JFFS2 can also be integrated into MA. We newly added around 3,000 lines of C code for the MA framework and revised almost all UBIFS source files in 3.8.0 Linux kernel.

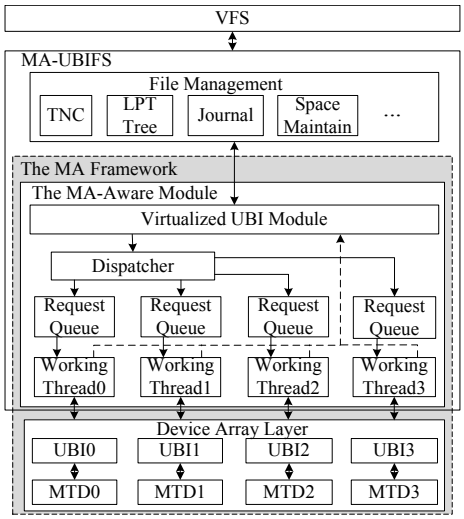

Figure 4. The architecture of MA-UBIFS.

3.2.1 Virtualized UBI and Address Re-mapping

A software layer called MA-aware module is integrated into the original file system as shown in Figure 4. Multiple MTD devices are first translated into UBI devices. Usually, one UBI device can have several volumes. In our design, each UBI device is only formatted into one volume so that multiple volumes mounted in UBIFS have fully parallel capability. Two components are designed in the MA-aware module: a virtualized UBI module and multiple MTD-oriented working flows. The virtualized UBI component has two major tasks: (1) assisting MA-UBIFS to initialize data structures with multiple UBI devices instead of one; (2) re-mapping those multiple UBI devices into a uniform address space so that parallel accesses can be easily achieved. Figure 5 shows the address mapping scheme used in MA-UBIFS.

Since MA-UBIFS mounts on multiple UBI devices, the data structure construction process must be redesigned to correctly use all of them. Four important parameters in MA-UBIFS are used in a mounting process: (1) the total number of LEBs (logical erase blocks) contained; (2) the size of each LEB; (3) the minimum size of one I/O request; and (4) the maximum write size. As a virtualized UBI module re-maps multiple identical UBI devices to a uniform address space, in our design the total number of LEBs in a virtualized UBI is the same as the number in a single UBI device used by MA-UBIFS. The size of an LEB in a virtualized UBI is enlarged to that of p LEBs if totally p UBI devices are used. Taking the Figure 5 as an example, assuming that four identical UBI devices are used and each one has m LEBs with each containing n pages. In the virtualized UBI module, the total number of LEBs is m and the size of a single LEB is augmented into $4n$. The minimum size of one I/O request in a virtualized UBI is equal to a flash page size. The maximum write size equals to the write buffer size used in UBIFS. After the write buffer is full, UBIFS flushes all the buffered

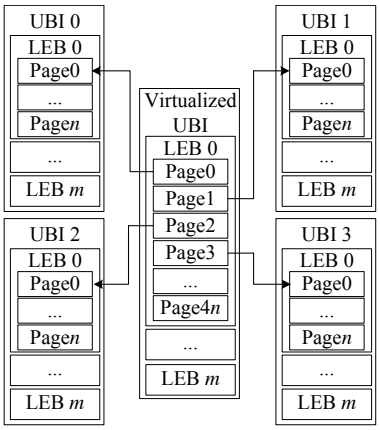

Figure 5. Address mapping scheme in MA-UBIFS.

data onto flash memory. In UBIFS, the write buffer size is set to the minimum I/O size (i.e. 1-page size). In contrast, MA-UBIFS uses $p \times minimum\_IO\_size$ (i.e. p-page size) as its write buffer size if there are p UBI devices. For example, in Figure 5 the maximum write size for this virtualized UBI is 4-page size. By employing this design, one operation of flushing write buffer can be split into four small requests with each containing 1-page size data. Next, the four requests can be issued to four UBI devices concurrently. In MA-UBIFS, we use a round-robin address mapping scheme to simplify our design. Equations shown as below are used to translate an LEB number and a page number in a virtualized UBI to four physical UBIs:

$$physical\_UBI\_num = page\_num\%num\_of\_UBIs,$$
$$physical\_page\_num = page\_num/num\_of\_UBIs.$$

For example, when the data in the write buffer is flushed to the virtualized UBI's LEB0 with an offset of page 0 as shown in Figure 5, the four sequential 1-page size data will be sent to the page 1 of LEB0 in the 4 UBI devices, respectively.

3.2.2 Request Dispatcher and Working Threads

MA-UBIFS exploits parallelism among multiple UBI devices. Each request in MA-UBIFS is split into several sub-requests according to its request size and sent to different UBIs. To achieve that MA-UBIFS uses a request dispatcher with a dedicated working thread for each UBI device. The main task for the request dispatcher is to identify sub-requests received from a virtualized UBI after applying the address mapping scheme and then to add each of them into the corresponding request queue. The request working thread that belongs to a particular UBI device fetches a sub-request in its request queue and then sends it to flash memory through UBI interfaces. The background working threads are created during mounting period and assigned to its associated UBI device respectively (see Figure 4). Take the write buffer flushing process described in Section 3.2.1 as an example. A one 4-page size write request is split into four 1-page size sub-requests, and then the request dispatcher sends each of the four sub-requests to its corresponding request queue in its associated UBI device. Each of the four working threads retrieves the sub-requests from its associated request queue and processes it independently. After the four requests are processed, a response will be directly sent back to the virtualized UBI module.

There are two main rationales behind the multi-thread design. Firstly, the response time of flash memory operations is significantly larger than the thread context switching time. Thus, the multi-thread design can take the advantage of interleaving operations between multiple devices. Secondly, embedded processors are becoming more powerful. For instance, Qualcomm announced a processor named Snapdragon with 4 cores (Qualcomm 2013). Hence, the multi-thread employed design is suited for these new platforms.

3.2.3 Asynchronous TNC Upgrade

UBIFS uses a cached index tree (i.e., TNC) to improve its performance. When a file is written/updated, its data are first stored in the journal space and UBIFS only modifies the TNC in RAM. UBIFS writes the whole TNC onto flash memory when the journal space is full or a commit command is issued. By using this strategy, UBIFS avoids writing the index onto flash frequently. During a mounting process, UBIFS does not fetch the index information from flash memory to RAM. Instead, it loads the index nodes when they are needed. Almost every file operation needs to read or modify its content, which makes the TNC critical. In UBIFS, for data safety only one TNC query is allowed to enter this critical section at one time. However, this approach serializes concurrent requests, which impedes the performance. After analyzing the TNC management, we find that loading on-flash index only happens once when it is first accessed and after that read operations do not change any content of TNC. Therefore, we believe that the critical section of TNC can be re-defined so that asynchronous accesses become possible.

In MA-UBIFS, a read/write semaphore is introduced to manage the access of the TNC. Hence, multiple reads can enter into this exclusive section at the same time. Meanwhile, a signal and a working queue are introduced to the index loading process. The first operation that loads the necessary index from the flash will set the signal before retrieving the data. After this, other operations accessing the same index will have to wait in the working queue until the index is loaded. In this way, while multiple read operations can concurrently access TNC, only one write operation can lock the entire TNC section at one time.

4. Micro-Benchmark Evaluation

4.1 Experimental Setup

Evaluation environment is set up on a 3.1GHz Intel Core i5 machine with 8 GB of RAM. Operating system is Ubuntu 13.04 with 3.8.0 Linux kernel.

Table 1. Flash memory major characteristics

Type	SLC	MLC	TLC
Page size (KB)	2	2	4
Read time (μs)	25	25	75
Program time (μs)	200	600	1,300
Erase time (ms)	1.5	2	4

Table 2. Flash memory and UBI volume size

Configurations	Conf_1	Conf_2	Conf_3
Device num.	1	2	4
MTD capacity (MB)	1,024	512	256
UBI Volume (MB)	986	493	246

Modifications of NANDsim: To the best of our knowledge, there is no publicly available flash controller that is capable of exposing its underlying raw flash devices or dies to upper layers so that they can be abstracted as an array of MTDs in Linux kernel. Hence, we use NANDsim (Linux 2013), a widely used NAND flash simulator (Linux 2013; Kang and Miller 2009), to simulate the hardware part of the MA framework in our micro-benchmark evaluations. It simulates NAND flash memory in RAM, which can imitate the behavior of different types of NAND chips and report wear out statistics (Linux 2013). The existing version of NANDsim, however, can emulate only one raw flash memory. In order to simulate an array of flash memory devices, we modify its global data structure and initialization process to repeatedly build a group of flash memory devices in RAM so that multiple MTDs can be instanced. The simulated flash memory devices hold the same characteristics and have identical capacity. Since the total capacity of our experimental computer is only 8 GB, at most 4 flash memory devices with each having 1 GB capacity can be simulated.

Configurations of Flash Memory Devices: To comprehensively understand the impacts of different flash memory configurations on MA-UBIFS, we choose three types of flash memory devices with different I/O features, which range from SLC to TLC. Table 1 summaries the parameters of three different flash memory types used in experiments.

The number of simulated raw flash memory devices is changed from 1 to 4 to measure the scalability of MA-UBIFS. The 1-MTD scenario represents current single-MTD architecture, which serves as a baseline in performance comparisons. Table 2 illustrates the capacity of each device in experiments. The total capacity of flash devices determines the number of garbage collections happened in flash memory (Bez et al. 2003), which can affect the overall performance of a flash file system. Hence, throughout all the experiments, the total capacity of flash memory devices used by MA-UBIFS remains to 1 GB to make the comparisons fair (see Table 2). For each MTD device, we create only one UBI volume. The size of UBI volume is equal to the maximum allowable size on each MTD device as shown in Table 2. In the rest of this paper, the terms UBI devices and UBI volumes are interchangeable because in our implementation only one UBI volume is created on one UBI device.

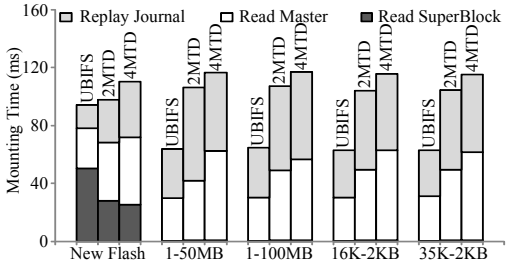

Figure 6. Mounting time; horizontal axis is the number of files contained in file system. 16K-2KB means the file system has 16,000 2KB files.

4.2 Mounting Time

We measure the mounting time in five scenarios: 1) mount the file system on an entirely new UBI volume; 2) mount the file system on an UBI volume that contains only one 50MB file; 3) mount the file system on an UBI volume that has one 100MB file; 4) mount the file system on an UBI volume that stores 16,000 2KB-size files; and 5) mount the file system on an UBI volume that contains 35,000 2KB-size files. The five cases can be categorized into two camps: *new mount* and *used mount*. While case 1 is a new mount, all the rest belong to used mount. Case 1 happens when a new flash memory device is formatted. In this case, all the necessary data structures (e.g., super block and master node) have to be created and the entire device is formatted into a UBI volume organization. In the other four cases, the necessary data structures and metadata already have been created and can be directly read from flash memory devices. Case 2 and 3 simulate applications like 3-D cameras that only a few number of files with big sizes are stored (Park and Park 2011), whereas case 4 and 5 emulate applications like web cache and personal documents. The mounting process for MA-UBIFS can be divided into three steps: reading superblock, reading master node, and replaying journal. Master node stores the position of all on-flash structures that are not at fixed logical positions (Hunter 2008). The existence of the journal is to improve the efficiency of index updating because it can reduce the number of index updating processes (Hunter 2008). During a mounting process, the leaf nodes in the journal must be read and re-indexed to make the on-flash index up-to-date. In our experiments, the journal size is set to 5% of total capacity (Hunter 2008).

Figure 6 shows the mounting time of MA-UBIFS and UBIFS in different configurations. Three conclusions can be directly drawn from the experimental results. Firstly, UBIFS outperforms MA-UBIFS constantly in terms of mounting time, especially in used mount situations. Secondly, the time

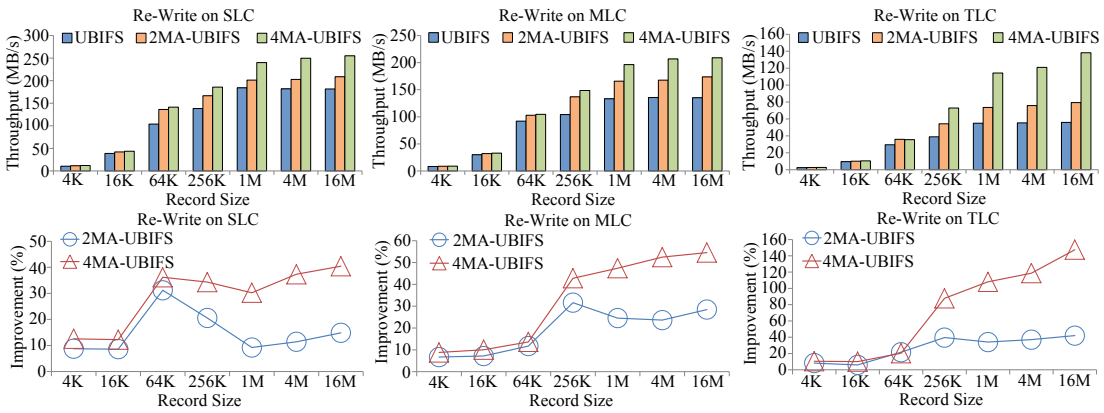

Figure 7. Re-write throughput on three flash memory types.

of a new mount for UBIFS is obviously longer than that of used mount. On the contrary, the time of a new mount for MA-UBIFS is similar to that of used mount. Thirdly, as the number of UBI devices increases the mounting time of MA-UBIFS becomes longer.

For MA-UBIFS, the mounting time of a new mount with 2 and 4 UBI devices is 3.8% and 17% worse than that of UBIFS, respectively. The increased time comes from the steps of reading master node and replaying journal. In these two steps, necessary data are read from flash memory devices to construct the file system data structures in RAM. Since the read operation of flash memory is relatively fast, the overhead of multi-thread management and context switching between threads become the dominant cost. Therefore, the time of these two steps is involuntarily increased. On the contrary, in the reading superblock step of new mount case, the time of creating new file system in MA-UBIFS is shorter than that of UBIFS. The reason is that write operation of flash memory is much slower than read operation (in this experiment is 17 times) so that MA-UBIFS largely decreases the time of writing file system related data onto the flash memory devices. The increased time in the last two steps of mounting process (i.e., reading master node and replaying journal) exceeds the decreased time of creating a new file system process. Hence, the overall mounting time of MA-UBIFS is longer than that of UBIFS. When the number of UBI devices increases the multi-thread management incurs a higher overhead. It increases the mounting time of MA-UBIFS while the number of mounted UBI devices escalates. In the used mount cases, the replaying journal step spends more time to read the data stored in the journal space. The time increment is roughly equal to the time of creating a new file system. Thus, the mounting time of MA-UBIFS for new mount and used mount does not change obviously.

4.3 I/O Performance

In this section, the I/O performance of MA-UBIFS is evaluated. Three different flash memory types (i.e., SLC, MLC, and TLC) and two configurations with different number of UBI devices are used. A widely used file system benchmark, IOzone (Norcott and Capps 2006), is utilized in our experiments to measure the performance of MA-UBIFS. The benchmark executes file I/O in record size ranging from 4K bytes to 16M bytes. To ensure every write to go completely to a flash memory device before returning to the benchmark, the O_SYNC flag is specified in evaluation. The file system is un-mounted and re-mounted between evaluations to eliminate the impacts of buffer cache on overall performance.

The MA framework can improve both read and write performance. Since the read operation is about 8 to 17 times faster than write, the read throughput improvement of MA-UBIFS over UBIFS is not substantial. Besides, UBIFS employs a cache to improve I/O performance, which makes the enhancement of read throughput measured by IOzone unnoticeable as most read requests are directly served by the cache. In our experiments, we find that the difference between MA-UBIFS and UBIFS in read performance is no larger than 4%. Therefore, due to space limit in this section we only present the performance of MA-UBIFS in terms of various writes. The read improvements on a hardware platform with random data will be illustrated in Section 6.

Figure 7 and Figure 8 illustrate the write throughput of UBIFS and MA-UBIFS in different record sizes ranging from 4KB to 16MB. The 2MA-UBIFS and 4MA-UBIFS represent MA-UBIFS with 2 MTDs and 4 MTDs, respectively. It is clear that for each type of flash memory (i.e., different access speeds) the trend of throughput improvement is the same. In all different record size cases, MA-UBIFS consistently outperforms UBIFS in both 2 UBI devices and 4 UBI devices configurations. Meanwhile, the write throughput becomes larger as the record size increases. The maximum throughputs of the three write types are all in TLC cases when 4 UBIs are used. The maximum throughput under re-write, sequential write, and random write are 138.3 MB/s, 116.1 MB/s, and 100.4MB/s, respectively. Both of the sequential write and random write are new write where related metadata must also be constructed and written to flash

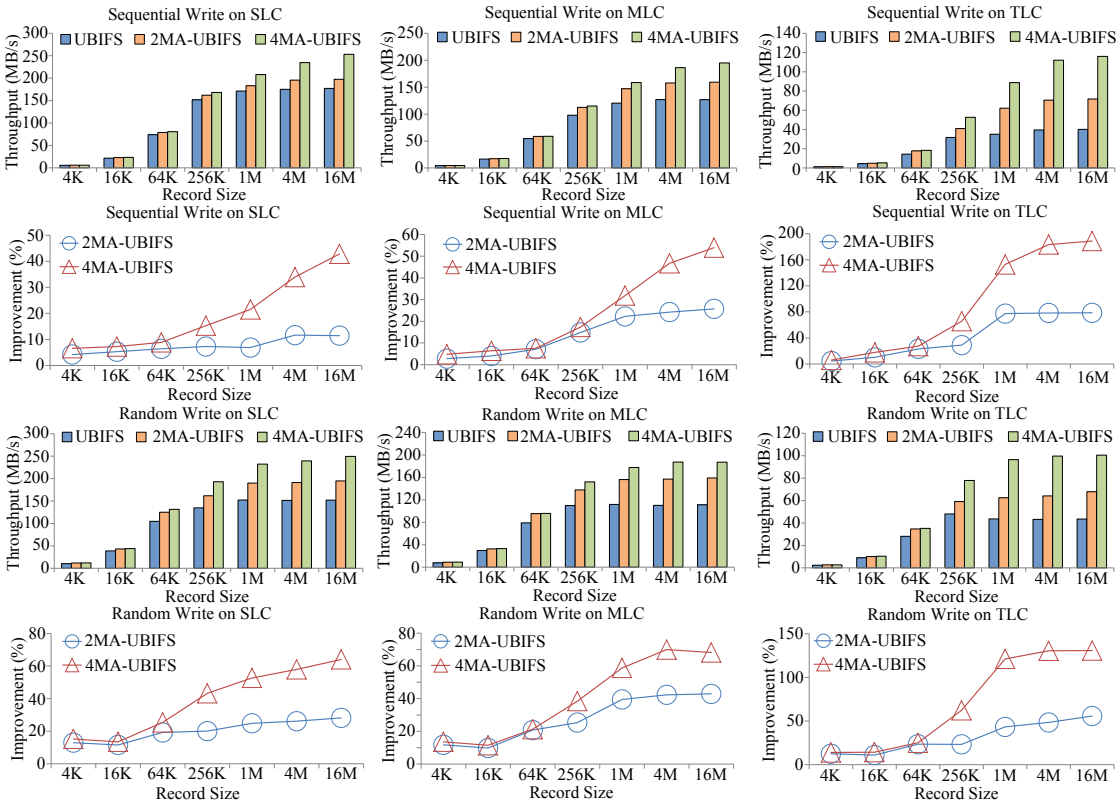

Figure 8. Sequential write and random write throughput on three flash memory types.

memory. Re-write, on the other hand, does not need to create and change the metadata. Hence, write operations can be served faster. When using TLC flash memory all three different write throughput improvements of MA-UBIFS under 4KB record size is roughly 4.5% when 2 UBIs are used. This improvement slightly increases to 6% under 4 UBI devices configuration. When the record size is increased to 16MB the improvement of sequential write throughput for MA-UBIFS is substantially promoted to 78.7% and 188.7% by using 2 UBI devices and 4 UBI devices, respectively. When 4 UBIs are employed, the maximum improvement of random write and re-write are 130% and 147%, respectively. In small record size cases, parallelism between UBI devices cannot be fully utilized because requests can only be served by one or a few number of the mounted UBI devices. Thus, adverse impacts from the later aspect neutralize the improvements gained from the first aspect, which results in a small overall improvement of throughput. On the other hand, when a big record size is used parallelism can be fully utilized. It is also obvious that the more UBI devices a file system uses, the larger throughput improvement that MA-UBIFS can achieve. When the number of UBI devices is increased from 2 to 4, the write performance under 16MB record size is boosted by 61.6% on average. Intuitively, more UBI devices can serve a request with larger data size by splitting the request into multiple small requests and concurrently processing them. Because of hardware limitation, the maximum number of UBI devices in experiments is set up to 4. We expect more performance improvements if more than 4 UBIs are used. The improvement trend lines in Figure 7 and 8 show that the improvement of MA-UBIFS using 4 UBI devices increases faster than that of MA-UBIFS using only 2 devices as the record size becomes larger. The performance improvements of MA-UBIFS are lower than expected. For example, MA-UBIFS cannot achieve a near 4-time improvement when 4 UBIs are used. There are two main reasons: (1) the overhead of managing multi-threads enlarges response times; (2) the frequently accessed TNC tree (see Section 3.2.3) cannot be concurrently accessed by new writes.

As shown in Table 1, three types of flash memory devices are simulated in our experiments. The main difference among these flash memory devices is the operation latency. SLC has the fastest operation speed while TLC is the slowest one. The general trend of performance improvement is that the bigger write latency the flash memory has the larger improvement the file system can achieve. Meanwhile, as the record size increases the performance improvement continuously goes up. The maximum performance improvement of sequential write between SLC and TLC is 5.6X under 4MB record size case. The reason behind this is that the latency of one request is determined by two factors. One is software stack latency and the other one is hardware latency.

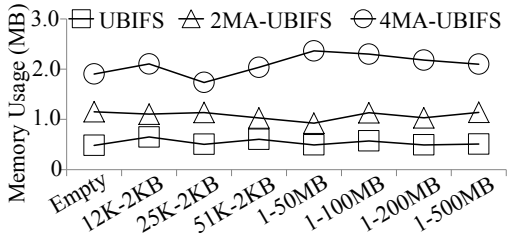

Figure 9. Memory usage; 12K-2KB represents a file system has 12,000 files with each 2KB.

The software stack latency for MA-UBIFS is constant for all three flash memory types. While the hardware latency varies when different types of flash memory are used. TLC with 1,300μs write latency has the largest hardware latency proportion in the overall request latency. Therefore, the improvement made by utilizing parallelism becomes more noticeable when TLC is used. For random write, the overall write performance improvement is not as significant as that of sequential write. The maximum improvement is 131% by using 4 TLC flash memory based UBI devices under 16MB record size. The minimum improvement of MA-UBIFS is 8.3% using 2 UBI devices under 8KB record size on the MLC flash. Under the same configuration, the improvement of sequential write is only 2.8%. When the record size is below 1MB, the improvements of re-write on three different flash memory devices are irregular.

4.4 RAM Memory Usage

We compare the RAM usage before and after MA-UBIFS is mounted. Figure 9 shows the RAM usage of MA-UBIFS and UBIFS with respect to different file sizes. Two conclusions can be drawn. First, for both file systems the memory usage are roughly independent of its contents. The total memory usage keeps the same under different sizes of contents. Secondly, the more UBI devices that MA-UBIFS uses, the more memory it consumes. The maximum memory usage is 2,090 KB on average when 4 UBI devices are used. When MA-UBIFS uses 2 UBI devices it consumes 1,081 KB RAM. UBIFS only uses roughly 500 KB memory in all cases. MA-UBIFS maintains multiple UBI devices. For each UBI device, an associated independent kernel thread is created to serve requests. The increased memory usage is consumed by those new kernel threads and their related data structures. According to the experimental results, each thread and its related data structures use only 327 KB RAM on average.

5. Real Application Evaluation

5.1 Real-World Mobile Appications

To collect real-world traces from a mobile device, we build an I/O monitor tool that is running in the Android 4.4 on a Nexus 5 smartphone (Google 2013). The Nexus 5 is designed by Google, which maintains and develops the Android OS. Thus, some system biases such as background services introduced by venders can be avoided to eliminate "noises" from traces. We customize the original kernel of Android 4.4 to add the I/O system monitor at the file system level. The monitor records systems calls such as *fopen*, *fread*, and *fwrite* while running a certain application on the smartphone. When an I/O system call is invoked the monitor records its parameters such as arriving time and I/O size into a trace file, which can be replayed later. The Nexus 5 is running with its default configuration during our experiments.

Table 3. Real-world traces

Application	Description
AngryBird	Playing angry bird for 1 hour.
GoogleMap	Navigating route for 30 mins.
Moive	Watching movie clips for 15 mins.
CameraVideo	Recording a 3-min video clip.
WebBrowsing	Surfing webs for 15 mins.

We select five common smartphone applications and record their I/O activities by using the I/O monitor tool. The five real-world traces are summarized in Table 3. They represent various types of workloads. For example, AngryBird and Movie are read-intensive traces, in which 96.9% and 87.4% requests are read requests, respectively. They read 2.79 GB and 110.3 MB data from flash memory, respectively. In contrast, the CameraVideo trace is write-dominant as 95.6% of its requests are writes. It writes 373.8 MB data into flash memory. GoogleMap and WebBrowsing have a similar amount of input and output data. Their read request percentages are 63.8% and 42.3%, respectively. An I/O replayer reads each trace line by line and then generates file system I/Os, which are fed into MA-UBIFS. The parameters of the simulated MLC flash devices are shown in Table 1. Table 2 shows the configurations of MA-UBIFS.

5.2 Experimental Results

Figure 10 compares the mean response times (MRTs). All values are normalized to UBIFS' MRTs. For data-intensive applications such as Movie MA-UBIFS noticeably reduces their MRTs. As the number of MTD devices increases, MA-UBIFS shows a consistent performance improvement. Under Movie, CameraVideo, and AngryBird 4MA-UBIFS reduces MRT by 71.6%, 62.9%, and 44.6%, respectively. For GoogleMap, the performance improvement of 2MA-UBIFS and 4MA-UBIFS are 16.4% and 18.2%, respectively. For WebBrowsing, however, we see a MRT increase in 2MA-UBIFS. The performance of MA-UBIFS exhibits a small fluctuation under WebBrowsing due to the following two reasons. Firstly, the majority of requests in WebBrowsing are small (average size is 1.8 KB). Hence, the performance improvement cannot compensate the overhead caused by managing multiple threads in MA-UBIFS. Secondly, the average request inter-arrival time is 14.5 ms, which is much longer than 1-page read/write latencies (see Table 1). Therefore, the parallelism provided by MA-UBIFS cannot be exploited.

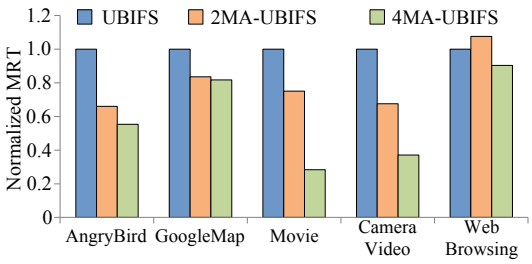

Figure 10. Results from real-world traces.

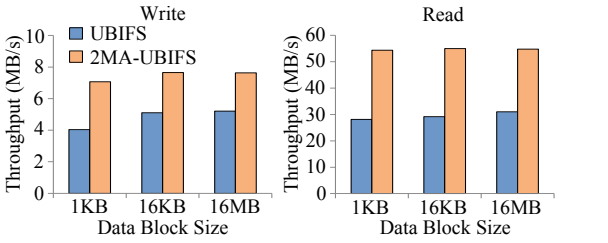

Figure 11. Results from the hardware platform.

6. Hardware Evaluation

6.1 Evaluation Platform

The evaluation platform consists of a Xilinx Xupv5-Lx110t FPGA board (XILINX 2011) and a flash daughter board (Bunker et al. 2012). The flash daughter board has two independent channels with each connecting to a raw flash device. An embedded system including an Microblaze processor and an ONFI 2.0 compatible flash controller are implemented in the FPGA. The flash controller provides two independent buses so that each of the two channels on the daughter board has its dedicated data path. Besides, it has a special register, through which an MTD driver can know the number of underlying flash devices and separately control them. Further, an embedded Linux with kernel version 3.0 is installed on top of the system. An MTD driver for the flash controller is implemented, which abstracts each raw flash device as an independent MTD device. On each channel, one MLC flash device (Micron 2013) is attached. Its typical read, programming, and erase latencies are 75 μs, 1,300 μs, and 3.8 ms, respectively. The size of each page is 8 KB and there are 256 pages in a block. To ensure data integrity, a software BCH (Bose-Chaudhuri-Hocquenghem) ECC code is used, which can correct 4-bit errors per 512 bytes of data.

6.2 Evaluation Results

The platform has limited hardware resources. Also, the embedded Linux operation system provides only a simplified library. Thus, popular file system benchmarks like IOzone cannot run on top of it. To evaluate the performance of MA-UBIFS on the embedded platform, we use a *bash* script that writes files to an MA-UBIFS mounted directory by copying data from */dev/random* using the *dd* command. The script reads files from the mounted directory to */dev/null* using *dd*. It can delete them later using the *rm* command. The script creates 128 20MB files to the file system with various data block sizes from 1 KB to 16 MB. In our experiments, UBIFS uses 1 MTD, whereas MA-UBIFS utilizes 2 MTDs.

Figure 11 shows the write and read performance of UBIFS and 2MA-UBIFS under different data block sizes. Under all data block sizes, 2MA-UBIFS exhibits a better performance. On average, 2MA-UBIFS improves write and read throughput by 55.2% and 84%, respectively. When the data block size enlarges, the write throughputs of UBIFS and 2MA-UBIFS both increase. While UBIFS improves write throughput by 29.1% as the data block size increases from 1 KB to 16 KB, 2MA-UBIFS only improves its performance by 7.9%. For read, the throughput of both UBIFS and 2MA-UBIFS almost keep unchanged when the data block size enlarges. This is because the main performance bottleneck of storage system on the embedded platform is the raw flash device I/O throughput, which is usually low (Micron 2013). The evaluation results of MA-UBIFS from the hardware platform demonstrate that the MA framework can noticeably improve the performance of storage system on real flash devices.

7. Conclusions

Existing flash file systems are not able to utilize device-level parallelism as they cannot "see" the multiple flash devices beneath a controller. In this research, we design, implement, and evaluate a new embedded flash storage system framework. In particular, we re-architect current single-MTD architecture to an MTD-array organization so that multiple flash memory devices each being abstracted as one MTD can work concurrently. Next, we design and implement MA-UBIFS in Linux kernel. It breaks the single-MTD device barrier so that it can access multiple flash memory devices in parallel to improve performance. We evaluate its performance on top of an array of simulated flash memory devices with micro-benchmarks and real-world traces. Lastly, a hardware platform is built to examine the effectiveness of MA-UBIFS on a real embedded system. Experimental results show that MA-UBIFS obviously outperforms UBIFS. The new flash file system including source code and its documents will be released to the public in the near future.

The MA framework provides an opportunity for an embedded flash file system to utilize a data redundancy scheme to enhance data reliability, which will be the future work of this research.

Acknowledgments

We would like to thank Steven Swanson's Non-volatile Systems Laboratory for providing us with the Ming II platform. Also, we would like to thank the anonymous reviewers for their constructive comments that improve this paper. This work is sponsored by the U.S. National Science Foundation under grant CNS-1320738.

References

N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX ATC2008*, 2008.

R. Bez, E. Camerlenghi, and A. Modelli. Introduction to flash memory. *Proceedings of the IEEE*, 91, 2003.

T. Bunker, M. Wei, and S. J. Swanson. Ming II: A flexible platform for NAND flash-based research. Technical report, Department of Computer Science and Engineering, University of California, San Diego, 2012.

J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

W. David. Memory technology device (mtd) subsystem for linux, http://www.linux-mtd.infradead.org/index.html, 2003.

J. Engel and R. Mertens. Logfs-finally a scalable flash file system. In *12th International Linux System Technology Conference*, 2005.

Google. Nesus 5 mobile device, 2013. URL `http://www.google.com/nexus/5/`.

L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.

T. Homma. Evaluation of flash file systems for large nand flash memory. In *CELF Embedded Linux Conference*, 2009.

Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.

A. Hunter. A brief introduction to the design of UBIFS, 2008.

D. Jung, J. Kim, J.-S. Kim, and J. Lee. Scaleffs: A scalable log-structured flash file system for mobile multimedia systems. *ACM TOMCCAP 2008*, 2008.

Y. Kang and E. L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 305–314. ACM, 2009.

H. Kim, Y. Won, and S. Kang. Embedded nand flash file system for mobile multimedia devices. *Consumer Electronics, IEEE Transactions on*, 55(2):545–552, 2009.

Linux. NAND simulator in Linux Kernel, 2013. URL `http://www.linux-mtd.infradead.org/faq/nand.html`.

L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, pages 31–44. USENIX Association, 2013.

C. Manning. YAFFS: Yet another flash file system, 2004.

Micron. MT29F64G08CBAAA, 2013. URL `http://www.micron.com/products/nand-flash/mlc-nand`.

W. D. Norcott and D. Capps. IOzone filesystem benchmark. *URL: www. iozone. org*, 55, 2006.

Y. Park and K. H. Park. High-performance scalable flash file system using virtual metadata storage with phase-change RAM. *Computers, IEEE Transactions on*, 60(3):321–334, 2011.

H. Postley. Sports: 3-d tv's toughest challenge. *Spectrum, IEEE*, 49(11):40–66, 2012.

Qualcomm. Qualcomm snapdragon processor, 2013. URL `http://www.qualcomm.com/snapdragon?source=google type=branded campaignid=snapdragon-exactadgroup= snapdragon-exact`.

J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 338–349. ACM, 2009.

J. M. Smith. The doctor will see you always. *Spectrum, IEEE*, 48 (10):56–62, 2011.

W. Wang, T. Xie, and D. Zhou. Understanding the impact of threshold voltage on mlc flash memory performance and reliability. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 201–210. ACM, 2014.

D. Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*, volume 2001, 2001.

XILINX. Ml507 evaluation platform user guide, 2011. URL `http://www.xilinx.com/support/documentation /boards_and_kits/ug347.pdf`.